In order to easier find the different function block libraries, they are separated by coloured paper sheets.

<table>
<thead>
<tr>
<th>Function block library 90 series</th>
<th>Colour of paper sheet at the beginning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>red</td>
</tr>
<tr>
<td>RCOM</td>
<td>green</td>
</tr>
<tr>
<td>ARCNET</td>
<td>orange</td>
</tr>
<tr>
<td>CS31 bus</td>
<td>blue</td>
</tr>
<tr>
<td>Communication</td>
<td>pink</td>
</tr>
<tr>
<td>SMC / Flash (data storage)</td>
<td>light green</td>
</tr>
<tr>
<td>System information</td>
<td>yellow</td>
</tr>
<tr>
<td>PROFIBUS</td>
<td>light blue</td>
</tr>
<tr>
<td>DeviceNet</td>
<td>light brown</td>
</tr>
<tr>
<td>Interbus</td>
<td>grey</td>
</tr>
<tr>
<td>CANopen</td>
<td>violett</td>
</tr>
<tr>
<td>Ethernet</td>
<td>red</td>
</tr>
</tbody>
</table>
Contents

1 Overview of the libraries for series 90 of 907 AC 1131 ........................................... 3
  1.1 Libraries of 907 AC 1131 in versions V4.0 and V4.1 ........................................... 3
  1.1.1 The libraries of 907 AC 1131 version V4.0 ...................................................... 3
  1.1.2 The libraries of 907 AC 1131 as of version V4.1 .............................................. 3
  1.1.3 Comparison of the libraries of versions V4.0 and as of version V4.1 ............... 4
  1.2 The libraries of the versions V4.2 and V4.3 ......................................................... 5
  1.3 The libraries of the version V5.0 ........................................................................ 6

2 The IEC operators and function blocks .................................................................. 7
  2.1 The operators in the 907 AC 1131 software ......................................................... 7
  2.1.1 Overview of the operators ............................................................................... 7
  2.1.2 Overview of possible type conversions .......................................................... 9
  2.2 The IEC standard library of the 907 AC 1131, 90 series .................................. 10
  2.3 IEC steps for sequential function chart ................................................................ 10

3 Function block libraries of the 90 series ............................................................... 11
  3.1 Base function block library 90 series ................................................................ 11
  3.1.1 Base library with all function blocks ............................................................. 11
  3.1.2 Generation of own libraries from Base_free_S90_V41.LIB ...................... 16
  3.1.3 Libraries generated from Base_S90_V41.LIB ........................................... 16
  3.2 RCOM/RCOM+ function block library, 90 series ............................................ 17
  3.3 ARCNET function block library, 90 series ....................................................... 17
  3.4 Function blocks for CS31 bus, 90 series ........................................................... 18
  3.5 Function blocks for serial communication, 90 series .................................... 18
  3.6 Function blocks for SMC and FLASH, 90 series ............................................. 19
  3.7 Function blocks for system information, 90 series .......................................... 19
  3.8 Library of coupler basic routines, 90 series ..................................................... 20
  3.9 PROFIBUS library, 90 series ............................................................................ 22
  3.9.1 PROFIBUS library, 90 series in versions V4.1 and V4.2 ......................... 22
  3.9.2 PROFIBUS library, 90 series as of version V4.3 ...................................... 22
3.10 DeviceNet library, 90 series ................................................................. 23
3.11 Interbus library, 90 series ................................................................. 24
3.12 CANopen library, 90 series .............................................................. 25
3.13 Ethernet library, 90 series ................................................................. 26

4 Conversion of 907 PC 331 blocks into 907 AC 1131, 90 series ................. 27
4.1 Converted blocks from 907 PC 331 to 907 AC 1131, 90 series ................. 27
4.2 Not converted 907 PC 331 blocks ......................................................... 30

5 Index ........................................................................................................ XXXI
1 Overview of the libraries for series 90 of 907 AC 1131

1.1 Libraries of 907 AC 1131 in versions V4.0 and V4.1

1.1.1 The libraries of 907 AC 1131 version V4.0

In version V4.0 of the software 907 AC 1131, the following libraries are included:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard.LIB</td>
<td>IEC standard library</td>
<td>extern/obj</td>
</tr>
<tr>
<td>iecsfc.LIB</td>
<td>IEC sequence steps</td>
<td>intern</td>
</tr>
<tr>
<td>ABB-BIB4.LIB</td>
<td>ABB function block library</td>
<td>extern</td>
</tr>
<tr>
<td>RCOM_V40.LIB</td>
<td>RCOM/RCOM+ function block library</td>
<td>extern/obj</td>
</tr>
<tr>
<td>CIF104.LIB</td>
<td>Coupler basic routines library</td>
<td>extern</td>
</tr>
<tr>
<td>PROFI_40.LIB</td>
<td>PROFIBUS library</td>
<td>intern</td>
</tr>
</tbody>
</table>

1.1.2 The libraries of 907 AC 1131 as of version V4.1

In order to get version V4.1, the libraries of the software were reworked and renamed. For distinction between the 40..50 series and the 90 series, the libraries for the 40..50 series are marked with 'S40' and the libraries for 90 series are marked with 'S90'. In addition, the software version of the library is inserted, here 'V41'. If the library is modified, these version identifiers will also be changed.

Furthermore, subject-oriented libraries were set up, e.g. for ARCNET, CS31 bus, serial communication, etc. In these libraries, there are some new function blocks and also blocks from the ABB-BIB4.LIB.

⚠️ Note:

The ABB-BIB4.LIB library can now and in future be used for taking over projects which were created with 907 PC 331. Therefore, this library is still installed in the library subdirectory of the programming software.

For compatibility purposes, the libraries of the version 4.0 are installed into the subdirectory 'Library/Library_V40'.

⚠️ Note:

It has to be kept in mind that function blocks must not exist twice!
For example, the library ABB-BIB4.LIB cannot be used together with the library ARCNET_S90_V41.LIB. Otherwise, the ARCNET function blocks would be exist twice.

⚠️ Note:

The function blocks of the ABB libraries do not run in the simulation mode. In the simulation mode, only the operators and the function blocks of the IEC_S90_V4x.LIB do run!
The following libraries come with version V4.1 of the 907 AC 1131 software:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEC_S90_V41.LIB</td>
<td>IEC standard library, 90 series</td>
<td>extern/obj</td>
</tr>
<tr>
<td>IECSFC_S90_V41.LIB</td>
<td>IEC sequence steps, 90 series</td>
<td>intern</td>
</tr>
<tr>
<td>Base_S90_V41.LIB</td>
<td>Basic function block library, 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>RCOM_S90_V41.LIB</td>
<td>RCOM/RCOM+ function block library, 90 series</td>
<td>extern/obj</td>
</tr>
<tr>
<td>ARCNET_S90_V41.LIB</td>
<td>ARCNET function block library, 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>CS31_S90_V41.LIB</td>
<td>Blocks for CS31 bus, 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>COM_S90_V41.LIB</td>
<td>Blocks for serial communication, 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>Datenablage_S90_V41.LIB</td>
<td>Blocks for SMC and FLASH, 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>SystemInfo_S90_V41.LIB</td>
<td>Blocks for system information, 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>Coupler_S90_V41.LIB</td>
<td>Coupler basic routines library, 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>Profibus_S90_V41.LIB</td>
<td>PROFIBUS function block library, 90 series</td>
<td>intern</td>
</tr>
</tbody>
</table>

In the following sections you will find tables containing the function blocks of the above mentioned libraries.

**Note:**
The function blocks of the Coupler_S90_V41.LIB library are not described individually because the user uses them together with the library of an internal coupler (except ARCNET), e.g. PROFIBUS DP with PROFIBUS_S90_V41.LIB.

### 1.1.3 Comparison of the libraries of versions V4.0 and as of version V4.1

Below you find a comparison of the libraries of versions V4.0 and as of V4.1 including the most important modifications.

<table>
<thead>
<tr>
<th>Version V4.1</th>
<th>Version V4.0</th>
<th>Modifications in V4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEC_S90_V41.LIB</td>
<td>Standard.LIB</td>
<td>Counter blocks CTU, CTO, CTUD</td>
</tr>
<tr>
<td>IECSFC_S90_V41.LIB</td>
<td>iecsfc.LIB</td>
<td>none</td>
</tr>
<tr>
<td>Base_S90_V41.LIB</td>
<td>ABB-BIB4.LIB</td>
<td>without groups ARCNET, CS31 bus, SMC, couplers, MODBUS, system</td>
</tr>
<tr>
<td>RCOM_S90_V41.LIB</td>
<td>RCOM_V40.LIB</td>
<td>none</td>
</tr>
<tr>
<td>ARCNET_S90_V41.LIB</td>
<td></td>
<td>Group ARCNET from ABB-BIB4.LIB and new blocks</td>
</tr>
<tr>
<td>CS31_S90_V41.LIB</td>
<td></td>
<td>Group CS31 bus from ABB-BIB4.LIB</td>
</tr>
<tr>
<td>COM_S90_V41.LIB</td>
<td></td>
<td>Groups couplers and MODBUS from ABB-BIB4.LIB and new blocks</td>
</tr>
<tr>
<td>Datenablage_S90_V41.LIB</td>
<td></td>
<td>Group SMC from ABB-BIB4.LIB and new blocks</td>
</tr>
<tr>
<td>SystemInfo_S90_V41.LIB</td>
<td></td>
<td>Group system from ABB-BIB4.LIB and new blocks</td>
</tr>
<tr>
<td>Coupler_S90_V41.LIB</td>
<td>CIF104.LIB</td>
<td>CIF104.LIB and new blocks</td>
</tr>
<tr>
<td>Profibus_S90_V41.LIB</td>
<td>PROFI40.LIB</td>
<td>none</td>
</tr>
</tbody>
</table>
1.2 The libraries of the versions V4.2 and V4.3

In version V4.2 of the 907 AC 1131, the following libraries were added:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base_S40_V41.LIB</td>
<td>Function block library for 40..50 series</td>
<td>extern</td>
</tr>
<tr>
<td>Base_S40_V41.BIB</td>
<td>Description file of the function blocks for S40..50</td>
<td>intern</td>
</tr>
<tr>
<td>ANAI4_20.LIB</td>
<td>This library contains the function block ANAI4_20. It serves for reading-in current signals with a signal range of 4...20 mA at an analog input, which is designed for an input signal range of 0...20 mA. The input current range of 4..20 mA is converted into the internal number range of 0..32760. An input current of &lt;3.6 mA is output as TRUE at the ERR output (open-circuit monitoring).</td>
<td>intern</td>
</tr>
<tr>
<td>PROFIBUS_S90_V42.LIB</td>
<td>Bug-fix: When the PROFIBUS coupler was used as slave, the bit NEW_PRM of the DPS_STAT function block was output incorrectly in the structure of the STATE_BITS output.</td>
<td>intern</td>
</tr>
</tbody>
</table>

In version V4.3 of the 907 AC 1131, the following libraries were added:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeviceNet_Master_S90_V43.LIB</td>
<td>DeviceNet library for 90 series</td>
<td>intern</td>
</tr>
<tr>
<td>Interbus_Master_S90_V43.LIB</td>
<td>Interbus library for 90 series</td>
<td>intern</td>
</tr>
<tr>
<td>PROFIBUS_Master_S90_V43.LIB</td>
<td>Function blocks for PROFIBUS DP masters 90 series</td>
<td>intern</td>
</tr>
<tr>
<td>PROFIBUS_Slave_S90_V43.LIB</td>
<td>Function blocks for PROFIBUS DP slaves 90 series</td>
<td>intern</td>
</tr>
<tr>
<td>SystemInfo_S90_V42.LIB</td>
<td>Added function blocks: HW_INFO – supplies hardware addresses of the DPR FPUEXINFO – evaluation of floating point exceptions</td>
<td>extern</td>
</tr>
<tr>
<td>Base_free_S90_V41.LIB</td>
<td>Base_S90_V41.LIB without password. Allows the user to create libraries of his own.</td>
<td>extern</td>
</tr>
<tr>
<td>COUNTW_S90_V41.LIB</td>
<td>COUNTW function block – high-speed counter</td>
<td>extern</td>
</tr>
<tr>
<td>FKG_S90_V41.LIB</td>
<td>FKGx function blocks – function generator</td>
<td>extern</td>
</tr>
<tr>
<td>LOOP_S90_V41.LIB</td>
<td>Closed loop blocks 90 series</td>
<td>extern</td>
</tr>
<tr>
<td>LowBase_S90_V41.LIB</td>
<td>selected, often used function blocks derived from Base_S90_V41.LIB</td>
<td>extern</td>
</tr>
</tbody>
</table>
1.3 The libraries of the version V5.0

The version V5.0 of the 907 AC 1131 contains all libraries of the preceding versions V4.x.

In **Version V5.0** of the 907 AC 1131, the following libraries were added:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupler_S90_V50.LIB</td>
<td>Socket functions for Ethernet added</td>
<td>extern</td>
</tr>
<tr>
<td>CANopen_Master_S90_V43.LIB</td>
<td>CANopen library 90 series</td>
<td>intern</td>
</tr>
<tr>
<td>Ethernet_S90_V50.LIB</td>
<td>Ethernet library 90 series</td>
<td>intern</td>
</tr>
<tr>
<td>RCOM_S90_V41.LIB/obj</td>
<td>(dated Oct. 14, 2002) – bug-fix in the block: READ (data was overwritten even in case of error)</td>
<td>extern/obj</td>
</tr>
</tbody>
</table>
# The IEC operators and function blocks

## The operators in the 907 AC 1131 software

**Note:**
The operators are directly integrated in the 907 AC 1131 software.

### 2.1.1 Overview of the operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>ID in AC11331</th>
<th>Block in ABB-BIBx</th>
<th>Difference ABB block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition of variables</td>
<td>ADD</td>
<td>ADDW, ADDD</td>
<td>Limitation to INT or DINT</td>
</tr>
<tr>
<td>Multiplication of variables</td>
<td>MUL</td>
<td>MULW, MULD</td>
<td>Limitation to INT or DINT</td>
</tr>
<tr>
<td>Subtraction of variables</td>
<td>SUB</td>
<td>SUBW, SUBD</td>
<td>Limitation to INT or DINT</td>
</tr>
<tr>
<td>Division of variables</td>
<td>DIV</td>
<td>DIVW, DIVD</td>
<td>Limitation to INT or DINT</td>
</tr>
<tr>
<td>Modulo division of a variable</td>
<td>MOD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Index of a block</td>
<td>INDEXOF</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>No. of bytes of a data type</td>
<td>SIZEOF</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Bitstring operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit-wise AND</td>
<td>AND</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Bit-wise OR</td>
<td>OR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Bit-wise XOR</td>
<td>XOR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Bit-wise NOT</td>
<td>NOT</td>
<td>NEGW, NEGD</td>
<td>Negation INT or DINT</td>
</tr>
<tr>
<td><strong>Bit-shift operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit-wise shift left</td>
<td>SHL</td>
<td>SHIFTW, SHIFTD</td>
<td>For INT or DINT; configuration is required</td>
</tr>
<tr>
<td>Bit-wise shift right</td>
<td>SHR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit-wise rotate left</td>
<td>ROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit-wise rotate right</td>
<td>ROR</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Selection operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary selection</td>
<td>SEL</td>
<td>-</td>
<td>In 907 PC 331 AWT</td>
</tr>
<tr>
<td>Maximum function</td>
<td>MAX</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Minimum function</td>
<td>MIN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Limitation</td>
<td>LIMIT</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Multiplexer</td>
<td>MUX</td>
<td>MUXR, MUXRD</td>
<td>Multiplexer with reset</td>
</tr>
<tr>
<td>Operator</td>
<td>ID in AC11331</td>
<td>Block in ABB-BIBx</td>
<td>Difference ABB block</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>-------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td><strong>Comparison operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greater Than</td>
<td>GT</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Less Than</td>
<td>LT</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Less than or Equal to</td>
<td>LE</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Greater than or Equal to</td>
<td>GE</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Equal</td>
<td>EQ</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Not Equal</td>
<td>NE</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td><strong>Address operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address function</td>
<td>ADR</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Contents operator</td>
<td>^</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td><strong>Call operator</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call of a function block</td>
<td>CAL</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td><strong>Numerical operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute value of a number</td>
<td>ABS</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Square root of a number</td>
<td>SQRT</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Natural logarithm</td>
<td>LN</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Logarithm of base 10</td>
<td>LOG</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Exponential function</td>
<td>EXP</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Sine of a number</td>
<td>SIN</td>
<td>SIN1</td>
<td>Result DINT with limitation</td>
</tr>
<tr>
<td>Cosine of a number</td>
<td>COS</td>
<td>COS1</td>
<td>Result DINT with limitation</td>
</tr>
<tr>
<td>Tangent of a number</td>
<td>TAN</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Arc sine of a number</td>
<td>ASIN</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Arc cosine of a number</td>
<td>ACOS</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Arc tangent of a number</td>
<td>ATAN</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Exp. Of 2 numbers</td>
<td>EXPT</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td><strong>Type conversions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOL_TO conversions</td>
<td>BOOL_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>TO_BOOL conversions</td>
<td>xxx_TO_BOOL</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Conversions between signed integers</td>
<td>xxx_TO_yyy</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>REAL_TO conversions</td>
<td>REAL_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>LREAL_TO conversions</td>
<td>LREAL_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>TIME_TO conversions</td>
<td>TIME_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>TIME_OF_DAY conversions</td>
<td>TOD_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>DATE_TO conversions</td>
<td>DATE_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>DATE_AND_TIME conversions</td>
<td>DT_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>STRING_TO conversions</td>
<td>STRING_TO_xxx</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>REAL→INT conversion, integer part</td>
<td>TRUNC</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
### 2.1.2 Overview of possible type conversions

<table>
<thead>
<tr>
<th>TO</th>
<th>BOOL</th>
<th>BYTE</th>
<th>DATE</th>
<th>DINT</th>
<th>DT</th>
<th>DWORD</th>
<th>INT</th>
<th>LREAL</th>
<th>REAL</th>
<th>SINT</th>
<th>STRING</th>
<th>TIME</th>
<th>TOD</th>
<th>UDINT</th>
<th>UINT</th>
<th>USINT</th>
<th>WORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>BYTE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DATE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DINT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DWORD</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>INT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LREAL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>REAL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SINT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>STRING</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>TIME</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>TOD</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>UDINT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>UINT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>USINT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>WORD</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
## 2.2 The IEC standard library of the 907 AC 1131, 90 series

Name: IEC_S90_V41.LIB

<table>
<thead>
<tr>
<th>Operator</th>
<th>Identifier in 907 AC 1131</th>
<th>Type</th>
<th>Further block</th>
<th>Difference ABB block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String functions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length of a string</td>
<td>LEN</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Left starting string of a string</td>
<td>LEFT</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Right starting string of a string</td>
<td>RIGHT</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Part of a string</td>
<td>MID</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Concatenation of two strings</td>
<td>CONCAT</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Insertion of a string beginning at a certain position</td>
<td>INSERT</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Deletion of a string part</td>
<td>DELETE</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Replacement of a string part</td>
<td>REPLACE</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Finding a string part</td>
<td>FIND</td>
<td>F</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Bistable function blocks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set dominant</td>
<td>SR</td>
<td>FB</td>
<td>-</td>
<td>in 907 PC 331 RS</td>
</tr>
<tr>
<td>Reset dominant</td>
<td>RS</td>
<td>FB</td>
<td>-</td>
<td>in 907 PC 331 SR</td>
</tr>
<tr>
<td>Software semaphore</td>
<td>SEMA</td>
<td>FB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Trigger</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rising edge</td>
<td>R_TRIG</td>
<td>FB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Falling edge</td>
<td>F_TRIG</td>
<td>FB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Counters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Up counter</td>
<td>CTU</td>
<td>FB</td>
<td>-</td>
<td>INT counter</td>
</tr>
<tr>
<td>Abwärtzähler</td>
<td>CTD</td>
<td>FB</td>
<td>-</td>
<td>INT counter</td>
</tr>
<tr>
<td>Up/down counter</td>
<td>CTUD</td>
<td>FB</td>
<td>VRZ, VRZD</td>
<td>INT counter Step width selectable</td>
</tr>
<tr>
<td><strong>Timers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monostable element</td>
<td>TP</td>
<td>FB</td>
<td>MOK, MOA</td>
<td>without actual-value display</td>
</tr>
<tr>
<td>ON delay</td>
<td>TON</td>
<td>FB</td>
<td>ESV</td>
<td>without actual-value display</td>
</tr>
<tr>
<td>OFF delay</td>
<td>TOF</td>
<td>FB</td>
<td>ASV</td>
<td>without actual-value display</td>
</tr>
</tbody>
</table>

## 2.3 IEC steps for sequential function chart

Name: IECSFC_S90_V41.LIB

<table>
<thead>
<tr>
<th>Operator</th>
<th>Identifier in AC11331</th>
<th>Type</th>
<th>Further block</th>
<th>Difference ABB block</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEC step control</td>
<td>SFCActionControl</td>
<td>FB</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### 3 Function block libraries of the 90 series

#### 3.1 Base function block library 90 series

#### 3.1.1 Base library with all function blocks

Name: Base_S90_V41.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V4.0</td>
</tr>
<tr>
<td><strong>Binary operations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIFSU</td>
<td>Bit searcher (set bit)</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BMELD127</td>
<td>Binary value change annunciator</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BMELD16</td>
<td>max 127 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMELD32</td>
<td>max 16 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMELD64</td>
<td>max 32 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMELD8</td>
<td>max 64 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFOB</td>
<td>First-in-first-out for bit operands</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDLB</td>
<td>Read binary variable, indexed</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDSB</td>
<td>Write binary variable, indexed</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INITB</td>
<td>Initialize operand memory area with</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INITSB</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITVB16</td>
<td>Initialize variables</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INITVB32</td>
<td>max 16 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDT</td>
<td>Illumination pushbutton control</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIFOB</td>
<td>Last-in-first-out for bit operands</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LZB</td>
<td>Run number block</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAJ</td>
<td>Majority / Bit</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Doubleword operations**

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD</td>
<td>Addition double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMELD16</td>
<td>Analog value change annunciator, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMELD32</td>
<td>max 16 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMELD63</td>
<td>max 32 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMELD8</td>
<td>max 63 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEGD</td>
<td>Limiter, double word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COUNTBD</td>
<td>Count set bits in double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DIVD</td>
<td>Division double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DMUXD16</td>
<td>Demultiplexer, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DMUXD256</td>
<td>max 16 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMUXD32</td>
<td>max 256 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMUXD64</td>
<td>max 32 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>max 64 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMUXD8</td>
<td>max 8 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWAES</td>
<td>Write double word when value changes</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DWOL</td>
<td>Read double word with enable</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DWOS</td>
<td>Write double word with enable</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DWUMC16</td>
<td>Double word recoder</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DWUMC256</td>
<td>max 16 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWUMC32</td>
<td>max 256 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWUMC64</td>
<td>max 32 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>max 64 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWUMC8</td>
<td>max 8 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITSD</td>
<td>Initialize operand memory area with TRUE (double word area)</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Group / Block</td>
<td>Description</td>
<td>Type</td>
<td>Can be run on</td>
<td>Ver.</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------</td>
<td>---------------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>MASKED</td>
<td>Mask, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAZD</td>
<td>Maximum value, function of time, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUL2ND</td>
<td>Multiplication with 2 pwr of N, double word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MULD</td>
<td>Multiplication, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXRD16</td>
<td>Multiplexer with max 16 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXRD256</td>
<td>Multiplexer with max 256 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXRD32</td>
<td>Multiplexer with max 32 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXRD64</td>
<td>Multiplexer with max 64 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXRD8</td>
<td>Multiplexer with max 8 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NEGD</td>
<td>Negation, double word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PACKD</td>
<td>Packing bits in double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SETBD</td>
<td>Set binary value in double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SHFTD</td>
<td>Shift double word left/right</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SUBD</td>
<td>Subtraction, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TESTBD</td>
<td>Test bit in double word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UNPACKD</td>
<td>Unpack double word into bits</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>USTD</td>
<td>Switch-over gate, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>USTRD</td>
<td>Switch-over gate with reset, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Conversions**

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>BCDDUAL</td>
<td>Conversion BCD-DUAL, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BCDDUALD</td>
<td>Conversion BCD-DUAL, double word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DUALBCD</td>
<td>Conversion DUAL-BCD, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DUALBCDD</td>
<td>Conversion DUAL-BCD, double word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DW2W</td>
<td>Double word → 2 words</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DWW</td>
<td>Double word to word conversion</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>W2WDW</td>
<td>2 words → double word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WDW</td>
<td>Word to double word conversion</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Loop control**

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>DT1</td>
<td>Differenciator with delay</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HLG</td>
<td>Ramp-function generator</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INTK</td>
<td>Integrator</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PI</td>
<td>Proportional-integral controller</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PIDT1</td>
<td>PIDT1 controller</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PT1</td>
<td>PT1 element (1st order)</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VGL3P</td>
<td>Comparator with 3-point response, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VGLEH</td>
<td>Comparator with unilateral hysteresis</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VGLUH</td>
<td>Comparator with asymmetrical hysteresis</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Word operations**

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>ADDW</td>
<td>Addition, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ADRWA16</td>
<td>Address selection max 16 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ADRWA8</td>
<td>Address selection max 8 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ADRWA127</td>
<td>Address selection max 127 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMELD127</td>
<td>Analog value change annunciator Max 127 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMELD16</td>
<td>Analog value change annunciator Max 16 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMELD32</td>
<td>Analog value change annunciator Max 32 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMELD8</td>
<td>Analog value change annunciator Max 8 values</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMW</td>
<td>Selection multiplexer</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Group / Block</td>
<td>Description</td>
<td>Type</td>
<td>Can be run on</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------</td>
<td>------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>BEG</td>
<td>Limiter, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COMPAR</td>
<td>Comparison of memory areas</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COPY</td>
<td>Copying of memory areas</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COS1</td>
<td>Cosine</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COUNTBW</td>
<td>Count number of set bits in word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DIVW</td>
<td>Division, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DMUX16</td>
<td>Demultiplexer, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DMUX256</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DMUX32</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DMUX64</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DMUX8</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FIFOW</td>
<td>First-in-first-out</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FKG16</td>
<td>Function generator</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FKG2</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FKG256</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FKG32</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FKG4</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FKG64</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDLm</td>
<td>Read word, indexed</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDSn</td>
<td>Write word, indexed</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INITSW</td>
<td>Initialize operand memory area with 0</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INITVW16</td>
<td>Initialize variables</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INITVW32</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIFOW</td>
<td>Last-in-first-out, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIZU16</td>
<td>List allocator</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIZU256</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIZU32</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIZU64</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIZU8</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MASKE</td>
<td>Mask, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAZ</td>
<td>Maximum value, function of time, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUL2N</td>
<td>Multiplication with 2 pwr of N, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MULDI</td>
<td>Multiplication + division, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MULW</td>
<td>Multiplication, word (with limitation)</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXR16</td>
<td>Multiplexer with reset</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXR256</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXR32</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXR64</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUXR8</td>
<td></td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NEGW</td>
<td>Negation, word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PACK</td>
<td>Packing bits into word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SETBW</td>
<td>Set bit in word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SHIFTW</td>
<td>Shift word left/right</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SIN1</td>
<td>Sine</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SUBW</td>
<td>Subtraction word with limitation</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TESTBW</td>
<td>Test a bit in word</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UNPACK</td>
<td>Unpack a word into bits</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Group / Block</td>
<td>Description</td>
<td>Type</td>
<td>Can be run on</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------</td>
<td>------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>USM</td>
<td>Switch-over MUX (to ADRWA)</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UST</td>
<td>Switch-over gate, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>USTR</td>
<td>Switch-over gate with reset, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WAES</td>
<td>Write word when value changes</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WDEC16</td>
<td>Wort decoder</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WDEC256</td>
<td>max 16 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDEC32</td>
<td>max 256 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDEC64</td>
<td>max 64 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDEC8</td>
<td>max 8 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WOL</td>
<td>Read word with enable</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WOS</td>
<td>Write word with enable</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WUMC16</td>
<td>Word recorder</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WUMC256</td>
<td>max 16 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WUMC32</td>
<td>max 256 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WUMC64</td>
<td>max 64 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WUMC8</td>
<td>max 8 values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASV</td>
<td>OFF delay</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ESV</td>
<td>ON delay</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MOA</td>
<td>Monostable element „abort”</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MOK</td>
<td>Monostable element „constant”</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PDM</td>
<td>Pulse duration modulator</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UHR</td>
<td>Display/set real-time clock</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VVZ</td>
<td>Variable delay element</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Counters</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNTW</td>
<td>High-speed counter</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VRZ</td>
<td>Up/down counter, word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VRZD</td>
<td>Up/down counter, double word</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
3.1.2 Generation of own libraries from Base_free_S90_V41.LIB

As of version 4.3 of the 907 AC 1131 software, the Base_free_S90_V41.LIB library is provided in addition to the Base_S90_V41.LIB library. This library contains all the function blocks of the Base_S90_V41.LIB library, but it is not protected by a password. This allows the user to arrange libraries of his own.

Using the Base_free_S90_V41-LIB library, the user can arrange a library of his own by the following steps:
1. Open the library with:
   - "File"=>"Open" with: File type=Library (*.lib) and File name=Base_free_S90_V41.LIB
2. Delete all function blocks and subdirectories, which are not necessary, with:
   - Select Block/Directory, right mouse button and then "Delete object"
3. Under "Resources"=>"Control system configuration", a CPU 90 series is entered
4. Under "Project"=>"Options"=>"Translation options", a 2 is entered under the number of data segments
5. Translate project with <F11>.

The warning: "This expression does not contain an allocation. No code is generated." is output for each function block.

In version V4.3, the following error, which could be ignored, was output: "Project does not contain any PLC_PRG function block".

6. Save library with:
   - "File"=>"Save as..." with File type=External library and File name: Name_given_by_you
7. Close library with:
   - "File"=>"Close"

In version V4.3, the prompt for “Save changes” must be answered with “No”. In version V5.0, this prompt does not appear.

Note:
It is not possible to insert new function blocks (or user-defined ones) in the library. The reason is that the library is an external one and the code of the function blocks is stored in the run-time system of the PLC. This is why the function blocks are not processed in the simulation mode.

3.1.3 Libraries generated from Base_S90_V41.LIB

As of version V4.3, the following subject-oriented libraries are provided, generated from the Base_S90_V41.LIB library:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNTW_S90_V41.LIB</td>
<td>COUNTW function block – high-speed counter</td>
</tr>
<tr>
<td>FKG_S90_V41.LIB</td>
<td>FKGx function blocks – function generator</td>
</tr>
<tr>
<td>LOOP_S90_V41.LIB</td>
<td>Loop control function blocks 90 series</td>
</tr>
<tr>
<td>LowBase_S90_V41.LIB</td>
<td>selected, often used function blocks from Base_S90_V41.LIB:</td>
</tr>
<tr>
<td></td>
<td>Conversions: DW2W, W2WDW</td>
</tr>
<tr>
<td></td>
<td>Double word operations: DMUXD16, DMUXD8, PACKD, UNPACKD</td>
</tr>
<tr>
<td></td>
<td>Timers: UHR, VVZ</td>
</tr>
<tr>
<td></td>
<td>Word operations: COMPARE, COPY, DMUX16, DMUX8, PACK, UNPACK</td>
</tr>
</tbody>
</table>
### 3.2 RCOM/RCOM+ function block library, 90 series

Name: **RCOM_S90_V41.LIB**

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
<td></td>
</tr>
<tr>
<td>RCOM communication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOCK</td>
<td>Set clock</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>COLDST</td>
<td>Cold start</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>DIAL</td>
<td>Dial communication partner</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>HANGUP</td>
<td>Hang-up telephone</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>NORMAL</td>
<td>Normalization</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>POLL</td>
<td>Perform event poll</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>RCOM</td>
<td>Initialize 07 KP 90 R303 for RCOM protocol</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>RCOM_PL</td>
<td>Initialize 07 KP 93 R303 for RCOM+ protocol</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from RCOM slave</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>READ_S</td>
<td>Provide data for READ task</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>RECV</td>
<td>Receive data from the RCOM partner</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>SYS_S</td>
<td>RCOM system services</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>TRANSM</td>
<td>Send data to the RCOM partner</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>WARMST</td>
<td>Warm start</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
</tbody>
</table>

### 3.3 ARCNET function block library, 90 series

Name: **ARCNET_S90_V41.LIB**

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCNET</td>
<td></td>
<td></td>
<td>07 KT 97</td>
<td>07 KT 98</td>
<td></td>
</tr>
<tr>
<td>AINIT</td>
<td>ARCNET initialization</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>AREC</td>
<td>ARCNET data reception</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>ASEND1</td>
<td>Send ARCNET data to 1 node</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>ASEND16</td>
<td>Send ARCNET data to max. 16 nodes</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>ASEND4</td>
<td>Send ARCNET data to max. 4 nodes</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>ASEND_B</td>
<td>Send ARCNET data, bytes to 1 node</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.0</td>
</tr>
<tr>
<td>ARCNET 5F communication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5F_ARC97</td>
<td>Driver for 5F-ARCNET protocols</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.05</td>
</tr>
<tr>
<td>ARCNET new</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AINIT_V</td>
<td>Initialization of ARCNET coupler</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.03</td>
</tr>
<tr>
<td>AREC_V</td>
<td>Receive ARCNET data packages</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.03</td>
</tr>
<tr>
<td>ASEND_V</td>
<td>Send ARCNET data packages to a participant</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.03</td>
</tr>
<tr>
<td>ASTO_V</td>
<td>Read ARCNET timeout data packages</td>
<td>FB</td>
<td>X</td>
<td>X</td>
<td>V4.03</td>
</tr>
</tbody>
</table>
### 3.4 Function blocks for CS31 bus, 90 series

Name: CS31_S90_V41.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>CS31CO</td>
<td>Configure AC31 modules</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CS31QU</td>
<td>Acknowledge errors on AC31 modules</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CS31TE</td>
<td>Test AC31 modules</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### 3.5 Function blocks for serial communication, 90 series

Name: COM_S90_V41.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>ASCII</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMAUTOLOGIN</td>
<td>Login 907 AC 1131 with communication cable (with connection 6-8)</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COMINIT</td>
<td>Initialization of serial interface</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COMREC</td>
<td>Data reception from a serial interface</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>COMSND</td>
<td>Data transmission via a serial interface</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ASCII</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATACMP</td>
<td>Comparison between two data areas</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAMOV</td>
<td>Composing two telegrams with different data formats</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAPOS</td>
<td>Isolate „place keeper“ areas within data areas</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MODBUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODINIT</td>
<td>MODBUS initialization od serial interface</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MODMAST</td>
<td>MODBUS master operation</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Serial coupler</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIREC</td>
<td>Network interface „Receive“</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SISEND</td>
<td>Network interface „Transmit“</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
### 3.6 Function blocks for SMC and FLASH, 90 series

**Name**: Datenablage_S90_V41.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td><strong>FCP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDEL</td>
<td>Delete segment in FLASH</td>
<td>FB</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>FRD</td>
<td>Read data set from FLASH</td>
<td>FB</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>FWR</td>
<td>Write data set into FLASH</td>
<td>FB</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td><strong>SMC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCDEL</td>
<td>Delete SmartMedia Card</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FCINIT</td>
<td>Initialize SmartMedia Card</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FCRD</td>
<td>Read data from SmartMedia Card</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FCWR</td>
<td>Write data to SmartMedia Card</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### 3.7 Function blocks for system information, 90 series

**Name**: SystemInfo_S90_V41.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td><strong>System information</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET_STACKINFO</td>
<td>Information about stack</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HEAP_INFO</td>
<td>Information about HEAP</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDENT</td>
<td>PLC identification</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Projektinfo</td>
<td>Information about the user program</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SYS_TIME</td>
<td>Read system time</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Name**: SystemInfo_S90_V42.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 95..97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td><strong>System information</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET_STACKINFO</td>
<td>Information about stack</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HEAP_INFO</td>
<td>Information about HEAP</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDENT</td>
<td>PLC identification</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Projektinfo</td>
<td>Information about the user program</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SYS_TIME</td>
<td>Read system time</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HW_INFO</td>
<td>Supplies the start addresses of the DPR</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FPUEXINFO</td>
<td>Info about floating-point exceptions</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
### 3.8 Library of coupler basic routines, 90 series

**Note:**
The function blocks of the Coupler_S90_V41.LIB library are not described individually because the user uses them together with the library of an internal coupler (except ARCNET), e.g. PROFIBUS DP with PROFIBUS_S90_V41.LIB.

**Name: Coupler_S90_V41.LIB**

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 97</td>
</tr>
<tr>
<td><strong>Coupler access</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIF_free_msg</td>
<td>Return of a memory area (buffer) to the operating system of the target hardware</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_data</td>
<td>Read data from the DP RAM of a coupler</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_info</td>
<td>Read information from a coupler</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_msg</td>
<td>Read a message from the operating system or from a coupler to the project</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_msg_buffer</td>
<td>Request of a memory area (buffer) from the operating system of the target hardware in order to generate a message there</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_put_data</td>
<td>Write data into the standard data area of the DP RAM of a coupler</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_put_msg</td>
<td>Write a message from the project to the operating system or to a coupler</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td><strong>Higher functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIF_check_all</td>
<td>Check slot number, coupler type and coupler model or operating mode</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_check_id</td>
<td>Check a slot number</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_dev_no</td>
<td>Read the module number of a coupler as a double word (DWORD)</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_fw_name_string</td>
<td>Read the firmware identifier of a coupler as STRING</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_fw_ver_string</td>
<td>Read the firmware version of a coupler as STRING</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_man_data_string</td>
<td>Read the manufacturing date of a coupler as STRING in the YYYY-MM-DD format</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_model</td>
<td>Read the model identifier of a coupler</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_op_mode</td>
<td>Read the current operating mode of a coupler</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_ser_no</td>
<td>Read the serial number of a coupler as a double word (DWORD)</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_state</td>
<td>Read the communication status of a master coupler as a byte (BYTE)</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_state_field</td>
<td>Read a status field of a master coupler as a bit field</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_get_type</td>
<td>Read the type identifier of a coupler</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>Group / Block</td>
<td>Description</td>
<td>Type</td>
<td>Can be run on</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 97</td>
</tr>
<tr>
<td>Coupler access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>see Coupler_S90_V41.LIB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higher functions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>see Coupler_S90_V41.LIB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIF_ETH_free_socket</td>
<td>Enable resources of a socket</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_ETH_get_socket</td>
<td>Request resources for a new socket</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_ETH_socket_buffer_handler</td>
<td>Create/enable protocol-specific buffers of a socket or poll information</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>about/from them</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIF_ETH_socket_handler</td>
<td>Execute operations on a socket</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>e.g. Open, Connect, Send, Receive, Close, Get/Set options</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIF_ETH_socket_info</td>
<td>Request informationen about a socket</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>CIF_ETH_socket_options_handler</td>
<td>Perform/poll protocol-specific settings of a socket</td>
<td>F</td>
<td>X</td>
</tr>
</tbody>
</table>
3.9 PROFIBUS library, 90 series

3.9.1 PROFIBUS library, 90 series in versions V4.1 and V4.2

Name: PROFIBUS_S90_V41.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 97</td>
</tr>
<tr>
<td>Information</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROFI_INFO</td>
<td>Read coupler information</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DP master blocks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPM_STAT</td>
<td>Read coupler status</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DPM_SLVDIAG</td>
<td>Read detailed PROFIBUS diagnosis data from a slave</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DPM_SYSDIAG</td>
<td>Read system diagnosis data</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DP slave blocks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPS_STAT</td>
<td>Read the coupler status</td>
<td>FB</td>
<td>X</td>
</tr>
</tbody>
</table>

3.9.2 PROFIBUS library, 90 series as of version V4.3

Name: PROFIBUS_Master_S90_V43.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 97</td>
</tr>
<tr>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROFI_INFO</td>
<td>Read coupler information</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>Status / Diagnosis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPM_STAT</td>
<td>Read the coupler status</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DPM_SLVDIAG</td>
<td>Read detailed PROFIBUS diagnosis from a slave</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DPM_SYSDIAG</td>
<td>Read system diagnosis</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPM_CTRL</td>
<td>Send control commands to DP slaves</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPM_SETPRM</td>
<td>Send user parameters to a DP slave during runtime</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>Acyclic read of I/O data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPM_READ_INPUT</td>
<td>Read input data from a slave not assigned to the master</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DPM_READ_OUTPUT</td>
<td>Read output data from a slave not assigned to the master</td>
<td>FB</td>
<td>X</td>
</tr>
</tbody>
</table>

Name: PROFIBUS_Slave_S90_V43.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 97</td>
</tr>
<tr>
<td>Status / Diagnosis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPS_STAT</td>
<td>Read coupler status</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>DPS_EXTDIAG</td>
<td>Send error flags of the basic unit as extended diagnosis</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPS_GETPRM</td>
<td>Read user parameters</td>
<td>FB</td>
<td>X</td>
</tr>
</tbody>
</table>
### 3.10 DeviceNet library, 90 series

**Name: DeviceNet_Master_S90_V43.LIB**

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td>07 SL 97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>DEVNET_INFO</td>
<td>Reading information about installed couplers</td>
<td>FB</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td><strong>DeviceNet_Master</strong></td>
<td></td>
<td></td>
<td>07 SL 97</td>
<td>07 KT 98</td>
</tr>
<tr>
<td>DNM_DEVDIAG</td>
<td>Polling diagnostic data from a slave</td>
<td>FB</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>DNM_STAT</td>
<td>Reading the DeviceNet coupler status</td>
<td>FB</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>DNM_SYSDIAG</td>
<td>Displaying status information about all slaves</td>
<td>FB</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>DNM_GET_ATTR</td>
<td>Reading an attribute from a slave object</td>
<td>FB</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>DNM_RESET_OBJ</td>
<td>Resetting a slave object</td>
<td>FB</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>DNM_SET_ATTR</td>
<td>Writing an attribute to a slave object</td>
<td>FB</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>
3.11 Interbus library, 90 series

Name: Interbus_Master_S90_V43.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 97</td>
</tr>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERBUS_INFO</td>
<td>Read information from couplers installed</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td><strong>Interbus master / diagnosis</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM_DEVDIAG</td>
<td>Poll diagnostic data from a slave</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td>IBM_STAT</td>
<td>Read status from the Interbus coupler</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td>IBM_SYSDIAG</td>
<td>Display status overview of all slaves</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td><strong>Interbus master / PCP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM_GET_OD</td>
<td>Read object descriptions from a PCP-supporting</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>slave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM_READ</td>
<td>Read parameters from a PCP-supporting slave</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td>IBM_READ_EN</td>
<td>Enable reading parameters from the control system</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>by a PCP-supporting slave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM_WRITE</td>
<td>Write parameters to a PCP-supporting slave</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td>IBM_WRITE_EN</td>
<td>Enable writing parameters to the control</td>
<td>FB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>system by a PCP-supporting slave</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 3.12 CANopen library, 90 series

Name: CANopen_Master_S90_V43.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
<th>Ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANopen_INFO</td>
<td>Reading information about the installed couplers</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>CAN 2.0 A \ Help functions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDENT_CAN2A_TO_WORD</td>
<td>Generating a general CAN telegram header according to CAN 2.0 A</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDENT_CANopen_TO_WORD</td>
<td>Generating a CANopen-specific CAN telegram header according to CAN 2.0 A</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WORD_TO_IDE NT_CAN2A</td>
<td>Splitting a general CAN telegram header according to CAN 2.0 A</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WORD_TO_IDE NT_CANopen</td>
<td>Splitting a CANopen-specific CAN telegram header according to CAN 2.0 A</td>
<td>F</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>CAN 2.0 A</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN_REC_2A</td>
<td>Receiving CAN telegrams with 11 bit identifiers according to CAN 2.0 A</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CAN_REC_FILTER_2A</td>
<td>Enabling of identifiers for receiving CAN telegrams with 11 bit identifiers according to CAN 2.0 A</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CAN_SEND_2A</td>
<td>Transmitting CAN telegrams with 11 bit identifiers according to CAN 2.0 A</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>CAN 2.0 B</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN_REC_2B</td>
<td>Receiving CAN telegrams with 29 bit identifiers according to CAN 2.0 B</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CAN_SEND_2B</td>
<td>Transmitting CAN telegrams with 29 bit identifiers according to CAN 2.0 B</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>CANopen Master \ NMT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANoM_NMT</td>
<td>Controlling NMT node states via network management</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>CANopen Master \ SDO</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANoM_SDO_READ</td>
<td>Reading an object from a slave</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CANoM_SDO_WRITE</td>
<td>Writing an object to a slave</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>CANopen Master \ Status/Diagnosis</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANoM_NODE DIAG</td>
<td>Polling diagnostic data from a slave</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CANoM_RES_ERR</td>
<td>Resetting the coupler's error indications</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CANoM_STAT</td>
<td>Reading the CANopen coupler status</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CANoM_SYSDIAG</td>
<td>Reading a status survey of all slaves</td>
<td>FB</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
3.13 Ethernet library, 90 series

Name: Ethernet_S90_V50.LIB

<table>
<thead>
<tr>
<th>Group / Block</th>
<th>Description</th>
<th>Type</th>
<th>Can be run on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>07 KT 97</td>
</tr>
<tr>
<td>General</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ETH_Info</td>
<td>Reading information about the installed couplers</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>MODBUS_TCP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETH_MODMAST</td>
<td>Processing Open MODBUS on TCP/IP client telegrams</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>ETH_MODSTAT</td>
<td>Open MODBUS processing</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>UDP_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETH_AINIT</td>
<td>Initializing the Ethernet UDP/IP data exchange</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>ETH_AREC</td>
<td>Reading an Ethernet UDP/IP data package from the receive buffer</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>ETH_ASEND</td>
<td>Transmitting an Ethernet UDP/IP data package</td>
<td>FB</td>
<td>X</td>
</tr>
<tr>
<td>ETH_ASTO</td>
<td>Reading Ethernet UDP/IP timeout data packages</td>
<td>FB</td>
<td>X</td>
</tr>
</tbody>
</table>
## 4 Conversion of 907 PC 331 blocks into 907 AC 1131, 90 series

### 4.1 Converted blocks from 907 PC 331 to 907 AC 1131, 90 series

<table>
<thead>
<tr>
<th>Description</th>
<th>CE identifier in 907 PC 331</th>
<th>Identifier in 907 AC 1131</th>
<th>ABB-BIB4 X = yes</th>
<th>Implemented in library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtraction / word</td>
<td>-</td>
<td>SUBW</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Multiplication / word</td>
<td>*</td>
<td>MULW</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Multiplication + division / word</td>
<td>*(MULDI)</td>
<td>MULDI</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Multiplication / double word</td>
<td>*(MULD)</td>
<td>MULD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Division / word</td>
<td>:</td>
<td>DIVW</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Division / double word</td>
<td>:D (DIVD)</td>
<td>DIVD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Addition / word</td>
<td>+</td>
<td>ADDW</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Addition / double word</td>
<td>+D</td>
<td>ADDD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>2 words -&gt; double word</td>
<td>2WDW</td>
<td>W2WDW</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Address selection</td>
<td>ADRWA</td>
<td>ADRWA</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>ARCNET initialization</td>
<td>AINIT</td>
<td>AINIT</td>
<td>X</td>
<td>ARCNET_S90_V41</td>
</tr>
<tr>
<td>Analog value change annunciator</td>
<td>AMELD</td>
<td>AMELD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Analog v. ch. Ann. / double word</td>
<td>AMELDD</td>
<td>AMELDD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>ARCNET data reception</td>
<td>AREC</td>
<td>AREC</td>
<td>X</td>
<td>ARCNET_S90_V41</td>
</tr>
<tr>
<td>ARCNET data transmission</td>
<td>ASEND</td>
<td>ASEND</td>
<td>X</td>
<td>ARCNET_S90_V41</td>
</tr>
<tr>
<td>OFF delay</td>
<td>ASV</td>
<td>ASV</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Selection multiplexer</td>
<td>AWM</td>
<td>AWM</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>BCD-&gt;DUAL conversion / word</td>
<td>BCDDUAL</td>
<td>BCDDUAL</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>BCD-&gt;DUAL conv. / double word</td>
<td>BCDDUALD</td>
<td>BCDDUALD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Limiter / word</td>
<td>BEG</td>
<td>BEG</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Limiter / double word</td>
<td>BEGD</td>
<td>BEGD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Bit searcher (set bit)</td>
<td>BITSU</td>
<td>BITSU</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Binary value change annunciator</td>
<td>BMELD</td>
<td>BMELD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Copying memory areas</td>
<td>COPY</td>
<td>COPY</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Cosine</td>
<td>COS1</td>
<td>COS1</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Test number of bits in word/double word</td>
<td>COUNTB</td>
<td>COUNTB</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Configure CS31 modules</td>
<td>CS31CO</td>
<td>CS31CO</td>
<td>X</td>
<td>CS31_S90_V41</td>
</tr>
<tr>
<td>Acknow. Error, CS31 modules</td>
<td>CS31QU</td>
<td>CS31QU</td>
<td>X</td>
<td>CS31_S90_V41</td>
</tr>
<tr>
<td>Subtraction / double word</td>
<td>-* (SUBD)</td>
<td>SUBD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Demultiplexer / word</td>
<td>DMUX</td>
<td>DMUX</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Demultiplexer / double word</td>
<td>DMUXD</td>
<td>DMUXD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Differentiator with delay</td>
<td>DT1</td>
<td>DT1</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Dual-&gt;BCD conversion / word</td>
<td>DUALBCD</td>
<td>DUALBCD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Dual-&gt;BCD conv. / double word</td>
<td>DUALBCDD</td>
<td>DUALBCDD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Double word -&gt; 2 words</td>
<td>DW2W</td>
<td>DW2W</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Write dword when value changes</td>
<td>DWAES</td>
<td>DWAES</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Read double word with enable</td>
<td>DWOL</td>
<td>DWOL</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Write double word with enable</td>
<td>DWOS</td>
<td>DWOS</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Double word recorder</td>
<td>DWUMC</td>
<td>DWUMC</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Double word-&gt;word conversion</td>
<td>DWW</td>
<td>DWW</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Description</td>
<td>CE identifier in 907 PC 331</td>
<td>Identifier in 907 AC 1131</td>
<td>ABB-BIB4 X = yes</td>
<td>Implemented in library</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------</td>
<td>---------------------------</td>
<td>----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>ON delay</td>
<td>ESV</td>
<td>ESV</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Delete FLASH segment</td>
<td>FDEL</td>
<td>FDEL</td>
<td></td>
<td>Datenablage_S90_V41</td>
</tr>
<tr>
<td>Write data set in FLASH</td>
<td>FWR</td>
<td>FWR</td>
<td></td>
<td>Datenablage_S90_V41</td>
</tr>
<tr>
<td>Read data set from FLASH</td>
<td>FRD</td>
<td>FRD</td>
<td></td>
<td>Datenablage_S90_V41</td>
</tr>
<tr>
<td>Delete Flash card</td>
<td>FCDEL</td>
<td>FCDEL</td>
<td>X</td>
<td>Datenablage_S90_V41</td>
</tr>
<tr>
<td>Read Flash card</td>
<td>FCRD</td>
<td>FCRD</td>
<td></td>
<td>Datenablage_S90_V41</td>
</tr>
<tr>
<td>Write Flash card</td>
<td>FCWR</td>
<td>FCWR</td>
<td>X</td>
<td>Datenablage_S90_V41</td>
</tr>
<tr>
<td>First-in-first-out</td>
<td>FIFO</td>
<td>FIFO</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Function generator</td>
<td>FKG</td>
<td>FKG</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Ramp-function generator</td>
<td>HLG</td>
<td>HLG</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Read indexed / bit</td>
<td>IDLB</td>
<td>IDLB</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Read indexed / word</td>
<td>IDLm (IDL)</td>
<td>IDLm(IDL)</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Write indexed / bit</td>
<td>IDSB</td>
<td>IDSB</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Write indexed / word</td>
<td>IDSm (IDS)</td>
<td>IDSm(IDS)</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Initialize operand memory area with 0</td>
<td>INITS</td>
<td>INITS</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Initialize variables</td>
<td>INITV</td>
<td>INITV</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Integrator (extended)</td>
<td>INTK</td>
<td>INTK</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Illumination pushbutton control</td>
<td>LDT</td>
<td>LDT</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Last-in-first-out</td>
<td>LIFO</td>
<td>LIFO</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>List allocator</td>
<td>LIZU</td>
<td>LIZU</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Run number block</td>
<td>LZB</td>
<td>LZB</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Majority / bit</td>
<td>MAJ</td>
<td>MAJ</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Mask / word</td>
<td>MASKE</td>
<td>MASKE</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Mask / double word</td>
<td>MASKED</td>
<td>MASKED</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Maximum value - time / word</td>
<td>MAZ</td>
<td>MAZ</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Maximum value - time / DWord</td>
<td>MAZD</td>
<td>MAZD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Monostable element „constant“</td>
<td>MOA</td>
<td>MOA</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Initialize MODBUS</td>
<td>MODINI</td>
<td>MODINIT</td>
<td>X</td>
<td>COM_S90_V41</td>
</tr>
<tr>
<td>MODBUS master</td>
<td>MODBUS</td>
<td>MODMAST</td>
<td>X</td>
<td>COM_S90_V41</td>
</tr>
<tr>
<td>Monostable element „constant“</td>
<td>MOK</td>
<td>MOK</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Multiplic. with 2 pwr of N / word</td>
<td>MUL2N</td>
<td>MUL2N</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Multiplic. with 2 pwr of N / DWord</td>
<td>MUL2ND</td>
<td>MUL2ND</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Multiplexer with reset / word</td>
<td>MUXR</td>
<td>MUXR</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Multiplexer with reset / DWord</td>
<td>MUXRD</td>
<td>MUXRD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Negation / double word</td>
<td>NEGD</td>
<td>NEGD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Pack bits-&gt;word</td>
<td>PACK</td>
<td>PACK</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Pack bits-&gt;double word</td>
<td>PACKD</td>
<td>PACKD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Pulse duration modulator</td>
<td>PDM</td>
<td>PDM</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Proportional-integral controller</td>
<td>PI</td>
<td>PI</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>PIDT1 controller</td>
<td>PIDT1</td>
<td>PIDT1</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>PT1 element</td>
<td>PT1</td>
<td>PT1</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Set binary value in wort/double word</td>
<td>SETB</td>
<td>SETB</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Shift / bits at the inputs</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Description</td>
<td>CE identifier in 907 PC 331</td>
<td>Identifier in 907 AC 1131</td>
<td>ABB-BIB4 X = yes</td>
<td>Implemented in library</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-----------------------------</td>
<td>---------------------------</td>
<td>-----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Sine</td>
<td>SIN1</td>
<td>SIN1</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Initialization of interface</td>
<td>SINIT</td>
<td>SINIT</td>
<td>-</td>
<td>COM_S90_V41</td>
</tr>
<tr>
<td>Network interface, „Receive“</td>
<td>SIREC</td>
<td>SIREC</td>
<td>X</td>
<td>COM_S90_V41</td>
</tr>
<tr>
<td>Network interface, „Send“</td>
<td>SISEND</td>
<td>SISEND</td>
<td>X</td>
<td>COM_S90_V41</td>
</tr>
<tr>
<td>Test bit in word/double word</td>
<td>TESTB</td>
<td>TESTB</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Display/set clock</td>
<td>UHR</td>
<td>UHR</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Unpack word-&gt;bits</td>
<td>UNPACK</td>
<td>UNPACK</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Unpack double word-&gt;bits</td>
<td>UNPACKD</td>
<td>UNPACKD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Switch-over MUX (to ADRWA)</td>
<td>USM</td>
<td>USM</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Switch-over gate / word</td>
<td>UST</td>
<td>UST</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Switch-over gate / double word</td>
<td>USTD</td>
<td>USTD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Switch-over gate with reset / word</td>
<td>USTR</td>
<td>USTR</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Switch-over gate with reset / DWord</td>
<td>USTRD</td>
<td>USTRD</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Comparator 3-point / word</td>
<td>VGL3P</td>
<td>VGL3P</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Comp. unilateral hysteresis</td>
<td>VGLEH</td>
<td>VGLEH</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Comp. asymm. hysteresis</td>
<td>VGLUH</td>
<td>VGLUH</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Up/down counter / word</td>
<td>VRZ</td>
<td>VRZ</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Up/down counter / DWord</td>
<td>VZ</td>
<td>VZ</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Variable delay element</td>
<td>VZ</td>
<td>VZ</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Write word when value changes.</td>
<td>WAES</td>
<td>WAES</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Word decoder</td>
<td>WDEC</td>
<td>WDEC</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Conversion word-&gt;double word</td>
<td>WDW</td>
<td>WDW</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Read word with enable</td>
<td>WOL</td>
<td>WOL</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Write word with enable</td>
<td>WOS</td>
<td>WOS</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
<tr>
<td>Word recoder</td>
<td>WUMC</td>
<td>WUMC</td>
<td>X</td>
<td>BASE_S90_V41</td>
</tr>
</tbody>
</table>
### 4.2 Not converted 907 PC 331 blocks

<table>
<thead>
<tr>
<th>Description</th>
<th>VE identifier in 907 PC 331</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional program end</td>
<td>=PE</td>
<td>Not necessary</td>
</tr>
<tr>
<td>Program abortion</td>
<td>ABORT</td>
<td>-</td>
</tr>
<tr>
<td>Adaption for controller</td>
<td>ADAPT</td>
<td>-</td>
</tr>
<tr>
<td>Convert 0..20 into 4..20</td>
<td>ANAI4_20</td>
<td>- as of V 4.3: contained in library ANAI4_20.LIB</td>
</tr>
<tr>
<td>Data packages to ARCNET</td>
<td>APOLL</td>
<td>Not necessary</td>
</tr>
<tr>
<td>ARCNET send extension</td>
<td>ASEND+</td>
<td>Not necessary</td>
</tr>
<tr>
<td>Call C subroutine</td>
<td>CALLC</td>
<td>-</td>
</tr>
<tr>
<td>Call Assembler subroutine</td>
<td>CALLUP</td>
<td>-</td>
</tr>
<tr>
<td>Read direct inputs</td>
<td>DIN</td>
<td>-</td>
</tr>
<tr>
<td>Write direct outputs</td>
<td>DOUT</td>
<td>-</td>
</tr>
<tr>
<td>Output of ASCII characters</td>
<td>DRUCK</td>
<td>Replacement block in COM_S90_V41.LIB</td>
</tr>
<tr>
<td>Receive ASCII characters</td>
<td>EMASm</td>
<td>Replacement block in COM_S90_V41.LIB</td>
</tr>
<tr>
<td>Addition for EMASm</td>
<td>EMASmVT</td>
<td>Replacement block in COM_S90_V41.LIB</td>
</tr>
<tr>
<td>Error searcher with delete</td>
<td>FEHSU</td>
<td>-</td>
</tr>
<tr>
<td>Configure process image</td>
<td>IOCON</td>
<td>-</td>
</tr>
<tr>
<td>Read byte from I/O address</td>
<td>IOR</td>
<td>-</td>
</tr>
<tr>
<td>Write byte to I/O address</td>
<td>IOW</td>
<td>-</td>
</tr>
<tr>
<td>Program end</td>
<td>PE</td>
<td>-</td>
</tr>
<tr>
<td>Read historical value / bit</td>
<td>RDB</td>
<td>Not necessary</td>
</tr>
<tr>
<td>Read historical value/ DWord</td>
<td>RDDW</td>
<td>-</td>
</tr>
<tr>
<td>Read historical value/ word</td>
<td>RDW</td>
<td>-</td>
</tr>
<tr>
<td>Error searcher with storage</td>
<td>SFEHSU</td>
<td>-</td>
</tr>
<tr>
<td>Bit-&gt;historical value</td>
<td>WRB</td>
<td>-</td>
</tr>
<tr>
<td>Double word&gt;historical value</td>
<td>WRDW</td>
<td>-</td>
</tr>
<tr>
<td>Word-&gt;historical value</td>
<td>WRW</td>
<td>-</td>
</tr>
</tbody>
</table>
5 Index

A

ARCNET function block library 90 series 17
ARCNET_S90_V41.LIB 17

B

Base function block library 90 series 11
Base function block library 90 series 11
Base_S90_V41.LIB 11

C

CANopen function block library, 90 series 25
CANopen_Master_S90_V43.LIB 25
COM_S90_V41.LIB 18
Comparison of the libraries of versions V4.0 and as of V4.1 4
Conversion of blocks from 907 PC 331, 90 series 27
Coupler_S90_V41.LIB 20
Coupler_S90_V50.LIB 21
CS31 bus function block library 90 series 18
CS31_S90_V41.LIB 18

D

Datenablage_S90_V41.LIB 19
DeviceNet library, 90 series 23
DeviceNet_Master_S90_V43.LIB 23

E

Ethernet function block library, 90 series 26
Ethernet_S90_V50.LIB 26

F

Function block libraries 90 series 11
Function blocks for serial communication, 90 series 18
Function blocks for SmartMedia Card and FLASH, series 90 19
Function blocks for system information, 90 series 19

G

Generation of own libraries from
  Base_free_S90_V41.LIB 90 series 16

I

IEC standard library 10
IEC steps for sequential function chart 10
IEC_S90_V41.LIB 10

IECSFC_S90_V41.LIB 10
Interbus function block library, 90 series 24
Interbus_Master_S90_V43.LIB 24

L

Libraries version V4.0 3
Libraries version V4.1 3
Library of coupler basic routines, 90 series 20

N

Not converted blocks from 907 PC 331, 90 series 30

O

Operators in 907 AC 1131 7

P

PROFIBUS library, 90 series 22
PROFIBUS_Master_S90_V43.LIB 22
PROFIBUS_S90_V41.LIB 22
PROFIBUS_Slave_S90_V43.LIB 22

R

RCOM function block library 90 series 17
RCOM_S90_V41.LIB 17

S

SystemInfo_S90_V41.LIB 19
SystemInfo_S90_V42.LIB 19
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

Base Function Block Library
90 Series
Contents

The Base_S90 function block library 4
Special characteristics of the Base_S90 function block library .................................................. 4
Functions: .......................................................................................................................... 4
Function blocks: .................................................................................................................. 4
Special characteristics of individual blocks: .............................................................................. 4

Overview of blocks arranged according to their call names 6

ADDITION DOUBLE WORD
ADDITION WORD
ADDRESS SELECTION
ANALOG VALUE CHANGE ANNUNCIATOR, WORD
ANALOG VALUE CHANGE ANNUNCIATOR, DOUBLE WORD
OFF DELAY
SELECTION MULTIPLEXER
BCD TO DUAL CONVERSION, WORD
BCD TO DUAL CONVERSION, DOUBLE WORD
LIMITER, WORD
LIMITER, DOUBLE WORD
BIT SEARCHER
BINARY VALUE CHANGE ANNUNCIATOR
COMPARISON OF TWO DATA BLOCKS
COPYING MEMORY AREAS
COSINE
BIT COUNTER, DOUBLE WORD
BIT COUNTER, WORD
HIGH-SPEED COUNTER
DIVISION DOUBLE WORD
DIVISION WORD
DEMULTIPLEXER, WORD
DEMULTIPLEXER, DOUBLE WORD
DIFFERENTIATOR WITH DELAY OF THE 1ST ORDER
DUAL TO BCD CONVERSION, WORD
DUAL TO BCD CONVERSION, DOUBLE WORD
DOUBLE WORD TO TWO WORDS CONVERSION
WRITE DOUBLE WORD IN THE EVENT OF VALUE CHANGE
READ DOUBLE WORD WITH ENABLING
WRITE DOUBLE WORD WITH ENABLING
DOUBLE WORD RECORDER
DOUBLE WORD TO WORD CONVERSION
ON DELAY
STACK MEMORY FIRST IN FIRST OUT, BINARY
STACK MEMORY FIRST IN FIRST OUT, WORD
FUNCTION GENERATOR
RAMP FUNCTION GENERATOR
READ BINARY VARIABLE, INDEXED
READ WORD VARIABLE, INDEXED
WRITE BINARY VARIABLE, INDEXED
WRITE WORD VARIABLE, INDEXED
INITIALIZE MEMORY AREA IN THE BINARY
OPERAND MEMORY WITH ZERO
INITIALIZE MEMORY AREA IN THE DOUBLE WORD
OPERAND MEMORY WITH ZERO
INITIALIZE MEMORY AREA IN THE WORD OPERAND MEMORY WITH ZERO
INITIALIZE VARIABLES, BINARY
INITIALIZE VARIABLES, WORD
INTEGRATOR (EXTENDED)
ILLUMINATION PUSHBUTTON CONTROL
STACK MEMORY LAST IN FIRST OUT, BINARY
STACK MEMORY LAST IN FIRST OUT, WORD
LIST ALLOCATOR
RUN NUMBER BLOCK
MAJORITY
MASK, WORD
MASK, DOUBLE WORD
MAXIMUM VALUE GENERATOR AS A FUNCTION OF TIME, WORD
MAXIMUM VALUE GENERATOR AS A FUNCTION OF TIME, DOUBLE WORD
MONOSTABLE ELEMENT »ABORT«
MONOSTABLE ELEMENT »CONSTANT«
MULTIPLICATION BY 2 TO THE POWER OF N, WORD
MULTIPLICATION BY 2 TO THE POWER OF N, DOUBLE WORD
MULTIPLICATION, DOUBLE WORD
MULTIPLICATION WITH DIVISION
MULTIPLICATION, WORD
MULTIPLIER WITH RESET, WORD
MULTIPLIER WITH RESET, DOUBLE WORD
NEGATION, DOUBLE WORD
NEGATION, WORD
PACK BINARY VARIABLES IN WORD
PACK BINARY VARIABLES IN DOUBLE WORD
PULSE DURATION MODULATOR
PROPORTIONAL INTEGRAL CONTROLLER
PIDT1 CONTROLLER
PT1 ELEMENT
SET BIT IN DOUBLE WORD
SET BIT IN WORD
SHIFT BLOCK, DOUBLE WORD
SHIFT BLOCK, WORD
SINE 0.0 TO 360.0 DEGREES
SUBTRACTION, DOUBLE WORD
SUBTRACTION, WORD
TEST BIT IN DOUBLE WORD
TEST BIT IN WORD
DISPLAY AND SET CLOCK
UNPACKING A WORD INTO BINARY VARIABLES
UNPACKING A DOUBLE WORD INTO BINARY VARIABLES
SWITCHOVER MULTIPLIER
SWITCHOVER GATE
SWITCHOVER GATE / DOUBLE WORD
SWITCHOVER GATE WITH RESET
SWITCHOVER GATE WITH RESET / DOUBLE WORD
COMPARATOR WITH 3-POINT RESPONSE
COMPARATOR WITH UNILATERAL HYSTERESIS
COMPARATOR WITH ASYMMETRICAL HYSTERESIS
UP-DOWN COUNTER
UP-DOWN COUNTER / DOUBLE WORD
VARIABLE DELAY ELEMENT
TWO WORDS INTO DOUBLE WORD CONVERSION W2WDW .......... 176
WRITE WORD IN THE EVENT OF VALUE CHANGE WAES ............... 177
WORD DECODER WDEC(8...256) .... 178
WORD TO DOUBLE WORD CONVERSION WDW ................. 180
READ WORD WITH ENABLING WOL .................. 181
WRITE WORD WITH ENABLING WOS ................. 182
WORD RECORDER WUMC(8...256).... 183

Glossary 185

Index 187
The Base_S90 function block library

Special characteristics of the Base_S90 function block library

The manufacturer's basic functions and function blocks are integrated in the Base_S90 library. These blocks were, identically in their functionality, converted from the well-known blocks of the 07 KT 94 EBS. However, there are some special things to keep in mind when the blocks are selected and called.

**Caution:**
The blocks of the Base Library only run in the RUN mode of the PLC, not in the simulation mode.

Functions:
The FBD blocks are identically constructed compared to the blocks available in the previous programming software. Some input and output names have been changed as not all characters are allowed. In all programming languages the order of the inputs and outputs is identical. The library window displays the assignment of the blocks (order and variable type).

Function blocks:
A main characteristic of the function blocks is that an instance has to be defined when calling them. In this case the following differences has to be observed.

1. **Function blocks with historical values:**
For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets have to be called.

**Example:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRZ1</td>
<td>VRZ</td>
</tr>
<tr>
<td>VRZ2</td>
<td>VRZ</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

2. **Function blocks without historical values:**
For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

**Special characteristics of individual blocks:**

1) Negation of inputs and outputs:
In the 907 AC 1131 the negation of inputs and outputs is only possible for bit operands (variable types: BOOL, BYTE, WORD, DWORD). The known operands word flag %MW (MW) and double word flag %MD (MD) are of the variable type INT and DINT. Negating of these variables is only possible with the blocks NEGW and NEGD.

2) Duplication of inputs:
Blocks: ADDW, DIVW, MULW, SUBW
The duplication of the inputs is performed by linking together several functions.

In addition the operators (ADD, DIV, MUL, SUB) can be used and duplicated as often as required. But these IEC blocks do not have a limitation.

3) Duplication of inputs and outputs:
Blocks: e.g. PACK, PACKD, UNPACK, UNPACKD
In the FBD editor the number of duplications is inquired when entering the data. Input n is assigned accordingly and the block is displayed.
4) Different data types:

    Blocks: SHIFTW, SHIFTD

    The SHIFT block available in the 907 PC 331 can process word as well as double word variables. This functionality was divided into two blocks:

    SHIFTW ... Shift word variable
    SHIFTD ... Shift double word variable

    Due to the division the switchover input WORD/DIGITAL was removed.

5) Special solutions:

    Blocks: FKG2, FKG4, FKG16, FKG32, FKG256

    According to the number of points the FKG block is selected. The number assigned to FKG represents the maximum number of available points. At input n the real number of points is indicated. Not used points do not have to be assigned.

    Example:
    – 7 points are required
      i.e. FKG16 must be selected
      n = 7
      the inputs XC0/YC0...XC6/YC6 are assigned.

    Blocks: WUMC8, WUMC16, WUMC32, WUMC64, WUMC256

    According to the number of comparison parameters the WUMC block is selected. The number assigned to WUMC represents the maximum number of available comparison parameters. At input n the real number of comparison parameters is indicated. Not used inputs do not have to be assigned.

    Example:
    – 10 comparison parameters are required
      i.e. WUMC16 must be selected
      n = 10
      the inputs EC0/AC0...EC9/AC9 are assigned

    Blocks: WDEC8, WDEC16, WDEC32, WDEC64, WDEC256

    According to the number of comparison parameters the WDEC block is selected. The number assigned to WDEC represents the maximum number of available comparison parameters. At input n the real number of comparison parameters is indicated. Not used inputs do not have to be assigned.

    Example:
    – 10 comparison parameters are required
      i.e. WDEC16 must be selected
      n = 10
      the inputs EC0...EC9 are assigned
Overview of blocks arranged according to their call names

Character description:

- **FBmV** ... Function block with historical values
- **FBoV** ... Function block without historical values
- **F** ... Function

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD</td>
<td>FBoV</td>
<td>Addition, double word</td>
<td>11</td>
</tr>
<tr>
<td>ADDW</td>
<td>F</td>
<td>Addition, word</td>
<td>12</td>
</tr>
<tr>
<td>ARWA8</td>
<td>FBoV</td>
<td>Address selection with a maximum of 8 comparison values</td>
<td>13</td>
</tr>
<tr>
<td>ARWA16</td>
<td>FBoV</td>
<td>Address selection with a maximum of 16 comparison values</td>
<td>13</td>
</tr>
<tr>
<td>AMELD8</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 8 comparison values, word</td>
<td>16</td>
</tr>
<tr>
<td>AMELD16</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 16 comparison values, word</td>
<td>16</td>
</tr>
<tr>
<td>AMELD32</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 32 comparison values, word</td>
<td>16</td>
</tr>
<tr>
<td>AMELD127</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 127 comparison values, word</td>
<td>16</td>
</tr>
<tr>
<td>AMELDD8</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 8 comparison values, Dword</td>
<td>18</td>
</tr>
<tr>
<td>AMELDD16</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 16 comparison values, Dword</td>
<td>18</td>
</tr>
<tr>
<td>AMELDD32</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 32 comparison values, Dword</td>
<td>18</td>
</tr>
<tr>
<td>AMELDD63</td>
<td>FBmV</td>
<td>Analog value change annunciator with max. 63 comparison values, Dword</td>
<td>18</td>
</tr>
<tr>
<td>ASV</td>
<td>FBmV</td>
<td>OFF delay</td>
<td>20</td>
</tr>
<tr>
<td>AWM</td>
<td>F</td>
<td>Selection multiplexer</td>
<td>21</td>
</tr>
<tr>
<td>BCDDUAL</td>
<td>F</td>
<td>BCD → DUAL conversion / word</td>
<td>22</td>
</tr>
<tr>
<td>BCDDUALD</td>
<td>F</td>
<td>BCD → DUAL conversion / double word</td>
<td>24</td>
</tr>
<tr>
<td>BEG</td>
<td>F</td>
<td>Limiter, word</td>
<td>26</td>
</tr>
<tr>
<td>BEGD</td>
<td>F</td>
<td>Limiter, double word</td>
<td>27</td>
</tr>
<tr>
<td>BITSU</td>
<td>FBmV</td>
<td>Bit searcher (set bit)</td>
<td>28</td>
</tr>
<tr>
<td>BMELD8</td>
<td>FBmV</td>
<td>Binary value change annunciator with max. 8 comparison values, binary</td>
<td>31</td>
</tr>
<tr>
<td>BMELD16</td>
<td>FBmV</td>
<td>Binary value change annunciator with max. 16 comparison values, binary</td>
<td>31</td>
</tr>
<tr>
<td>BMELD32</td>
<td>FBmV</td>
<td>Binary value change annunciator with max. 32 comparison values, binary</td>
<td>31</td>
</tr>
<tr>
<td>BMELD64</td>
<td>FBmV</td>
<td>Binary value change annunciator with max. 64 comparison values, binary</td>
<td>31</td>
</tr>
<tr>
<td>BMELD127</td>
<td>FBmV</td>
<td>Binary value change annunciator with max. 127 comparison values, binary</td>
<td>31</td>
</tr>
<tr>
<td>COMPARE</td>
<td>F</td>
<td>Comparison of two data sets</td>
<td>33</td>
</tr>
<tr>
<td>COPY</td>
<td>F</td>
<td>Copying memory areas</td>
<td>34</td>
</tr>
<tr>
<td>COS1</td>
<td>FBoV</td>
<td>Cosine</td>
<td>35</td>
</tr>
<tr>
<td>COUNTBD</td>
<td>FBoV</td>
<td>Bit counter, double word</td>
<td>37</td>
</tr>
<tr>
<td>COUNTBW</td>
<td>FBoV</td>
<td>Bit counter, word</td>
<td>38</td>
</tr>
<tr>
<td>COUNTW</td>
<td>FBmV</td>
<td>High-speed counter</td>
<td>39</td>
</tr>
<tr>
<td>DIVD</td>
<td>FBoV</td>
<td>Division, double word</td>
<td>43</td>
</tr>
<tr>
<td>CE Name</td>
<td>Type</td>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>---------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>DIVW</td>
<td>F</td>
<td>Division, word</td>
<td>45</td>
</tr>
<tr>
<td>DMUX8</td>
<td>FBoV</td>
<td>Word demultiplexer with max. 8 word inputs</td>
<td>46</td>
</tr>
<tr>
<td>DMUX16</td>
<td>FBoV</td>
<td>Word demultiplexer with max. 16 word inputs</td>
<td>46</td>
</tr>
<tr>
<td>DMUX32</td>
<td>FBoV</td>
<td>Word demultiplexer with max. 32 word inputs</td>
<td>46</td>
</tr>
<tr>
<td>DMUX64</td>
<td>FBoV</td>
<td>Word demultiplexer with max. 64 word inputs</td>
<td>46</td>
</tr>
<tr>
<td>DMUX256</td>
<td>FBoV</td>
<td>Word demultiplexer with max. 256 word inputs</td>
<td>46</td>
</tr>
<tr>
<td>DMUXD8</td>
<td>FBoV</td>
<td>Double word demultiplexer with max. 8 word inputs</td>
<td>48</td>
</tr>
<tr>
<td>DMUXD16</td>
<td>FBoV</td>
<td>Double word demultiplexer with max. 16 word inputs</td>
<td>48</td>
</tr>
<tr>
<td>DMUXD32</td>
<td>FBoV</td>
<td>Double word demultiplexer with max. 32 word inputs</td>
<td>48</td>
</tr>
<tr>
<td>DMUXD64</td>
<td>FBoV</td>
<td>Double word demultiplexer with max. 64 word inputs</td>
<td>48</td>
</tr>
<tr>
<td>DMUXD256</td>
<td>FBoV</td>
<td>Double word demultiplexer with max. 256 word inputs</td>
<td>48</td>
</tr>
<tr>
<td>DT1</td>
<td>FBmV</td>
<td>Differentiator with delay of the 1st order</td>
<td>50</td>
</tr>
<tr>
<td>DUALBCD</td>
<td>F</td>
<td>DUAL → BCD conversion / word</td>
<td>52</td>
</tr>
<tr>
<td>DUALBCDD</td>
<td>F</td>
<td>DUAL → BCD conversion / double word</td>
<td>54</td>
</tr>
<tr>
<td>DW2W</td>
<td>FBoV</td>
<td>Double word to 2-word conversion</td>
<td>56</td>
</tr>
<tr>
<td>DWAES</td>
<td>FBmV</td>
<td>Write double word in the event of value change</td>
<td>57</td>
</tr>
<tr>
<td>DWO</td>
<td>F</td>
<td>Read double word, with enable</td>
<td>58</td>
</tr>
<tr>
<td>DWOS</td>
<td>F</td>
<td>Write double word, with enable</td>
<td>59</td>
</tr>
<tr>
<td>DWUMC8</td>
<td>FBoV</td>
<td>Double word recoder with max. 8 comparison values</td>
<td>60</td>
</tr>
<tr>
<td>DWUMC16</td>
<td>FBoV</td>
<td>Double word recoder with max. 16 comparison values</td>
<td>60</td>
</tr>
<tr>
<td>DWUMC32</td>
<td>FBoV</td>
<td>Double word recoder with max. 32 comparison values</td>
<td>60</td>
</tr>
<tr>
<td>DWUMC64</td>
<td>FBoV</td>
<td>Double word recoder with max. 64 comparison values</td>
<td>60</td>
</tr>
<tr>
<td>DWUMC256</td>
<td>FBoV</td>
<td>Double word recoder with max. 256 comparison values</td>
<td>60</td>
</tr>
<tr>
<td>DWW</td>
<td>FBoV</td>
<td>Double word to word conversion</td>
<td>62</td>
</tr>
<tr>
<td>ESV</td>
<td>FBmV</td>
<td>ON delay</td>
<td>63</td>
</tr>
<tr>
<td>FIFOB</td>
<td>FBmV</td>
<td>Stack memory First in First out, binary</td>
<td>64</td>
</tr>
<tr>
<td>FIFOW</td>
<td>FBmV</td>
<td>Stack memory First in First out, word</td>
<td>66</td>
</tr>
<tr>
<td>FKG2</td>
<td>FBoV</td>
<td>Function generator with max. 2 points</td>
<td>68</td>
</tr>
<tr>
<td>FKG4</td>
<td>FBoV</td>
<td>Function generator with max. 4 points</td>
<td>68</td>
</tr>
<tr>
<td>FKG16</td>
<td>FBoV</td>
<td>Function generator with max. 16 points</td>
<td>68</td>
</tr>
<tr>
<td>FKG32</td>
<td>FBoV</td>
<td>Function generator with max. 32 points</td>
<td>68</td>
</tr>
<tr>
<td>FKG64</td>
<td>FBoV</td>
<td>Function generator with max. 64 points</td>
<td>68</td>
</tr>
<tr>
<td>FKG256</td>
<td>FBoV</td>
<td>Function generator with max. 256 points</td>
<td>68</td>
</tr>
<tr>
<td>HLG</td>
<td>FBmV</td>
<td>Ramp function generator</td>
<td>70</td>
</tr>
<tr>
<td>IDLB</td>
<td>FBoV</td>
<td>Read binary variable, indexed</td>
<td>74</td>
</tr>
<tr>
<td>IDLm</td>
<td>FBoV</td>
<td>Read word variable, indexed</td>
<td>76</td>
</tr>
<tr>
<td>IDS B</td>
<td>FBoV</td>
<td>Write binary variable, indexed</td>
<td>78</td>
</tr>
</tbody>
</table>
## OVERVIEW OF BLOCKS ARRANGED ACCORDING TO THEIR CALL NAMES

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDSm</td>
<td>FBoV</td>
<td>Write word variable, indexed</td>
<td>80</td>
</tr>
<tr>
<td>INITSB</td>
<td>FBmV</td>
<td>Initialize memory area in the binary operand memory with zero</td>
<td>82</td>
</tr>
<tr>
<td>INITSD</td>
<td>FBmV</td>
<td>Initialize memory area in the double word operand memory with zero</td>
<td>83</td>
</tr>
<tr>
<td>INITSW</td>
<td>FBmV</td>
<td>Initialize memory area in the word operand memory with zero</td>
<td>84</td>
</tr>
<tr>
<td>INITVB16</td>
<td>FBmV</td>
<td>Initialize binary variables with max. 16 duplications</td>
<td>85</td>
</tr>
<tr>
<td>INITVB32</td>
<td>FBmV</td>
<td>Initialize binary variables with max. 32 duplications</td>
<td>85</td>
</tr>
<tr>
<td>INITVW16</td>
<td>FBmV</td>
<td>Initialize word variables with max. 16 duplications</td>
<td>87</td>
</tr>
<tr>
<td>INITVW32</td>
<td>FBmV</td>
<td>Initialize word variables with max. 32 duplications</td>
<td>87</td>
</tr>
<tr>
<td>INTK</td>
<td>FBmV</td>
<td>Integrator (extended)</td>
<td>89</td>
</tr>
<tr>
<td>LDT</td>
<td>FBmV</td>
<td>Illumination pushbutton control</td>
<td>93</td>
</tr>
<tr>
<td>LIFOB</td>
<td>FBmV</td>
<td>Stack memory Last in First out, binary</td>
<td>95</td>
</tr>
<tr>
<td>LIFOW</td>
<td>FBmV</td>
<td>Stack memory Last in First out, word</td>
<td>97</td>
</tr>
<tr>
<td>LIZU8</td>
<td>FBoV</td>
<td>List allocator with max. 8 word inputs</td>
<td>99</td>
</tr>
<tr>
<td>LIZU16</td>
<td>FBoV</td>
<td>List allocator with max. 16 word inputs</td>
<td>99</td>
</tr>
<tr>
<td>LIZU32</td>
<td>FBoV</td>
<td>List allocator with max. 32 word inputs</td>
<td>99</td>
</tr>
<tr>
<td>LIZU64</td>
<td>FBoV</td>
<td>List allocator with max. 64 word inputs</td>
<td>99</td>
</tr>
<tr>
<td>LIZU256</td>
<td>FBoV</td>
<td>List allocator with max. 256 word inputs</td>
<td>99</td>
</tr>
<tr>
<td>LZB</td>
<td>FBmV</td>
<td>Run number block</td>
<td>101</td>
</tr>
<tr>
<td>MAJ</td>
<td>F</td>
<td>Majority</td>
<td>102</td>
</tr>
<tr>
<td>MASKE</td>
<td>FBoV</td>
<td>Mask, word</td>
<td>103</td>
</tr>
<tr>
<td>MASKED</td>
<td>FBoV</td>
<td>Mask, double word</td>
<td>104</td>
</tr>
<tr>
<td>MAZ</td>
<td>FBmV</td>
<td>Maximum value generator as a function of time, word</td>
<td>105</td>
</tr>
<tr>
<td>MAZD</td>
<td>FBmV</td>
<td>Maximum value generator as a function of time, double word</td>
<td>106</td>
</tr>
<tr>
<td>MOA</td>
<td>FBoV</td>
<td>Monostable element »abort«</td>
<td>107</td>
</tr>
<tr>
<td>MOK</td>
<td>FBoV</td>
<td>Monostable element »constant«</td>
<td>108</td>
</tr>
<tr>
<td>MUL2N</td>
<td>F</td>
<td>Word multiplication by 2 to the power of n</td>
<td>109</td>
</tr>
<tr>
<td>MUL2ND</td>
<td>F</td>
<td>Double word multiplication by 2 to the power of n</td>
<td>111</td>
</tr>
<tr>
<td>MULD</td>
<td>FBoV</td>
<td>Multiplication, double word</td>
<td>113</td>
</tr>
<tr>
<td>MULDI</td>
<td>F</td>
<td>Multiplication with division</td>
<td>114</td>
</tr>
<tr>
<td>MULW</td>
<td>F</td>
<td>Multiplication, word</td>
<td>115</td>
</tr>
<tr>
<td>MUXR8</td>
<td>FBoV</td>
<td>Word multiplexer with reset with max. 8 word outputs</td>
<td>116</td>
</tr>
<tr>
<td>MUXR16</td>
<td>FBoV</td>
<td>Word multiplexer with reset with max. 16 word outputs</td>
<td>116</td>
</tr>
<tr>
<td>MUXR32</td>
<td>FBoV</td>
<td>Word multiplexer with reset with max. 32 word outputs</td>
<td>116</td>
</tr>
<tr>
<td>MUXR64</td>
<td>FBoV</td>
<td>Word multiplexer with reset with max. 64 word outputs</td>
<td>116</td>
</tr>
<tr>
<td>MUXR256</td>
<td>FBoV</td>
<td>Word multiplexer with reset with max. 256 word outputs</td>
<td>116</td>
</tr>
<tr>
<td>MUXRD8</td>
<td>FBoV</td>
<td>Double word multiplexer with reset with max. 8 double word outputs</td>
<td>118</td>
</tr>
<tr>
<td>MUXRD16</td>
<td>FBoV</td>
<td>Double word multiplexer with reset with max. 16 double word outputs</td>
<td>118</td>
</tr>
<tr>
<td>CE Name</td>
<td>Type</td>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td>--------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>MUXRD32</td>
<td>FBoV</td>
<td>Double word multiplexer with reset with max. 32 double word outputs</td>
<td>118</td>
</tr>
<tr>
<td>MUXRD64</td>
<td>FBoV</td>
<td>Double word multiplexer with reset with max. 64 double word outputs</td>
<td>118</td>
</tr>
<tr>
<td>MUXRD256</td>
<td>FBoV</td>
<td>Double word multiplexer with reset with max. 256 double word outputs</td>
<td>118</td>
</tr>
<tr>
<td>NEGD</td>
<td>F</td>
<td>Negation, double word</td>
<td>120</td>
</tr>
<tr>
<td>NEGW</td>
<td>F</td>
<td>Negation, word</td>
<td>121</td>
</tr>
<tr>
<td>PACK</td>
<td>FBoV</td>
<td>Pack binary variables in word</td>
<td>122</td>
</tr>
<tr>
<td>PACKD</td>
<td>FBoV</td>
<td>Pack binary variables in double word</td>
<td>124</td>
</tr>
<tr>
<td>PDM</td>
<td>FBmV</td>
<td>Pulse duration modulator</td>
<td>126</td>
</tr>
<tr>
<td>PI</td>
<td>FBmV</td>
<td>PI controller</td>
<td>128</td>
</tr>
<tr>
<td>PIDT1</td>
<td>FBmV</td>
<td>PIDT1 controller</td>
<td>132</td>
</tr>
<tr>
<td>PT1</td>
<td>FBmV</td>
<td>PT1 controller</td>
<td>137</td>
</tr>
<tr>
<td>SETBD</td>
<td>FBoV</td>
<td>Set bit in double word</td>
<td>139</td>
</tr>
<tr>
<td>SETBW</td>
<td>FBoV</td>
<td>Set bit in word</td>
<td>140</td>
</tr>
<tr>
<td>SHIFTD</td>
<td>FBoV</td>
<td>Shift block, double word</td>
<td>141</td>
</tr>
<tr>
<td>SHIFTW</td>
<td>FBoV</td>
<td>Shift block, word</td>
<td>144</td>
</tr>
<tr>
<td>SIN1</td>
<td>FBoV</td>
<td>Sine</td>
<td>147</td>
</tr>
<tr>
<td>SUBD</td>
<td>FBoV</td>
<td>Subtraction, double word</td>
<td>149</td>
</tr>
<tr>
<td>SUBW</td>
<td>F</td>
<td>Subtraction, word</td>
<td>150</td>
</tr>
<tr>
<td>TESTBD</td>
<td>F</td>
<td>Test bit in double word</td>
<td>151</td>
</tr>
<tr>
<td>TESTBW</td>
<td>F</td>
<td>Test bit in word</td>
<td>152</td>
</tr>
<tr>
<td>UHR</td>
<td>FBmV</td>
<td>Display and set clock</td>
<td>153</td>
</tr>
<tr>
<td>UNPACK</td>
<td>FBoV</td>
<td>Unpacking a word into binary variables</td>
<td>156</td>
</tr>
<tr>
<td>UNPACKD</td>
<td>FBoV</td>
<td>Unpacking a double word into binary variables</td>
<td>158</td>
</tr>
<tr>
<td>USM</td>
<td>F</td>
<td>Switchover multiplexer</td>
<td>160</td>
</tr>
<tr>
<td>UST</td>
<td>FBoV</td>
<td>Switchover gate, word</td>
<td>161</td>
</tr>
<tr>
<td>USTD</td>
<td>FBoV</td>
<td>Switchover gate, double word</td>
<td>162</td>
</tr>
<tr>
<td>USTR</td>
<td>FBoV</td>
<td>Switchover gate with reset, word</td>
<td>163</td>
</tr>
<tr>
<td>USTRD</td>
<td>FBoV</td>
<td>Switchover gate with reset, double word</td>
<td>164</td>
</tr>
<tr>
<td>VGL3P</td>
<td>FBoV</td>
<td>Comparator with 3-point response</td>
<td>165</td>
</tr>
<tr>
<td>VGLEH</td>
<td>FBmV</td>
<td>Comparator with unilateral hysteresis</td>
<td>167</td>
</tr>
<tr>
<td>VGLUH</td>
<td>FBmV</td>
<td>Comparator with asymmetrical hysteresis</td>
<td>169</td>
</tr>
<tr>
<td>VRZ</td>
<td>FBmV</td>
<td>Up/down counter, word</td>
<td>171</td>
</tr>
<tr>
<td>VRZD</td>
<td>FBmV</td>
<td>Up/down counter, double word</td>
<td>173</td>
</tr>
<tr>
<td>VVZ</td>
<td>FBmV</td>
<td>Variable delay element</td>
<td>175</td>
</tr>
<tr>
<td>W2WDW</td>
<td>F</td>
<td>Conversion of two words into double word</td>
<td>176</td>
</tr>
<tr>
<td>WAES</td>
<td>FBmV</td>
<td>Write word in the event of value change</td>
<td>177</td>
</tr>
<tr>
<td>WDEC8</td>
<td>FBoV</td>
<td>Word decoder with max. 8 comparison values</td>
<td>178</td>
</tr>
</tbody>
</table>
## Overview of Blocks Arranged According to Their Call Names

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDEC16</td>
<td>FBoV</td>
<td>Word decoder with max. 16 comparison values</td>
<td>178</td>
</tr>
<tr>
<td>WDEC32</td>
<td>FBoV</td>
<td>Word decoder with max. 32 comparison values</td>
<td>178</td>
</tr>
<tr>
<td>WDEC64</td>
<td>FBoV</td>
<td>Word decoder with max. 64 comparison values</td>
<td>178</td>
</tr>
<tr>
<td>WDEC256</td>
<td>FBoV</td>
<td>Word decoder with max. 256 comparison values</td>
<td>178</td>
</tr>
<tr>
<td>WDW</td>
<td>F</td>
<td>Word to double word conversion</td>
<td>180</td>
</tr>
<tr>
<td>WOL</td>
<td>F</td>
<td>Read word, with enable</td>
<td>181</td>
</tr>
<tr>
<td>WOS</td>
<td>F</td>
<td>Write word, with enable</td>
<td>182</td>
</tr>
<tr>
<td>WUMC8</td>
<td>FBoV</td>
<td>Word recoder with max. 8 comparison values</td>
<td>183</td>
</tr>
<tr>
<td>WUMC16</td>
<td>FBoV</td>
<td>Word recoder with max. 16 comparison values</td>
<td>183</td>
</tr>
<tr>
<td>WUMC32</td>
<td>FBoV</td>
<td>Word recoder with max. 32 comparison values</td>
<td>183</td>
</tr>
<tr>
<td>WUMC64</td>
<td>FBoV</td>
<td>Word recoder with max. 64 comparison values</td>
<td>183</td>
</tr>
<tr>
<td>WUMC256</td>
<td>FBoV</td>
<td>Word recoder with max. 256 comparison values</td>
<td>183</td>
</tr>
</tbody>
</table>
ADDITION DOUBLE WORD

The value of the operand at input E1 is added to the value of the operand at input E2 and the result is assigned to the operand at output A.

The result is limited to the maximum or minimum value of the number range. If limiting occurred, a TRUE signal is assigned to the binary operand at output Q. If no limiting occurred, a FALSE signal is assigned to the binary operand at output Q.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Summand 1</td>
</tr>
<tr>
<td>E2</td>
<td>DINT</td>
<td>Summand 2</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Total</td>
</tr>
<tr>
<td>Q</td>
<td>BOOL</td>
<td>Total, limited</td>
</tr>
</tbody>
</table>

Description

The value of the operand at input E1 is added to the value of the operand at input E2 and the result is assigned to the operand at output A.

The result is limited to the maximum or minimum value of the number range (-2147483647 … 2147483647). If limiting occurred, a TRUE signal is assigned to the binary operand at output Q. If no limiting occurred, a FALSE signal is assigned to the binary operand at output Q.

The inputs and outputs can neither be duplicated nor negated.

Function call in IL

CAL ADDD1(E1 := ADDD_E1, E2 := ADDD_E2)

LD ADDD1.Q
ST ADDD .Q
LD ADDD1.A
ST ADDD_A

Function call in ST

ADDD1(E1 := ADDD_E1, E2 := ADDD_E2);
ADDD.Q:=ADDD1.Q;
ADDD_A:=ADDD1.A;
The values of the operands at the inputs of the function are added and the result is assigned to the operand at the output.

The result is limited to the maximum or minimum value of the number range.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Summand 1</td>
</tr>
<tr>
<td>E2</td>
<td>INT</td>
<td>Summand 2</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Total</td>
</tr>
</tbody>
</table>

### Description

The values of the operands at the inputs of the function are added and the result is assigned to the operand at the output.

The result is limited to the maximum value 32767 and the minimum value –32767.

### Function call in IL

```
LD      ADDW_E1
ADDW    ADDW_E2
ST      ADDW_A
```

### Function call in ST

```
ADDW_A:=ADDW(ADDW_E1,ADDW_E2);
```
ADDRESS SELECTION

One of the operands planned at the inputs AT0…ATn-1 is selected with this function block. Of this selected operand, the indirect address is generated and is provided at output ADR.

The ADRWA number indicates the maximum number of reference values. The following address selection blocks are available:

ADRWA8  Address selection with a maximum of 8 reference values
ADRWA16 Address selection with a maximum of 16 reference values

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>ADRWA(8..16)</td>
<td>Instance name</td>
</tr>
<tr>
<td>E</td>
<td>INT</td>
<td>Input value</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Number of reference values</td>
</tr>
<tr>
<td>EC0..ECn-1</td>
<td>INT</td>
<td>Input code</td>
</tr>
<tr>
<td>AT0..ATn-1</td>
<td>INT</td>
<td>Output table</td>
</tr>
<tr>
<td>E=EC</td>
<td>BOOL</td>
<td>Input value = input code</td>
</tr>
<tr>
<td>ADRE</td>
<td>DWORD</td>
<td>32 bit address</td>
</tr>
</tbody>
</table>

Description

Advantages of indirect addressing:

• In suitable applications, the PLC program is simplified substantially, thus reducing the planning effort.

• Access to any operands (multiplex function) is achieved with only one block (AWM or USM). In this process, the ADRWA block represents a powerful tool with which the operands to be accessed can be selected in a very flexible manner.

Reading/writing operands indirectly

The blocks AWM or USM use the 32 bit address generated by the block ADRWA in order to read or write the operand selected with the block ADRWA. Therefore, the block ADRWA and the block AWM or USM is needed to read or write operands indirectly. To do this, the operands to be read or written are listed at the inputs AT0 … ATn-1 and the read or write access it then performed by the AWM or USM block.
ADDRESS SELECTION

Selecting an operand from the output table
AT0 ... ATn-1

The block compares the value at input E successively against the values at the inputs EC0 ... ECn-1. The comparison is restarted each time the block is called, i.e. it begins with input EC0.

• If the value at input E agrees with one of the values at the inputs EC0 ... ECn-1:
  – The output E=EC is set to TRUE (hit).
  – The allocated operand is selected from the output table AT0 ... ATn-1.

• If the value at input E does not agree with one of the values at the inputs EC0 ... ECn-1:
  – The output E=EC is set to FALSE (no hit).
  – No operand is selected from the output table AT0 ... ATn-1.

Convention for allocation between EC0 ... ECn-1 and AT0 ... ATn-1:

- EC0 \rightarrow AT0
- EC1 \rightarrow AT1
  .
  .
- ECn-1 \rightarrow ATn-1

Generation of the 32 bit address

The 32 bit address of an operand is generated if it has been selected from the output table AT0 ... ATn-1. This is done by taking the address of the selected operand and by allocating it as a value to the operand specified at the ADR output. The value of the operand specified at the ADR output is therefore the address of the operand selected from the output table AT0 ... ATn-1.

Def.: An indirect address is an operand whose value is the address of another operand.

Use of an indirect address

The blocks AWM and USM access operands with 32 bit addressing. Therefore, the blocks AWM and USM require at the inputs the 32 bit address generated by the ADRWA block.

E

With the aid of the operand specified at input E and the operands specified at the inputs EC0 ... ECn-1 one of the operands is selected from the output table AT0 ... ATn-1 and its 32 bit address is then generated. To do this, the block compares the value at input E successively against the values at the inputs EC0 ... ECn-1. The comparison restarts each time the block is called, i.e. beginning with input EC0.

• If the value at input E agrees with one of the values at the inputs EC0 ... ECn-1:
  – The output E=EC is set to TRUE (hit).
  – The allocated operand is selected from the output table AT0 ... ATn-1 and its 32 bit address is generated.

• If the value at input E does not agree with one of the values at the inputs EC0 ... ECn-1:
  – The output E=EC is set to FALSE (no hit).
  – No operand is selected from the output table AT0 ... ATn-1 and accordingly no 32 bit address is generated either.

n

The number of planned inputs EC0 ... ECn-1 is specified at input n.

Example:
The following are planned: EC0, EC1, EC2 \rightarrow n = 3

EC0-....ECn-1

The operands for the comparison values are specified at the inputs EC0 ... ECn-1. The input EC0 can be duplicated. The value at input E is compared against these comparison values and checked for conformity. If they agree, the allocated operand is selected from the output table AT0 ... ATn-1 and its 32 bit address is generated. The values at the inputs EC0 ... ECn-1 are variable because they are normal operands. This is why the function for comparison between the value at input E and the values at the inputs EC0 ... ECn-1 is very flexible and powerful.

Allocation between EC0 ... ECn-1 and AT0 ... ATn-1:

- EC0 \rightarrow AT0
- EC1 \rightarrow AT1
  .
  .
- ECn-1 \rightarrow ATn-1

AT0-....ATn-1

The operands whose 32 bit addresses are to be generated are specified at the outputs AT0 ... ATn-1. When input EC0 is duplicated, input AT0 is also duplicated automatically.

E=EC

The output E=EC indicates whether or not the value at input E agrees with one of the values at the inputs EC0 ... ECn-1.

E=EC = FALSE \rightarrow No agreement
E=EC = TRUE \rightarrow The value at input E agrees with one of the values at the inputs EC0 ... ECn-1.
ADDRESS SELECTION

ADRE DWORD

The value of the operand at output ADR represents a 32 bit address. This is the 32 bit address of the operand selected from the output table AT0 ... ATn-1. The 32 bit address is produced by allocating the address of the selected operand as a value to the operand at output ADR.

If no agreement between input E and the inputs EC0 ... ECn-1 is determined during comparison, then no 32 bit address is generated either. Therefore, no value is assigned to output ADR. In this case, the ADR output is not updated.

Function call in IL

CAL ADRWA81(E := ADRWA_E, n := 3, EC0 := EC0, EC1 := EC1, EC2 := EC2, AT0 := AT0, AT1 := AT1, AT2 := AT2)
LD ADRWA81.ADRE
ST ADRWA_ADRE
LD ADRWA81.E_EC
ST ADRWA_E_EC

Note: The function call in IL has to be performed in one line.

Function call in ST

ADRWA81(E := ADRWA_E, n := 3, EC0 := EC0, EC1 := EC1, EC2 := EC2, AT0 := AT0, AT1 := AT1, AT2 := AT2); ADRWA_ADRE:=ADRWA81.ADRE;
ADRWA_E_EC:=ADRWA81.E_EC;
This block monitors the analog values present at input E0 capable of duplication for a change.

The AMELD number indicates the maximum number of input values. The following change annunciators are available:

- **AMELD8**: Analog value change annunciator with a maximum of 8 input values
- **AMELD16**: Analog value change annunciator with a maximum of 16 input values
- **AMELD32**: Analog value change annunciator with a maximum of 32 input values
- **AMELD127**: Analog value change annunciator with a maximum of 127 input values

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>AMELD(8..32)</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Block enabling</td>
</tr>
<tr>
<td>RESET</td>
<td>BOOL</td>
<td>Reset</td>
</tr>
<tr>
<td>n</td>
<td>INT</td>
<td>Number of input values</td>
</tr>
<tr>
<td>E0..En-1</td>
<td>INT</td>
<td>Input values; input can be duplicated</td>
</tr>
<tr>
<td>NR</td>
<td>INT</td>
<td>Number of the input value</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Current input value</td>
</tr>
<tr>
<td>AEND</td>
<td>BOOL</td>
<td>Change detected</td>
</tr>
</tbody>
</table>

**Description**

This block monitors the analog values present at the input E0..En-1 capable of duplication for a change.

The inputs and outputs cannot be negated/inverted. The input E0 can be duplicated.

**Recognition of a change**

Each time the block is processed, the current input values at the inputs E0 ... En-1 are successively compared against the historical values (input values from the previous processing of the block). If a change is recognized at one of the inputs E0 ... En-1:

- this is indicated at the AEND output
- the number of the input where the change was recognized is applied at output NR
- the changed input value is applied at output A

Each time the block is processed, a change at only one input is recognized. If a change is recognized, the inputs following the one where the change was previously determined are monitored the next time the block is processed.

**Initialization of historical values**

The first time the block is processed after PLC initialization (FREI = TRUE) or enabling of processing after it has been disabled (FREI changes from FALSE to TRUE), all current input values are assumed once as historical values and all outputs are set to the value FALSE. These initialized historical values now represent the starting basis for recognition of changes.

**FREI** BOO

Processing of the block is enabled with the FREI input.

- FREQ = FALSE → Block is not processed
- FREQ = TRUE → Processing of the block is enabled

If FREQ = FALSE, the outputs of the block are also no longer updated.

**RESET** BOO

The block can be reset with the RESET input.
ANALOG VALUE CHANGE ANNUNCIATOR, WORD

AMELD(8..127)

RESET = FALSE  →  No reset
RESET = TRUE   →  Reset of the block

A reset means:
– Adoption of the current values at the inputs E0 ... En-1 as historical values.
– All outputs are set to the value FALSE.

n INT

The number of values to be monitored at the inputs E0 ... En-1 is specified at input n.
Range for n: 1 ≤ n ≤ max. number (8..127)

E0...En-1 INT

The input E0 can be duplicated (E0 ... En-1). The operands to be monitored for a change are specified at the inputs E0 ... En-1.

NR INT

The serial number of the input E0 ... En-1 where a change has been determined is applied at output NR. If no output change is determined during processing of the block, the number of the last changed input is still applied at output NR.

The following allocation applies:

Change determined at E0  →  NR = 0
Change determined at E1  →  NR = 1
Change determined at En-1 → NR = n-1

A INT

If a change is determined at one of the inputs E0 ... En-1, the changing input value is assigned to output A. If no change is determined at the inputs E0 ... En-1 during processing of the block, the value of the last changed input is still applied at output A.

AEND BOOL

The output AEND indicates whether or not a change has been determined at the inputs E0 ... En-1.

AEND = FALSE  →  No change determined
AEND = TRUE   →  Change determined

Function call in IL

CAL AMELD81(FREI := AM_FREI, RESET := AM_RESET, n := 3, E0 := AM_E0, E1 := AM_E1, E2 := AM_E2)

LD AMELD81.A
ST AM_A
LD AMELD81.AEND
ST AM_AEND
LD AMELD81.NR
ST AM_NR

Note: In IL, the function call has to be performed in one line.

Function call in ST

AMELD81(FREI := AM_FREI, RESET := AM_RESET, n := 3, E0 := AM_E0, E1 := AM_E1, E2 := AM_E2);
AM_A:=AMELD81.A;
AM_AEND:=AMELD81.AEND;
AM_NR:=AMELD81.NR;
ANALOG VALUE CHANGE ANNUNCIATOR, DOUBLE WORD

This block monitors the analog values present at input E0 capable of duplication for a change.

The AMELD number indicates the maximum number of input values. The following change annunciators are available:

AMELDD8  Analog value change annunciator with a maximum of 8 input values
AMELDD16 Analog value change annunciator with a maximum of 16 input values
AMELDD32 Analog value change annunciator with a maximum of 32 input values
AMELDD63 Analog value change annunciator with a maximum of 63 input values

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>AMELD(8..32) Instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL Block enabling</td>
</tr>
<tr>
<td>RESET</td>
<td>BOOL Reset</td>
</tr>
<tr>
<td>N</td>
<td>INT Number of input values</td>
</tr>
<tr>
<td>E0..En-1</td>
<td>DINT Input values; input can be duplicated</td>
</tr>
<tr>
<td>NR</td>
<td>INT Number of the input value</td>
</tr>
<tr>
<td>A</td>
<td>DINT Current input value</td>
</tr>
<tr>
<td>AEND</td>
<td>BOOL Change detected</td>
</tr>
</tbody>
</table>

Description

This block monitors the analog values present at input E0..En-1 capable of duplication for a change.

The inputs and outputs cannot be negated/inverted. The input E0 can be duplicated.

Recognition of a change

Each time the block is processed, the current input values at the inputs E0 ... En-1 are successively compared against the historical values (input values from the previous processing of the block). If a change is recognized at one of the inputs E0 ... En-1:

- this is indicated at the AEND output
- the number of the input where the change was recognized is applied at output NR
- the changed input value is applied at output A

Each time the block is processed, a change at only one input is recognized. If a change is recognized, the inputs following the one where the change was previously determined are monitored the next time the block is processed.

Initialization of historical values

The first time the block is processed after PLC initialization (FREI = TRUE) or enabling of processing after it has been disabled (FREI changes from FALSE to TRUE), all current input values are assumed once as historical values and all outputs are set to the value FALSE. These initialized historical values now represent the starting basis for recognition of changes.

FREI

BOOL

Processing of the block is enabled with the FREI input.

FREI = FALSE → Block is not processed
FREI = TRUE   → Processing of the block is enabled

If FREI = FALSE, the outputs of the block are also no longer updated.
ANALOG VALUE CHANGE ANNUNCIATOR, DOUBLE WORD

RESET

The block can be reset with the RESET input.

RESET = FALSE → No reset
RESET = TRUE → Reset of the block

A reset means:
- Adoption of the current values at the inputs E0...En-1 as historical values.
- All outputs are set to the value FALSE.

n

The number of values to be monitored at the inputs E0 ... En-1 is specified at input n.

Range for n: 1 ≤ n ≤ max. number (8..63)

E0...En-1

The input E0 can be duplicated (E0 ... En-1). The operands to be monitored for a change are specified at the inputs E0 ... En-1.

NR

The serial number of the input E0 ... En-1 where a change has been determined is applied at output NR. If no output change is determined during the processing of the block, the number of the last changed input is still applied at output NR.

The following allocation applies:
Change determined at E0 → NR = 0
Change determined at E1 → NR = 1
Change determined at En-1 → NR = n-1

A

DOUBLE WORD

If a change is determined at one of the inputs E0...En-1, the changed input value is assigned to output A. If no change is determined at the inputs E0 ... En-1 during the processing of the block, the value of the last changed input is still applied at output A.

AEND

The output AEND indicates whether or not a change has been determined at the inputs E0 ... En-1.

AEND = FALSE → No change determined
AEND = TRUE → Change determined

Function call in IL

CAL AMELDD81(FREI := AMD_FREI, RESET := AMD_RESET, n := 3, E0 := AMD_E0, E1 := AMD_E1, E2 := AMD_E2)

LD AMELDD81.A
ST AMD_A
LD AMELDD81.AEND
ST AMD_AEND
LD AMELDD81.NR
ST AMD_NR

Note: In IL, the function call has to be performed in one line.

Function call in ST

AMELDD81(FREI := AMD_FREI, RESET := AMD_RESET, n := 3, E0 := AMD_E0, E1 := AMD_E1, E2 := AMD_E2);
AMD_A := AMELDD81.A;
AMD_AEND := AMELDD81.AEND;
AMD_NR := AMELDD81.NR;
OFF DELAY

The TRUE/FALSE edge of input E is delayed by the time T and is output as a TRUE/FALSE edge at output A.

If input E returns to the TRUE level before the time T is expired, output A remains in the TRUE level.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>ASV</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>BOOL</td>
<td>Input signal</td>
</tr>
<tr>
<td>T</td>
<td>TIMER</td>
<td>Delay time</td>
</tr>
<tr>
<td>A</td>
<td>BOOL</td>
<td>Delayed signal</td>
</tr>
</tbody>
</table>

Description

The TRUE/FALSE edge of input E (input 0HT in the figure) is delayed by the time T and is output as a TRUE/FALSE edge at output A (output Q in the figure).

If input E returns to the TRUE level before the time T is expired, output A remains in the TRUE level.

Maximum time offset at the output: < 1 cycle time

Reasonable range for T: > 1 cycle time

The inputs and the output can neither be duplicated nor inverted.

General behavior

Started timers are processed by the PLC’s operating system and are therefore completely independent of PLC program processing. The operating system does not issue a corresponding message to the corresponding timer block in the PLC program until the timer has elapsed.

Function call in IL

CAL ASV1(E := ASV_E, T := T#200ms)
LD ASV1.A
ST ASV_A

Function call in ST

ASV1(E := ASV_E, T := T#200ms);
ASV_A:=ASV1.A;
This block reads the value of an operand, using the method of indirect addressing. The value read is assigned to output A.

Note:
The AWM block can only be used meaningfully in conjunction with the ADRWA block.

### Block type

Function

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADRESSE</td>
<td>DWORD</td>
<td>Indirect address of the operand to be read</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Value of the operand read</td>
</tr>
</tbody>
</table>

### Description

This block reads the value of an operand, using the method of indirect addressing. The value read is assigned to output A.

Note:
The AWM block can only be used meaningfully in conjunction with the ADRWA block.

The value of the operand at input ADRESSE is interpreted as an address of the operand to be read (indirect addressing).

This indirect address is generated by the ADRWA block.

The input and output can neither be duplicated nor negated.

Note:
An explanation of the indirect addressing method and the possibilities of using the AWM block are described in the section dealing with the ADRWA block.

### Function call in IL

```
LD    AWM_ADR
AWM
ST    AWM_A
```

### Function call in ST

```
AWM_A:=AWM( AWM_ADR);
```
The positive BCD coded number at input E is converted to a binary number and is assigned to the operand at output A.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>INT</td>
<td>BCD coded number</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Binary number</td>
</tr>
</tbody>
</table>

**Description**

The input and the output can neither be duplicated nor negated.

**Definition:**

The significance of the digits in a BCD coded number and a hexadecimal number is defined as follows:

**BCD NUMBER**

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>11</th>
<th>7</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z4</td>
<td></td>
<td></td>
<td>Z3</td>
<td>Z2</td>
<td>Z1</td>
</tr>
</tbody>
</table>

Numerical value:

- \( Z1 \times 1 \)
- \( Z2 \times 10 \)
- \( Z3 \times 100 \)
- \( Z4 \times 1000 \)
- \( 0 \leq Z_i \leq 9 \)

**HEXDEC NUMBER**

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>11</th>
<th>7</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z4</td>
<td></td>
<td></td>
<td>Z3</td>
<td>Z2</td>
<td>Z1</td>
</tr>
</tbody>
</table>

Numerical value:

- \( Z1 \times 1 \)
- \( Z2 \times 16 \)
- \( Z3 \times 256 \)
- \( Z4 \times 4096 \)
- \( 0 \leq Z_i \leq F \)

**Remark:**

At the BCD input, the block additionally accepts digits to which the following applies:

- \( 0 \leq Z_i \leq F \)
Example 1

<table>
<thead>
<tr>
<th>BCD NUMBER</th>
<th>HEXDEC NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 4 D 2</td>
<td></td>
</tr>
<tr>
<td>1 2 3 4</td>
<td></td>
</tr>
</tbody>
</table>

Z1 = 4 * 1 = 4
Z2 = 3 * 10 = 30
Z3 = 2 * 100 = 200
Z4 = 1 * 1000 = 1000

Z1 = 2 * 1 = 2
Z2 = 13 * 16 = 208
Z3 = 4 * 256 = 1024
Z4 = 0 * 4096 = 0

Example 2

<table>
<thead>
<tr>
<th>BCD NUMBER</th>
<th>HEXDEC NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 8 7 2</td>
<td></td>
</tr>
<tr>
<td>A 2 F 4</td>
<td></td>
</tr>
</tbody>
</table>

Z1 = 4 * 1 = 4
Z2 = 15 * 10 = 150
Z3 = 2 * 100 = 200
Z4 = 10 * 1000 = 10000

Z1 = 2 * 1 = 2
Z2 = 7 * 16 = 112
Z3 = 8 * 256 = 2048
Z4 = 2 * 4096 = 8192

Representation of a negative BCD number

A negative BCD number can be represented in the PLC by separate representation of the value and the sign. In doing so, the value of the BCD number is stored in a word variable and the information about the sign is stored in a binary variable.

Function call in IL

LD  BCD_E
BCDDUAL
ST  BCD_A

Function call in ST

BCD_A:=BCDDUAL(BCD_E);
The positive BCD coded number at input E is converted to a binary number and assigned to the operand at output A.

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>E</th>
<th>DINT</th>
<th>BCD coded number</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>DINT</td>
<td>Binary number</td>
</tr>
</tbody>
</table>

**Description**

The positive BCD coded number at input E is converted to a binary number and assigned to the operand at output A.

The input and the output can neither be duplicated nor negated.

**Definition:**

The significance of the digits in a BCD coded number and a hexadecimal number is defined as follows:

<table>
<thead>
<tr>
<th>BCD NUMBER</th>
<th>HEXDEC NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Z8</td>
<td>Z1</td>
</tr>
<tr>
<td>Z7</td>
<td>Z2</td>
</tr>
<tr>
<td>Z6</td>
<td>Z3</td>
</tr>
<tr>
<td>Z5</td>
<td>Z4</td>
</tr>
<tr>
<td>Z4</td>
<td>Z5</td>
</tr>
<tr>
<td>Z3</td>
<td>Z6</td>
</tr>
<tr>
<td>Z2</td>
<td>Z7</td>
</tr>
<tr>
<td>Z1</td>
<td>Z8</td>
</tr>
</tbody>
</table>

Numerical value:

- Z1 * 1
- Z2 * 10
- Z3 * 100
- Z4 * 1000
- Z5 * 10000
- Z6 * 100000
- Z7 * 1000000
- Z8 * 10000000

0 ≤ Zi ≤ 9

Numerical value:

- Z1 * 1
- Z2 * 16
- Z3 * 256
- Z4 * 4096
- Z5 * 65536
- Z6 * 1048576
- Z7 * 16777216
- Z8 * 268435456

0 ≤ Zi ≤ F

**Remark:**

At the BCD input, the block additionally accepts digits to which the following applies:

0 ≤ Zi ≤ F
Example 1

<table>
<thead>
<tr>
<th>BCD NUMBER</th>
<th>BIT</th>
<th>HEXDEC NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 2 3 4 5 6 7 8</td>
<td>Z1 = 8 * 1 = 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z2 = 7 * 10 = 70</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z3 = 6 * 100 = 600</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z4 = 5 * 1000 = 5000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z5 = 4 * 10000 = 40000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z6 = 3 * 100000 = 300000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z7 = 2 * 1000000 = 2000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z8 = 1 * 10000000 = 10000000</td>
<td></td>
</tr>
<tr>
<td>12345678</td>
<td></td>
<td>0 0 1 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td></td>
<td>Z2 = 4 * 16 = 64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z3 = 1 * 256 = 256</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z4 = 6 * 4096 = 24576</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z5 = 12 * 65536 = 786432</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z6 = 11 * 1048576 = 11534336</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z7 = 0 * 16777216 = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z8 = 0 * 268435456 = 0</td>
<td></td>
</tr>
<tr>
<td>12345678</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 2

<table>
<thead>
<tr>
<th>BCD NUMBER</th>
<th>BIT</th>
<th>HEXDEC NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 2 3 4 5 6 7 8</td>
<td>Z1 = 4 * 1 = 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z2 = 15 * 10 = 150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z3 = 2 * 100 = 200</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z4 = 10 * 1000 = 10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z5 = 4 * 10000 = 40000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z6 = 3 * 100000 = 300000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z7 = 11 * 1000000 = 11000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z8 = 1 * 10000000 = 10000000</td>
<td></td>
</tr>
<tr>
<td>21350354</td>
<td></td>
<td>0 0 1 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td></td>
<td>Z2 = 13 * 16 = 208</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z3 = 7 * 256 = 1792</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z4 = 12 * 4096 = 49152</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z5 = 5 * 65536 = 327680</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z6 = 4 * 1048576 = 4194304</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z7 = 1 * 16777216 = 16777216</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z8 = 0 * 268435456 = 0</td>
<td></td>
</tr>
<tr>
<td>21350354</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Representation of a negative BCD number

A negative BCD number can be represented in the PLC by separate representation of the value and the sign. In doing so, the value of the BCD number is stored in a double word variable and the information about the sign is stored in a binary variable.

Function call in IL

LD BCDD_E
BCDDUALD
ST BCDD_A

Function call in ST

BCDD_A:=BCDDUALD(BCDD_E);
The value of the operand at input E is limited to the range between the upper and lower limits.

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>INT</td>
<td>Input value</td>
</tr>
<tr>
<td>OG</td>
<td>INT</td>
<td>Upper limit</td>
</tr>
<tr>
<td>UG</td>
<td>INT</td>
<td>Lower limit</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Limited value</td>
</tr>
</tbody>
</table>

**Description**

The value of the operand at input E is limited to the range between the upper and lower limits.

The upper limit is specified by the operand at the OG input and the lower limit is specified by the one at the UG input.

The following applies:

\[
A = UG \text{ for } E < UG \\
A = E \text{ for } UG < E < OG \\
A = OG \text{ for } E > OG
\]

The inputs and the output can neither be duplicated nor negated.

**Function call in IL**

LD \texttt{BEG}_E  
BEG \texttt{BEG}_OG,\texttt{BEG}_UG  
ST \texttt{BEG}_A

**Function call in ST**

\texttt{BEG}_A:=\texttt{BEG}(\texttt{BEG}_E, \texttt{BEG}_OG,\texttt{BEG}_UG);
**LIMITER, DOUBLE WORD**

The value of the operand at input \( E \) is limited to the range between the upper and lower limits.

**Block type**

Function

**Parameters**

- **E**: DINT, Input value
- **OG**: DINT, Upper limit
- **UG**: DINT, Lower limit
- **A**: DINT, Limited value

**Description**

The value of the operand at input \( E \) is limited to the range between the upper and lower limits.

The upper limit is specified by the operand at the \( OG \) input and the lower limit is specified by the one at the \( UG \) input.

The following applies:

- \( A = UG \) for \( E < UG \)
- \( A = E \) for \( UG < E < OG \)
- \( A = OG \) for \( E > OG \)

The inputs and the output can neither be duplicated nor negated.

**Function call in IL**

- LD BEGD_E
- BEGD BEGD_OG,BEGD_UG
- ST BEGD_A

**Function call in ST**

\[
\text{BEGD_A} := \text{BEGD} \left( \text{BEGD_E}, \text{BEGD_OG}, \text{BEGD_UG} \right)
\]
BIT SEARCHER

This block searches through a bit field for a set bit. The bit field consists of successive word operands. If a set bit is found, this is indicated at the block’s outputs.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>BITSU</td>
<td>Instance name</td>
</tr>
<tr>
<td>RESET</td>
<td>BOOL</td>
<td>Reset of the block and new search</td>
</tr>
<tr>
<td>R_V</td>
<td>BOOL</td>
<td>Search direction, down/up</td>
</tr>
<tr>
<td>ANZ</td>
<td>INT</td>
<td>Field length, number of word operands</td>
</tr>
<tr>
<td>ANF</td>
<td>INT</td>
<td>Field start, 1st word operand</td>
</tr>
<tr>
<td>BIT</td>
<td>INT</td>
<td>Number of set bits to be skipped</td>
</tr>
<tr>
<td>END</td>
<td>BOOL</td>
<td>Field end reached</td>
</tr>
<tr>
<td>NR</td>
<td>INT</td>
<td>Number of the word operand containing the set bit</td>
</tr>
<tr>
<td>POS</td>
<td>INT</td>
<td>Bit position within the word operand</td>
</tr>
</tbody>
</table>

Description

This block searches through a bit field for a set bit. The bit field consists of successive word operands. If a set bit is found, this is indicated at the block’s outputs.

The inputs and outputs can neither be duplicated nor negated/inverted.

The bit field consists of successive word operands. The first word operand in the bit field is specified at input ANF.

After the start of the PLC program (1st program cycle), the block sets all outputs to the value 0 or FALSE and immediately begins searching for set bits. Therefore, the search is already active in the first program cycle.

If the block finds a set bit, its position is indicated at the block’s outputs and the search is ended. The next time the block is called in the next program cycle, it continues the search, doing so at the bit position directly following the bit found last. If the end of the bit field is reached during the search, this is indicated and a new search can be started again by applying a reset signal. The new search again begins at the start of the bit field.

RESET

The input RESET is used to reset the block and start a new search from the beginning of the bit field.

RESET = FALSE → No reset
RESET = TRUE → Reset is triggered

A TRUE signal at input RESET resets all of the block’s outputs to the value 0 or FALSE and results in immediate starting of a new search from the beginning of the bit field. Therefore, the new search begins in the same processing cycle in which the block’s outputs are reset.

The search within the reset cycle ends either at the first set bit or, if no bit is set, at the end of the field. The next time the block is called, the reset input must be set to 0 or FALSE so that the block will be able to continue searching through the bit field for set bits. The reset input is dominant with respect to the other inputs.

R_V

The searching direction is specified at input R_V.

R_V = FALSE → Search up
R_V = TRUE → Search down

R_V = FALSE means: The start of the bit field within the meaning of the search is identical with the physical start of the bit field.

R_V = TRUE means: The start of the bit field within the meaning of the search is identical with the physical end of the bit field.

ANZ

The number of word operands of which the bit field consists is specified at input ANZ.
ANF INT
The word operand with which the bit field physically begins is specified at input ANF. The entire bit field consists of the operand at input ANF and the subsequent operands corresponding to the operand numbering. The total number of word operands is specified at input ANZ.

BIT INT
The way in which the block is to indicate the bits set in the bit field is specified at input BIT.

BIT = 1 → Each set bit in the bit field is indicated at the outputs NR and POS.
BIT = 2 → The first set bit in the bit field is indicated. Of the other set bits in the bit field, only every second set bit is now indicated.
BIT = 3 → The first set bit in the bit field is indicated. Of the other set bits in the bit field, only every third set bit is now indicated.
BIT = n → The first set bit in the bit field is indicated. Of the other set bits in the bit field, only every n-th set bit is now indicated.

Special function: BIT = 0
If the block finds a set bit, in the next program cycle the search is not automatically continued at the subsequent bit position. The search is interrupted at this point until the bit found has assumed the significance 0. During interruption of the search, the position of the bit found last continues to be indicated at the outputs. If the bit found last assumes the value 0, the search is continued at the next bit position. If a further bit is set in the selected search direction, this bit's position is indicated and the search is interrupted again.

If no more bits are set in the search direction, the position of the bit found last is indicated. The output END (field end reached) is set to TRUE.

If no bits are set after a reset, the block runs through the bit field up to its end and stops there. The output END (field end reached) is set to TRUE. The outputs NR and POS are set to 0.

If bits are set after a reset, the block indicates the first set bit in the search direction and stops at this bit.

END BOOL
The output END indicates whether or not the end of the bit field has been reached during the search.

END = FALSE → End of bit field not reached
END = TRUE → End of bit field reached

Search up:
The end of the bit field is defined by bit position 15 in the last word operand of the physical bit field.

Search down:
The end of the bit field is defined by bit position 0 in the first word operand of the physical bit field.

If the bit is set at the last bit position of the field and it is found and indicated during a search, the output END is not yet set to the value TRUE. This is not done until the next processing cycle of the block. Therefore, the prerequisite for setting the output END is that no set bit has been found during the current search.

NR, POS INT
If the block finds a set bit in the planned bit field, its position is indicated at the outputs NR and POS.

Meanings:
NR: Current number of the word operand in which the set bit has been found.
POS: Position of the set bit within the word operand.

Current number of the word operand

NR = 0 → 1st word operand of the bit field
NR = 1 → 2nd word operand of the bit field
... .
NR = -1 → nth word operand of the bit field

Position of the set bit within the word operand

Numbering within a word operand is from 0 ... 15. Position 0 corresponds to the least significant bit and position 15 to the most significant bit within the word operand. If the end of the field is reached during a search without a set bit having been found, the position of the bit found last continues to be output at the outputs NR and POS. This is performed until a new search is forced from the start of the bit field by a reset at input RESET.

If no bits are set at all within the entire bit field, the outputs assume the following values at the end of the first search:
END = TRUE
NR = 0
POS = 0

This state can be terminated again by a TRUE signal at input RESET.
Function call in IL

```
CAL BITSU1(RESET := BITS_RES,
       R_V := BITS_RV, ANZ := BITS_ANZ,
       BIT := BITS_BIT, ANF := BITS_ANF)
LD BITSU1.NR
ST BITS_NR
LD BITSU1.POS
ST BITS_POS
LD BITSU1.END
ST BITS_END
```

Note: In IL, the function call has to be performed in one line.

Function call in ST

```
BITSU1(RESET := BITS_RES, R_V := BITS_RV,
ANZ := BITS_ANZ, BIT := BITS_BIT,
ANF := BITS_ANF);
BITS_NR:=BITSU1.NR;
BITS_POS:=BITSU1.POS;
BITS_END:=BITSU1.END;
```
This block monitors the binary values present at input E0 capable of duplication for a change. The BMELD number indicates the maximum number of input values. The following change annunciators are available:

- **BMELD8**: Binary value change annunciator with a maximum of 8 input values
- **BMELD16**: Binary value change annunciator with a maximum of 16 input values
- **BMELD32**: Binary value change annunciator with a maximum of 32 input values
- **BMELD127**: Binary value change annunciator with a maximum of 127 input values

**Block type**

Function block with historical values

**Parameters**

- **Instance**: BMELD(8..32)
- **FREI**: BOOL (Block enabling)
- **RESET**: BOOL (Reset)
- **N**: INT (Number of input values)
- **E0 ... En-1**: INT (Input values; input can be duplicated)
- **NR**: INT (Number of the input value)
- **A**: INT (Current input value)
- **AEND**: BOOL (Change detected)

**Description**

This block monitors the binary values present at the input E0 ... En-1 capable of duplication for a change. The inputs and outputs cannot be negated/inverted. The input E0 can be duplicated.

**Recognition of a change**

Each time the block is processed, the current input values at the inputs E0 ... En-1 are successively compared against the historical values (input values from the previous processing of the block). If a change is recognized at one of the inputs E0 ... En-1:

- this is indicated at output AEND
- the number of the input where the change was recognized is applied at output NR
- the changed input value is applied at output A

Each time the block is processed, a change at only one input is recognized. If a change is recognized, the inputs following the one where the change was previously determined are monitored the next time the block is processed.

**Initialization of historical values**

The first time the block is processed after PLC initialization (FREI = TRUE) or enabling of processing after it has been disabled (FREI changes from FALSE to TRUE), all current input values are assumed once as historical values and all outputs are set to the value 0. These initialized historical values now represent the starting basis for recognition of changes.

**FREI**: BOOL

Processing of the block is enabled with the FREI input.

FREI = FALSE → Block is not processed
FREI = TRUE → Processing of the block is enabled

If FREI = FALSE, the outputs of the block are also no longer updated.
**RESET**  \( \text{BOOL} \)

The block can be reset with the RESET input.

- **RESET = FALSE** \( \rightarrow \) No reset
- **RESET = TRUE** \( \rightarrow \) Reset of the block

A reset means:
- Adoption of the current values at the inputs \( E_0 \ldots E_{n-1} \) as historical values.
- All outputs are set to the value FALSE (0).

**n**  \( \text{INT} \)

The number of values to be monitored at the inputs \( E_0 \ldots E_{n-1} \) is specified at input \( n \).

Range for \( n: 1 \leq n \leq \text{max. number (8..127)} \)

**E0...En-1**  \( \text{BOOL} \)

The input \( E_0 \) can be duplicated (\( E_0 \ldots E_{n-1} \)). The operands to be monitored for a change are specified at the inputs \( E_0 \ldots E_{n-1} \).

**Function call in IL**

```
CAL BMELD81(FREI := BM_FREI,
             RESET := BM_RES, n := 3, E0 := BM_E0,
             E1 := BM_E1, E2 := BM_E2)
```

LD BMELD81.A
ST BM_A
LD BMELD81.AEND
ST BM_AEND
LD BMELD81.NR
ST BM_NR

Note: In IL, the function call has to be performed in one line.

**NR**  \( \text{INT} \)

The serial number of the input \( E_0 \ldots E_{n-1} \) where a change has been determined is applied at output NR. If no change is determined during processing of the block, the number of the last changed input is still applied at output NR.

The following allocation applies:

- Change determined at \( E_0 \rightarrow NR = 0 \)
- Change determined at \( E_1 \rightarrow NR = 1 \)
- Change determined at \( E_{n-1} \rightarrow NR = n-1 \)

**A**  \( \text{BOOL} \)

If a change is determined at one of the inputs \( E_0 \ldots E_{n-1} \), the changed input value is assigned to output A. If no change is determined at the inputs \( E_0 \ldots E_{n-1} \) during processing of the block, the value of the last changed input is still applied at output A.

**AEND**  \( \text{BOOL} \)

The output AEND indicates whether or not a change has been determined at the inputs \( E_0 \ldots E_{n-1} \).

- **AEND = FALSE** \( \rightarrow \) No change determined
- **AEND = TRUE** \( \rightarrow \) Change determined

**Function call in ST**

```
BMELD81(FREI := BM_FREI, RESET := BM_RES,
        n := 3, E0 := BM_E0, E1 := BM_E1, E2 := BM_E2);
BM_A:=BMELD81.A;
BM_AEND:=BMELD81.AEND;
BM_NR:=BMELD81.NR;
```
COMPARISON OF TWO DATA BLOCKS

This function compares two data blocks on the addresses QADR and ZADR with a length of exactly ANZ bytes.

Block type
Function

Parameters
ANZ INT Number of bytes to be compared
QADR DWORD 32 bit address of the begin of the source area
ZADR DWORD 32 bit address of the begin of the target area
Return INT Result of comparison

Description
The COMPARE function compares two data blocks on the addresses QADR and ZADR with a length of exactly ANZ bytes.

Since the bytes are compared as unsigned char, COMPARE supplies one of the following values:
< 0 if the block on QADR is less than the block on ZADR.
= 0 if the first ANZ characters as of QADR and ZADR are equal to each other.
> 0 if the block on QADR is greater than the block on ZADR.

Function call in IL
LD Var1
ADR
ST DWORD_0
LD Var2
ADR
ST DWORD_1
LD Comp_Anz
COMPARE DWORD_0, DWORD_1
ST Comp_Erg

Function call in ST
Comp_Erg := COMPARE(COMP_ANZ,
ADR(var1),
ADR(var2));
This block copies \( n \) words from a source memory area into a target memory area.

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Block enabling</td>
</tr>
<tr>
<td>ANZ</td>
<td>INT</td>
<td>Number of word(s) to be copied</td>
</tr>
<tr>
<td>QADR</td>
<td>DWORD</td>
<td>32 bit address of the start of the source area</td>
</tr>
<tr>
<td>ZADR</td>
<td>DWORD</td>
<td>32 bit address of the start of the target area</td>
</tr>
</tbody>
</table>

**Description**

This block copies \( n \) words from a source memory area into a target memory area.

The contents of the source memory area are not changed.

In each case, the start of the source and target memory areas is specified at the block's inputs by means of the 32 bit address.

The inputs and outputs can neither be duplicated nor negated/inverted.

**ANZ**

Number \( n \) of words to be copied.

The following applies: \( 0 \leq n \leq +8000\)H

\( n = 0 \): No copying

\( n = 8000\)H: A complete segment (64 kbytes) is copied

**QADR**

32 bit address of the start of the source area.

**ZADR**

32 bit address of the start of the target area.

Note:
If the COPY function block is located in the PLC program with the lower priority, programming can be interrupted by the program with the higher priority. An interrupt can be prevented by planning the COPY block in the PLC program with the higher priority.

**Function call in IL**

\[
\begin{align*}
LD & \quad MW0 \\
ADR & \\
ST & \quad _DWORD_0 \\
LD & \quad MW1 \\
ADR & \\
ST & \quad _DWORD_1 \\
LD & \quad COPY\_FREI \\
COPY & \quad COPY\_ANZ, _DWORD_0, _DWORD_1
\end{align*}
\]

**Function call in ST**

\[
\begin{align*}
_ADWORD_0:=ADR(MW0); \\
_ADWORD_1:=ADR(MW1); \\
COPY(COPY\_FREI,COPY\_ANZ, _DWORD_0, _DWORD_1);
\end{align*}
\]
COSINE COS1

The block calculates the cosine value from input ANG and assigns it to output AD.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>COS1</th>
<th>Instance name</th>
<th>ANG</th>
<th>INT</th>
<th>Angle 0 ... 3600 (corresponds to 0.0°...360.0°)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD</td>
<td>DINT</td>
<td>Cosine of the input value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error, if input value is negative or greater than 3600</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description
The block calculates the cosine value from input ANG and assigns it to output AD. The result is within the range of -100,000 to +100,000. If the value at ANG is negative or greater than 3600 (360°), output AD is set to 0 and output ERR is set to TRUE.

The maximum deviation of the result is ±0.5.

**ANG**

The cosine value from the input operand ANG is calculated. The result is available at the output operand AD.

Input:
- 0000 for 0 degrees
- 0001 for 0.1 degrees
- 0010 for 1.0 degrees
- ...
- 3600 for 360.0 degrees

**AD**

The cosine value of the input ANG is available at the output AD.

**ERR**

This output ERR indicates whether the input value is within the correct range (0 ≤ ANG ≤ 3600).

Input 0 ≤ ANG ≤ 3600
→ ERR = FALSE und AD = COS(ANG)

Input ANG < 0 or ANG > 3600
→ ERR = TRUE and AD = 0

Examples for cosine values

<table>
<thead>
<tr>
<th>x angle</th>
<th>cos (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>180°</td>
<td>-100000</td>
</tr>
<tr>
<td>236°</td>
<td>-55919</td>
</tr>
<tr>
<td>90°</td>
<td>0</td>
</tr>
<tr>
<td>270°</td>
<td>0</td>
</tr>
<tr>
<td>0°</td>
<td>100000</td>
</tr>
<tr>
<td>360°</td>
<td>100000</td>
</tr>
<tr>
<td>45°</td>
<td>70711</td>
</tr>
</tbody>
</table>
Function call in IL
CAL COS11(ANG := COS_ANG)
LD COS11.ERR
ST COS_ERR
LD COS11.AD
ST COS_AD

Function call in ST
COS11(ANG := COS_ANG);
COS_ERR:=COS11.ERR;
COS_AD:=COS11.AD;
BIT COUNTER, DOUBLE WORD

The set bits of the operand applied to input E are counted and output to the output Anz. At output A0 the number of the lowest set bit is output and at the output A1 the number of the highest set bit is output.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>COUNTBD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>DINT</td>
<td>Input value</td>
</tr>
<tr>
<td>Anz</td>
<td>INT</td>
<td>Number of set bits</td>
</tr>
<tr>
<td>A0</td>
<td>INT</td>
<td>Number of the lowest set bit</td>
</tr>
<tr>
<td>A1</td>
<td>INT</td>
<td>Number of the highest set bit</td>
</tr>
</tbody>
</table>

Description
The set bits of the operand applied to input E are counted and output to the output Anz. At output A0 the number of the lowest set bit is output and at the output A1 the number of the highest set bit is output.

For E = 0 the following applies:

- Anz = 0
- A0 = 0
- A1 = 0

The inputs and the output can neither be duplicated nor negated.

Function call in IL

```
CAL COUNTBD1(E := COD_E)
LD COUNTBD1.A0
ST COD_A0
LD COUNTBD1.A1
ST COD_A1
LD COUNTBD1.Anz
ST COD_ANZ
```

Function call in ST

```
COUNTBD1(E := COD_E);
COD_A0:=COUNTBD1.A0;
COD_A1:=COUNTBD1.A1;
COD_ANZ:=COUNTBD1.Anz;
```
BIT COUNTER, WORD

The set bits of the operand applied to input E are counted and output to the output Anz. At output A0 the number of the lowest set bit is output and at the output A1 the number of the highest set bit is output.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>COUNTBW</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>INT</td>
<td>Input value</td>
</tr>
<tr>
<td>Anz</td>
<td>INT</td>
<td>Number of set bits</td>
</tr>
<tr>
<td>A0</td>
<td>INT</td>
<td>Number of the lowest set bit</td>
</tr>
<tr>
<td>A1</td>
<td>INT</td>
<td>Number of the highest set bit</td>
</tr>
</tbody>
</table>

Description

The set bits of the operand applied to input E are counted and output to the output Anz. At output A0 the number of the lowest set bit is output and at the output A1 the number of the highest set bit is output.

For E = 0 the following applies:

Anz = 0
A0 = 0
A1 = 0

The inputs and the output can neither be duplicated nor negated.

Function call in IL

CAL COUNTBW1(E := COW_E)
LD COUNTBW1.A0
ST COW_A0
LD COUNTBW1.A1
ST COW_A1
LD COUNTBW1.Anz
ST COW_ANZ

Function call in ST

COUNTBW1(E := COW_E);
COW_A0:=COUNTBW1.A0;
COW_A1:=COUNTBW1.A1;
COW_ANZ:=COUNTBW1.Anz;
The High-speed counter in the central unit 07 KT 97 works independently of the user program. Therefore, the counter can react to external signals fast. It can be used in seven different configurable operating modes. The operating mode is configured in the system constant KW 85.02 / %MW3085.2. The planned operating mode is only activated in the initialization phase (power on, cold start, warm start). The same function block COUNTW is used for all operating modes.

### Block type

Function block with historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>COUNTW</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>BOOL</td>
<td>Counter number: NO=FALSE ... counter0; NO=TRUE ... counter1</td>
</tr>
<tr>
<td>UD</td>
<td>BOOL</td>
<td>Counting direction: UD=FALSE ... up; UD=TRUE ... down</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the counting process</td>
</tr>
<tr>
<td>SET</td>
<td>BOOL</td>
<td>Set to initial value</td>
</tr>
<tr>
<td>STA</td>
<td>INT</td>
<td>Start value</td>
</tr>
<tr>
<td>END</td>
<td>INT</td>
<td>End value</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error</td>
</tr>
<tr>
<td>STAT</td>
<td>INT</td>
<td>Status</td>
</tr>
<tr>
<td>CF</td>
<td>BOOL</td>
<td>Carry flag: CF=TRUE ... End value reached</td>
</tr>
<tr>
<td>Istwert</td>
<td>INT</td>
<td>Count (actual value)</td>
</tr>
</tbody>
</table>

### Description

The high-speed counter in the central unit 07 KT 97 works independently of the user program. Therefore, the counter can react to external signals fast. It can be used in seven different configurable operating modes. The operating mode is configured in the system constant KW 85.02 / %MW3085.2. The planned operating mode is only activated in the initialization phase (power on, cold start, warm start). The same function block COUNTW is used for all operating modes.

Independent of the operating mode the following properties are valid:

- Either the pulses of the counter input signal are counted or the evaluated signals of track A and track B in case of connected synchro transmitters.
- The maximum counting frequency is 50 kHz.
- The counter uses the terminals 2 (E 62.00 / %IX62.0) and 3 (E 62.01 / %IX62.1) as fast inputs and also the output terminal 46 (A 62.00 / %QX62.0) in one operating mode. It is not possible to switch off the counting function of the 07 KT 97 in order to use all inputs and outputs for other purposes.

The counter can count up in all operating modes. It counts beginning with the start value (set value) up to the end value (maximum range from -32768 to +32767 or from 8000 H to 7FFF H). If the value +32767 is exceeded, the counter jumps to -32768. If the counter reaches the planned end value in the function block (END), output CF is set to TRUE (end value reached) and stored. When setting the counter the output CF is reset to FALSE.
In particular operating modes 2 counters can be processed independently of each other. In this case the function block is planned twice. Then the value 0 (for counter 0) has to be applied to the first function block and the value 1 (for counter 1) to the second function block. Normally NO is planned with the value 0.

In some operating modes the counter can also count down. If this is required, the input U/D (Up/Down) of the function block has to be configured with the value TRUE. In this case the counter counts beginning with the start value (set value) up to the end value (maximum range from -32768 to +32767 or from 8000H to 7FFFH). If the value -32768 is exceeded the counter jumps to +32767.

The processing of the counting signals must be enabled. Depending on the operating mode this is performed using a terminal or the input EN (Enable) of the function block (EN is set to TRUE).

The counter can be set to a start value. This start value must be applied to the input STA (start value) of the function block. With a set signal (depending on the operating mode either using a terminal or the input SET of the function block) the value of the word variable is transferred to the counter.

Note: If enabling is active and the set signal is available during several processing cycles, the processing processor always sets the counter at the program end. During the remaining time of the processing cycle the counter counts pulses.

The counter can be set to a start value. This start value must be applied to the input STA (start value) of the function block. With a set signal (depending on the operating mode either using a terminal or the input SET of the function block) the value of the word variable is transferred to the counter.

Note: If enabling is active and the set signal is available during several processing cycles, the processing processor always sets the counter at the program end. During the remaining time of the processing cycle the counter counts pulses.

If the counter reaches the planned end value END, the binary output CF is set to TRUE and the value is stored. When setting the counter the output CF is reset to FALSE.

The counter current value (actual value) can be retrieved at any time using the output Istwert of the function block.

The following section describes which operating modes can be configured, how they can be planned and how they differ from each other. The system constant KW 85,02 / %MW3085.2 configures the counter operating mode.

A hex value of the low byte (bit 0 to bit 7) of KW 85,02 / %MW3085.2 has the following meaning:

00 H = No counter (default setting)
01 H = Mode 1, one count-up counter
02 H = Mode 2, one count-up counter with release input
03 H = Mode 3, two count-up/count-down counters
04 H = Mode 4, two count-up/count-down counters, the second counter counts the pulses on the falling edge
05 H = Mode 5, one count-up/count-down counter with dynamic set input, setting at rising edge
06 H = Mode 6, one count-up/count-down counter with dynamic set input, setting at falling edge
07 H = Mode 7, one count-up/count-down counter with directional discriminator

The high byte is configured with 00 H.

CAUTION: The planned operating mode is only activated in the initialization phase (power on, cold start, warm start).
- **Operating mode 0: No counter**
  This operating mode must be selected if the internal counter is not required. All digital inputs and outputs can then be used for other purposes.

- **Operating mode 1: 1 count-up counter**
  The following terminals are reserved for the counter and cannot be used for other purposes:
  - Terminal 2 (E 62,00): Counting input
  - Terminal 46 (A 62,00): Output "End value reached". Enabling of the counting input and the output "End value reached" is performed via the function block by setting EN to TRUE.

- **Operating mode 2: 1 count-up counter with release input via terminal**
  The following three terminals are reserved for the counter and cannot be used for other purposes:
  - Terminal 2 (E 62,00): Counting input
  - Terminal 3 (A 62,01): Release input
  - Terminal 46 (A 62,00): Output "End value reached". The release input enables the counting input as well as the output "End value reached". The counter is only enabled if the release input is set to TRUE, i.e. input EN = TRUE.

- **Operating mode 3: 2 up-down counters**
  In this operating mode two counters exist independently of one another. The message "End value reached" can only be inquired via the function blocks, not via the terminals. The following two terminals are reserved for the counter and cannot be used for other purposes:
  - Terminal 2 (E 62,00): Counting input counter 0
  - Terminal 3 (A 62,01): Counting input counter 1
  The function block COUNTW is required twice. Note:
  Changing the counting direction (via U/D) is not planned during running operation. If U/D is switched over, at the same time the counter is set to the start value and after this counting is continued in the new counting direction.

- **Operating mode 4: 2 counters (1 counting input inverted)**
  This operating mode corresponds completely to the operating mode 3 with the only exception that the second counting input (counter 1) is inverted. It counts the edges from TRUE to FALSE at terminal 3 (E 62,01 / %IX62.1).

- **Operating mode 5: 1 up-down counter with dynamic set input via terminal**
  In this operating mode one up-down counter with dynamic set input is available. Dynamic means that setting has only an effect on the signal edge and not during application of the signal. The following two terminals are reserved for the counter and cannot be used for other purposes:
  - Terminal 2 (E 62,00 / %IX62.0): Counting input
  - Terminal 3 (E 62,01 / %IX62.1): Dynamic set input
  The dynamic set input at terminal 3 acts on the FALSE/TRUE edge. The message "End value reached" can only be inquired via the function block, not via a terminal.
  Note:
  Changing the counting direction (via U/D) is not planned during running operation. If U/D is switched over, at the same time the counter is set to the start value and after this counting is continued in the new counting direction.

- **Operating mode 6: 1 count-up counter with dynamic set input via terminal**
  This operating mode corresponds completely to the operating mode 5 with the only exception that the dynamic set input acts on the TRUE/FALSE edge.

- **Operating mode 7: Counter with directional discriminator**
  In this operating mode the synchro transmitters can be connected using two counting pulses with an offset of 90 degrees compared to each other. Depending on the position of the pulses the counter counts up or down. A pulse multiplication (x2 or x4) does not occur. For this procedure synchro transmitters with a signal of 24 V can be used. Signals of 5 V synchro transmitters must be increased. A zero track of the synchro transmitter is not processed. The message "End value reached" can only be inquired via the function block, not via a terminal. The following two terminals are reserved for the counter and cannot be used for other purposes:
  - Terminal 2 (E62,00 / %IX62.0): Track A of the synchro transmitter
  - Terminal 3 (E 62,01 / %IX62.1): Track B of the synchro transmitter
Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>COUNTW1(NO := CW_NO, UD := CW_UD, EN := CW_EN, SET := CW_SET, STA := CW_STA, END := CW_END)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>COUNTW1.STAT</td>
</tr>
<tr>
<td>ST</td>
<td>CW_STAT</td>
</tr>
<tr>
<td>LD</td>
<td>COUNTW1.CF</td>
</tr>
<tr>
<td>ST</td>
<td>CW_CF</td>
</tr>
<tr>
<td>LD</td>
<td>COUNTW1.istwert</td>
</tr>
<tr>
<td>ST</td>
<td>CW_ISTWERT</td>
</tr>
<tr>
<td>LD</td>
<td>COUNTW1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>CW_ERR</td>
</tr>
</tbody>
</table>

Function call in ST

COUNTW1(NO := CW_NO, UD := CW_UD, EN := CW_EN, SET := CW_SET, STA := CW_STA, END := CW_END);

CW_STAT := COUNTW1.STAT;
CW_CF := COUNTW1.CF;
CW_ISTWERT := COUNTW1.istwert;
CW_ERR := COUNTW1.ERR;

Note: In IL, the function call has to be performed in one line.
DIVISION DOUBLE WORD

The value of the operand at input E1 is divided by the value of the operand at input E2 and the result is assigned to the operand at output A. The remainder is assigned to the operand at the output REST. If a remainder is produced, the result will always be rounded down. If the result lies outside of the permissible number range, it will be limited to the maximum or minimum value of the number range. If a limiting has been performed, a TRUE signal is assigned to the binary operand at output Q and the value 0 is assigned to output REST. If no limiting occurred, a FALSE signal is assigned to the binary operand at output Q.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DIVD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Dividend</td>
</tr>
<tr>
<td>E2</td>
<td>DINT</td>
<td>Divisor</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Result (quotient)</td>
</tr>
<tr>
<td>REST</td>
<td>DINT</td>
<td>Remainder</td>
</tr>
<tr>
<td>Q</td>
<td>BOOL</td>
<td>Result limited</td>
</tr>
</tbody>
</table>

Description

As the remainder is available at the output REST, the user can compare this to the divisor and can round the result at output A according to his own requirements.

Example:
Remainder > divisor/2 → round up the result at output A.

Division by »zero«

If the divisor has the value »zero«, the positive or negative limit of the number range is assigned to output A.

For the division by »zero« the following applies:
A = -2147483647 (8000 0001H) if dividend is negative
A = +2147483647 (7FFF FFFFH) if dividend is positive

REST = 0    Output for the remainder

Q = TRUE     Output to signalize that the value at output A has been limited

Invalid result value

If the invalid result value 8000 0000H is the result of the division, this will be corrected to the permissible limit 8000 0001H (-2 147 483 647), the binary output Q will be set to the value TRUE and the output REST will be set to the value 0.
Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>DIVD1(E1 := DIVD_E1, E2 := DIVD_E2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>DIVD1.REST</td>
</tr>
<tr>
<td>ST</td>
<td>DIVD1.REST</td>
</tr>
<tr>
<td>LD</td>
<td>DIVD1.Q</td>
</tr>
<tr>
<td>ST</td>
<td>DIVD1.Q</td>
</tr>
<tr>
<td>LD</td>
<td>DIVD1.A</td>
</tr>
<tr>
<td>ST</td>
<td>DIVD1.A</td>
</tr>
</tbody>
</table>

Function call in ST

DIVD1(E1 := DIVD_E1, E2 := DIVD_E2);
DIVD_REST := DIVD1.REST;
DIVD_Q := DIVD1.Q;
DIVD_A := DIVD1.A;
The value of the operand at input E1 is divided by the value of the operand at input E2 and the result is assigned to the operand at output A.

The result of the division is always an integer value. Rounding (i.e. considering the digits after the decimal point) does not take place.

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>E1</th>
<th>INT</th>
<th>Dividend</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2</td>
<td>INT</td>
<td>Divisor</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Result (quotient)</td>
</tr>
</tbody>
</table>

**Description**

The value of the operand at input E1 is divided by the value of the operand at input E2 and the result is assigned to the operand at output A.

The result of the division is always an integer value. Rounding (i.e. considering the digits after the decimal point) does not take place.

The result is limited to the maximum value 32767 and the minimum value -32767.

The inputs and the output can neither be duplicated nor negated.

**Division by »zero«**

If the divisor has the value »zero«, the positive or negative limit of the number range is assigned to output A.

For the division by »zero« the following applies:

- A = -32767 (8001H) if the dividend is negative
- A = +32767 (7FFFH) if the dividend is positive

**Function call in IL**

LD DIVW_E1
DIVW DIVW_E2
ST DIVW_A

**Function call in ST**

DIVW_A:=DIVW(DIVW_E1, DIVW_E2);
**DEMUTIPLEXER, WORD**

This block connects one of the inputs E0 ... En-1 to output A depending on input INDEX.

The DMUX number indicates the maximum number of input values. The following demultiplexers are available:

- **DMUX8** Demultiplexer word with a maximum of 8 input values
- **DMUX16** Demultiplexer word with a maximum of 16 input values
- **DMUX32** Demultiplexer word with a maximum of 32 input values
- **DMUX64** Demultiplexer word with a maximum of 64 input values
- **DMUX256** Demultiplexer word with a maximum of 256 input values

**Block type**

Function block without historical values

**Parameters**

| Instance | DMUX(8..256) | Instance name
| INDEX | INT | Index input
| N | INT | Number of word inputs E0 ... En-1
| E0..En-1 | INT | Word input
| INOK | BOOL | Range monitoring of the index input
| A | INT | Word output to which one of the inputs E0...En-1 is switched through

**Description**

This block connects one of the inputs E0 ... En-1 to output A depending on input INDEX.

The value at input INDEX is monitored for validity. The output A is set to 0 if the word input INDEX is not within the valid range.

**Relationship between E0...En-1, INDEX and A:**

The input INDEX is used to define which of the inputs E0 ... En-1 is connected to output A.

The following applies:

- INDEX = 1: E0 → A
- INDEX = 2: E1 → A
- INDEX = 3: E2 → A
- ...
- INDEX = n: En-1 → A

where: 1 ≤ INDEX ≤ n (number of inputs E0..En-1)

If input INDEX = 0 the output is set to 0.

**INDEX**

Index input for selection of one of the inputs E0 ... En-1.

The following applies:

1 ≤ INDEX ≤ n (number of inputs E0 ... En-1)

**Remark:**

INDEX = 0 → Initialization of output A with 0.

**n**

Number n of word inputs E0 ... En-1

The following applies:

1 ≤ n ≤ DMUX number (8..256)

**E0..En-1**

Input E0 ... En-1 capable of duplication

One of the inputs E0 ... En-1 is connected to output A.
The output INOK indicates whether or not the input INDEX is within the valid range. The outputs INOK and A are set to 0 if the word input INDEX is not within the valid range.

The following applies:

\[ \text{INOK} = \text{TRUE} \rightarrow \text{INDEX in the valid range} \]
\[ \text{INOK} = \text{FALSE} \rightarrow \text{INDEX in the invalid range} \rightarrow A = 0 \]

Valid range for the index: \( 1 \leq \text{INDEX} \leq n \)

### Function call in IL

```
CAL DMUX81(INDEX := DM_INDEX, n := 3, E0 := DM_E0, E1 := DM_E1, E2 := DM_E2)
LD DMUX81.A
ST DM_A
LD DMUX81.INOK
ST DM_INOK
```

Note: In IL, the function call has to be performed in one line.

### Function call in ST

```
DMUX81(INDEX := DM_INDEX, n := 3, E0 := DM_E0, E1 := DM_E1, E2 := DM_E2);
DM_A:=DMUX81.A;
DM_INOK:=DMUX81.INOK;
```
DEMUTIPLEXER, DOUBLE WORD

This block connects one of the inputs E0 ... En-1 to output A depending on input INDEX. The DMUXD number indicates the maximum number of input values. The following demultiplexers are available:

DMUXD8  Demultiplexer double word with a maximum of 8 input values
DMUXD16 Demultiplexer double word with a maximum of 16 input values
DMUXD32 Demultiplexer double word with a maximum of 32 input values
DMUXD64 Demultiplexer double word with a maximum of 64 input values
DMUXD256 Demultiplexer double word with a maximum of 256 input values

Block type

Function block without historical values

PARAMETERS

Instance  DMUXD(8..256)  Instance name
INDEX   INT  Index input
N     INT  Number n of double word inputs E0 ... En-1
E0..En-1 DINT  Double word input
INOK  BOOL  Range monitoring of the index input
A  DINT  Double word output to which one of the inputs E0...En-1 is switched through

Description

This block connects one of the inputs E0 ... En-1 to output A depending on input INDEX. The value at input INDEX is monitored for validity. The output A is set to 0 if the double word input INDEX is not within the valid range.

Relationship between E0...En-1, INDEX and A:

The input INDEX is used to define which of the inputs E0 ... En-1 is connected to output A.

The following applies:

INDEX = 1: E0 → A
INDEX = 2: E1 → A
INDEX = 3: E2 → A
... ...
INDEX = n: En-1 → A

where: 1 ≤ INDEX ≤ n (number of inputs E0..En-1)

If input INDEX = 0 the output is set to 0.

INDEX   INT

Index input for selection of one of the inputs E0 ... En-1. The following applies:

1 ≤ INDEX ≤ n (number of inputs E0 ... En-1)

Remark:
INDEX = 0 → Initialization of output A with 0.

n  INT

Number n of double word inputs E0 ... En-1. The following applies:

1 ≤ n ≤ DMUXD number (8..256)

E0..En-1  DINT

Input E0 ... En-1 capable of duplication. One of the inputs E0 ... En-1 is connected to output A.
DEMULPLEXER, DOUBLE WORD

DMUXD(8..256)

INOK BOOL
Range monitoring for input INDEX
The output INOK indicates whether or not the input INDEX is within the valid range. The outputs INOK and A are set to 0 if the word input INDEX is not within the valid range.

The following applies:
INOK = TRUE
→ INDEX in the valid range
INOK = FALSE
→ INDEX in the invalid range → A = 0
Valid range for the index: 1 ≤ INDEX ≤ n

A DINT
Output to which one of the inputs E0...En-1 is switched through. The input INDEX is used to select one of the inputs E0...En-1 and to allocate its value to output A. The output A is set to 0 if the word input INDEX is not within the valid range.

Function call in IL
CAL DMUXD81(INDEX := DMD_INDEX, n := 3,
E0 := DMD_E0, E1 := DMD_E1,
E2 := DMD_E2)
LD DMUXD81.A
ST DMD_A
LD DMUXD81.INOK
ST DMD_INOK

Function call in ST
DMUXD81(INDEX := DMD_INDEX, n := 3,
E0 := DMD_E0, E1 := DMD_E1, E2 := DMD_E2);
DMD_A:=DMUXD81.A;
DMD_INOK:=DMUXD81.INOK;

Note: In IL, the function call has to be performed in one line.
DIFFERENTIATOR WITH DELAY OF THE 1ST ORDER

The controlled variable X is multiplied by the proportional coefficient KP. The proportional coefficient is specified as a percentage value. If the controlled variable no longer changes, the manipulated variable Y moves towards the value 0 in an exponential function. The time constant T1 specifies the time when the step response has dropped to approximately 37% of its initial value. The value has dropped below 1% after the time of 5 * T1.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>INT</td>
<td>Controlled variable</td>
</tr>
<tr>
<td>KP</td>
<td>INT</td>
<td>Proportional coefficient, specified as a percentage value</td>
</tr>
<tr>
<td>T1_TZ</td>
<td>INT</td>
<td>Time constant normalized to cycle time</td>
</tr>
<tr>
<td>Y</td>
<td>INT</td>
<td>Manipulated variable</td>
</tr>
</tbody>
</table>

Description (see next page)
The controlled variable $X$ is multiplied by the proportional coefficient $KP$. The proportional coefficient is specified as a percentage value. If the controlled variable no longer changes, the manipulated variable $Y$ moves towards the value 0 in an exponential function. The time constant $T1$ specifies the time when the step response has dropped to approximately 37% of its initial value. The value has dropped below 1% after the time of $5 \times T1$.

The inputs and the output can neither be duplicated nor negated.

Transfer function:

\[ F(s) = \frac{KP \times s \times T1}{s \times T1 + 1} \]

Transfer function:

Function call in IL

\[
\text{CAL DT11(X := DT1_X, KP := DT1_KP, } \\
T1_TZ := DT1_T1TZ) \\
\text{LD DT11.Y} \\
\text{ST DT1_Y}
\]

Note: In IL, the function call has to be performed in one line.

Function call in ST

\[
\text{DT11(X := DT1_X, KP := DT1_KP, } \\
T1_TZ := DT1_T1TZ); \\
\text{DT1_Y := DT11.Y;}
\]
**DUAL TO BCD CONVERSION, WORD**

The binary number at input E is converted to a BCD coded number and assigned to the operand at output A.

The binary number is represented in 16 bits and must lie within the range $0 \leq E < 270FH$ (corresponding to BCD 9999). The BCD number is limited to 9999 if it lies outside of this range. The BCD number is stored in a 16 bit word.

**Block type**
- Function

**Parameters**

<table>
<thead>
<tr>
<th>E</th>
<th>INT</th>
<th>Binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>INT</td>
<td>BCD coded number</td>
</tr>
</tbody>
</table>

**Description**

The binary number at input E is converted to a BCD coded number and assigned to the operand at output A.

The binary number is represented in 16 bits and must lie within the range $0 < \text{DUAL} < 270FH$ (corresponding to BCD 9999). The BCD number is limited to 9999 if it lies outside of this range. The BCD number is stored in a 16 bit word.

**Definition:**

The input and the output can neither be duplicated nor negated.

**Definition:**

The significance of the digits in a hexadecimal number and a BCD coded number is defined as follows:

<table>
<thead>
<tr>
<th>HEXDEC NUMBER</th>
<th>BCD NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 11 7 3 0</td>
<td>15 11 7 3 0</td>
</tr>
<tr>
<td>Z4 Z3 Z2 Z1</td>
<td>Z4 Z3 Z2 Z1</td>
</tr>
</tbody>
</table>

Numerical value:

- $Z1 \times 1$
- $Z2 \times 16$
- $Z3 \times 256$
- $Z4 \times 4096$
- $0 \leq Z_i \leq F$

Numerical value:

- $Z1 \times 1$
- $Z2 \times 10$
- $Z3 \times 100$
- $Z4 \times 1000$
- $0 \leq Z_i \leq 9$
Example

<table>
<thead>
<tr>
<th>HEXDEC NUMBER</th>
<th>BCD NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 11 7 3 0</td>
<td>15 11 7 3 0</td>
</tr>
<tr>
<td>0 4 D 2</td>
<td>1 2 3 4</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
Z_1 &= 2 \times 1 = 2 \\
Z_2 &= 13 \times 16 = 208 \\
Z_3 &= 4 \times 256 = 1024 \\
Z_4 &= 0 \times 4096 = 0 \\
\end{align*}
\]

Conversion of a negative binary number to a BCD number

A negative binary number with an amount less than 270FH can be converted to a BCD number, whereby the value and the sign of the BCD number are each stored in one flag.

Function call in IL

LD DUAL_E
DUALBCD
ST DUAL_A

Function call in ST

DUAL_A := DUALBCD(DUAL_E);

Converting a binary number with an amount higher than 270FH

Binary numbers with an amount higher than 270FH are first of all converted to a double word (function block WDW). They are then converted from BINARY to BCD by means of the DUALBCDD function block.
The binary number at input E is converted to a BCD coded number and assigned to the operand at output A.

The binary number is represented in 32 bits and must lie within the range $0 \leq E \leq 5F5E0FFH$ (corresponds to BCD $99999999$). The BCD number is limited to $99999999$ if it lies outside of this range. The BCD number is stored in a 32 bit word.

**Block type**

**Function**

**Parameters**

<table>
<thead>
<tr>
<th>E</th>
<th>DINT</th>
<th>Binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>DINT</td>
<td>BCD coded number</td>
</tr>
</tbody>
</table>

**Description**

The binary number at input E is converted to a BCD coded number and assigned to the operand at output A.

The binary number is represented in 32 bits and must lie within the range $0 \leq E \leq 5F5E0FFH$ (corresponds to BCD $99999999$). The BCD number is limited to $99999999$ if it lies outside of this range. The BCD number is stored in a 32 bit word.

**Definition:**

The input and the output can neither be duplicated nor negated.

The significance of the digits in a hexadecimal number and a BCD coded number is defined as follows:

**HEXDEC NUMBER**

<table>
<thead>
<tr>
<th>Z8</th>
<th>Z7</th>
<th>Z6</th>
<th>Z5</th>
<th>Z4</th>
<th>Z3</th>
<th>Z2</th>
<th>Z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>15</td>
<td>0</td>
<td>BIT</td>
<td>31</td>
<td>15</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Numerical value:

- $Z1 \times 1$
- $Z2 \times 16$
- $Z3 \times 256$
- $Z4 \times 4096$
- $Z5 \times 65536$
- $Z6 \times 1048576$
- $Z7 \times 16777216$
- $Z8 \times 268435456$
- $0 \leq Z1 \leq F$

**BCD NUMBER**

<table>
<thead>
<tr>
<th>Z8</th>
<th>Z7</th>
<th>Z6</th>
<th>Z5</th>
<th>Z4</th>
<th>Z3</th>
<th>Z2</th>
<th>Z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>15</td>
<td>0</td>
<td>BIT</td>
<td>31</td>
<td>15</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Numerical value:

- $Z1 \times 1$
- $Z2 \times 10$
- $Z3 \times 100$
- $Z4 \times 1000$
- $Z5 \times 10000$
- $Z6 \times 100000$
- $Z7 \times 1000000$
- $Z8 \times 10000000$
- $0 \leq Z1 \leq 9$
Example

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
<th>6</th>
<th>1</th>
<th>4</th>
<th>E</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>C</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

BCD NUMBER

<table>
<thead>
<tr>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
<th>Z4</th>
<th>Z5</th>
<th>Z6</th>
<th>Z7</th>
<th>Z8</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 * 1</td>
<td>6 * 16</td>
<td>1 * 256</td>
<td>6 * 4095</td>
<td>12 * 65536</td>
<td>11 * 1048576</td>
<td>0 * 16777216</td>
<td>0 * 268435456</td>
</tr>
<tr>
<td>= 14</td>
<td>= 64</td>
<td>= 256</td>
<td>= 24576</td>
<td>= 786432</td>
<td>= 11534336</td>
<td>= 0</td>
<td>= 0</td>
</tr>
</tbody>
</table>

HEXDEC NUMBER

<table>
<thead>
<tr>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
<th>Z4</th>
<th>Z5</th>
<th>Z6</th>
<th>Z7</th>
<th>Z8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 * 1</td>
<td>7 * 10</td>
<td>6 * 100</td>
<td>5 * 1000</td>
<td>4 * 10000</td>
<td>3 * 100000</td>
<td>2 * 1000000</td>
<td>1 * 10000000</td>
</tr>
<tr>
<td>= 8</td>
<td>= 70</td>
<td>= 600</td>
<td>= 5000</td>
<td>= 40000</td>
<td>= 300000</td>
<td>= 2000000</td>
<td>= 10000000</td>
</tr>
</tbody>
</table>

Conversion of a negative binary number to a BCD number:

see function block DUALBCD

Function call in IL

LD DUALD_E
DUALBCD
ST DUALD_A

Function call in ST

DUALD_A:= DUALBCD(DUALD_E);
DOUBLE WORD TO TWO WORDS CONVERSION

This block allows to split up a double word into two individual words.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DW2W</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW</td>
<td>DINT</td>
<td>Input double word</td>
</tr>
<tr>
<td>MSW</td>
<td>INT</td>
<td>Output higher significant word</td>
</tr>
<tr>
<td>LSW</td>
<td>INT</td>
<td>Output lower significant word</td>
</tr>
</tbody>
</table>

Description

With this block it is possible to split up a double word into two 16 bit words. For example, this is necessary to transfer double words.

The flag words must be specified directly at the outputs MSW and LSW. Data errors can occur, if the assignment is performed via intermediate flags (correction of the value 8000H).

Caution:
For data records sent by MasterPiece systems, the higher significant word is located on the lower address.

Function call in IL

<table>
<thead>
<tr>
<th>Function call in IL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
</tr>
<tr>
<td>LD</td>
</tr>
<tr>
<td>ST</td>
</tr>
<tr>
<td>LD</td>
</tr>
<tr>
<td>ST</td>
</tr>
</tbody>
</table>

Function call in ST

<table>
<thead>
<tr>
<th>Function call in ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW2W1(DW := DW_DW);</td>
</tr>
<tr>
<td>DW_LSW:=DW2W1.LSW;</td>
</tr>
<tr>
<td>DW_MSW:=DW2W1.MSW;</td>
</tr>
</tbody>
</table>
WRITE DOUBLE WORD IN THE EVENT OF VALUE CHANGE

If the value of the operand at input E1 changes in comparison with the value during previous processing of the block, the value of the operand at input E1 is written to the specified physical address.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DWAES</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Input for the operand to be read</td>
</tr>
<tr>
<td>ADRESSE</td>
<td>DWORD</td>
<td>Address of the memory location to which the value of E1 must be written in the event of a change.</td>
</tr>
</tbody>
</table>

Description

If the value of the operand at input E1 changes in comparison with the value during previous processing of the block, the value of the operand at input E1 is written to the specified physical address.

The physical address consists of a segment and an offset.

The inputs can neither be duplicated nor inverted.

Function call in IL

```
LD  MD
ADR
ST  DWAES1.ADRESSE
CAL  DWAES1(E1 := DWAES_E1)
```

Function call in ST

```
DWAES1.ADRESSE:=ADR(MD);
DWAES1(E1 := DWAES_E1);
```
READ DOUBLE WORD WITH ENABLING    DWOL

If a TRUE signal is present at input FREI, the value of the specified physical address is read and assigned to the operand at output A.

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling the block</td>
</tr>
<tr>
<td>ADRESSE</td>
<td>DWORD</td>
<td>Address of the memory location whose double word value is to be read.</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Output to which the value read is assigned.</td>
</tr>
</tbody>
</table>

**Description**

If a TRUE signal is present at input FREI, the value of the specified physical address is read and assigned to the operand at output A.

Reading is not performed if there is a FALSE signal at input FREI.

The physical address is specified at input ADRESSE.

**Function call in IL**

LD DWOL_FR
DWOL DWOL_ADR
ST DWOL_A

**Function call in ST**

DWOL_A:=DWOL(DWOL_FR,DWOL_ADR);
### WRITE DOUBLE WORD WITH ENABLING

If there is a TRUE signal at input FREI, the value of the operand at input E1 is read and then written to the specified physical address.

<table>
<thead>
<tr>
<th>Block type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
</tr>
<tr>
<td>E1</td>
<td>DINT</td>
</tr>
<tr>
<td>ADRESSE</td>
<td>DWORD</td>
</tr>
</tbody>
</table>

#### Description

If there is a TRUE signal at input FREI, the value of the operand at input E1 is read and then written to the specified physical address.

Writing is not performed if there is a FALSE signal at input FREI.

The physical address is specified at input ADRESSE.

The inputs can neither be duplicated nor inverted.

**FREI**

BOOL

Processing of the block is enabled or disabled with the operand at input FREI. The following applies:

- FREI = FALSE → Processing disabled
- FREI = TRUE → Processing enabled

**E1**

DINT

The operand at input E1 is read and its value is written to the address defined by the input ADR.

**ADRESSE**

DWORD

The 32 bit address to be read is specified at input ADRESSE.

### Function call in IL

```
LD   DWOS_FREI
DWOS DWOS_E1,DWOS_ADR
```

### Function call in ST

```
DWOS(DWOS_FR,DWOS_E1,DWOS_ADR);
```
DOUBLE WORD RECODER

This block compares the value of the operand at input E to the reference values of the operands at the inputs EC0 ... ECn-1. If input E agrees with at least one of the reference values EC, output E_EC is set to TRUE. Output A receives the value of the output code AC which is allocated to the found reference value EC.

The DWUMC number indicates the maximum number of reference values. The following double word recoders are available:

- DWUMC8: Double word recoder with a maximum of 8 reference values
- DWUMC16: Double word recoder with a maximum of 16 reference values
- DWUMC32: Double word recoder with a maximum of 32 reference values
- DWUMC64: Double word recoder with a maximum of 64 reference values
- DWUMC256: Double word recoder with a maximum of 256 reference values

**Block type**

Function block without historical values

**Parameter**

<table>
<thead>
<tr>
<th>Instanz</th>
<th>DWUMC(8..256)</th>
<th>Instanzname</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>DINT</td>
<td>Eingang</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Anzahl der Referenzwerte</td>
</tr>
<tr>
<td>EC0..ECn-1</td>
<td>DINT</td>
<td>Referenzwert; Eingang ist doppelbar</td>
</tr>
<tr>
<td>AC0..ACn-1</td>
<td>DINT</td>
<td>Ausgangscode; Eingang ist doppelbar</td>
</tr>
<tr>
<td>E_EC</td>
<td>BOOL</td>
<td>Koinzidenzanzeige</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Ausgabe des Werls des Ausgangscodes</td>
</tr>
</tbody>
</table>

**Description**

This block compares the value of the operand at input E to the reference values of the operands at the inputs EC0 ... ECn-1. If input E agrees with at least one of the reference values EC, output E_EC is set to TRUE. Output A receives the value of the output code AC which is allocated to the found reference value EC.

An operand for the output code ACi is allocated to each reference value at the inputs ECi. The allocation of EC to AC is recognizable by the index i. The index begins with 0 and is generated automatically in the event of duplication.

The number of reference values EC must be specified at input n.

The inputs and outputs cannot be negated/inverted.
AC0 … ACn-1 DINT

When the input EC0 is duplicated, input AC0 is also duplicated automatically. The output codes are specified at the inputs AC0 … ACn-1. The output code ACi is output at output A if input E agrees with one of the reference values ECi.

Assignment between reference values and output codes:

EC0 ↔ AC0
EC1 ↔ AC1
...
ECn-1 ↔ ACn-1

Function call in IL

CAL DWUMC81(E := DWU_E, n := 3,
EC0 := DWU_EC0, AC0 := DWU_AC0,
EC1 := DWU_EC1, AC1 := DWU_AC1,
EC2 := DWU_EC2, AC2 := DWU_AC2,)
LD DWUMC81.A
ST DWU_A
LD DWUMC81.E_EC
ST DWU_EEC

Note: In IL, the function call has to be performed in one line.

E_EC BOOL

Agreement between the operand value of input E and one of the reference values is indicated at output E=EC.

The following applies:

E=EC = FALSE → No agreement
E=EC = TRUE → Agreement

A DINT

The output code ACi is output at output A if input E agrees with one of the reference values ECi.

The following applies:

A = 0 → No agreement
A = ACi → Agreement

Function call in ST

DWUMC81(E := DWU_E, n := 3,
EC0 := DWU_EC0, AC0 := DWU_AC0,
EC1 := DWU_EC1, AC1 := DWU_AC1,
EC2 := DWU_EC2, AC2 := DWU_AC2,);
DWU_A:=DWUMC81.A
DWU_EEC:=DWUMC81.E_EC
DOUBLE WORD TO WORD CONVERSION

The value of the double word operand at input \( E \) is converted to a word variable and the result is assigned to the word operand at output \( A1 \).

**Block type**

Function block without historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>DWW</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>DINT</td>
<td>Double word variable to be converted</td>
</tr>
<tr>
<td>A1</td>
<td>INT</td>
<td>Result of conversion, word variable</td>
</tr>
<tr>
<td>A2</td>
<td>BOOL</td>
<td>Result limited</td>
</tr>
</tbody>
</table>

**Description**

The value of the double word operand at input \( E \) is converted to a word variable and the result is assigned to the word operand at output \( A1 \).

The result is limited to the maximum or minimum number range.

max. number range: +32767 (7FFFH)

min. number range: -32767 (8001H)

If limiting occurred, a TRUE signal is assigned to the binary operand at output \( A2 \). If no limiting occurred, a FALSE signal is assigned to the binary operand at output \( A2 \).

The input and the outputs can neither be duplicated nor negated.

**Function call in IL**

```
CAL DWW1(E := DWW_E)
LD DWW1.A2
ST DWW_A2
LD DWW1.A1
ST DWW_A1
```

**Function call in ST**

```
DWW1(E := DWW_E);
DWW_A2:=DWW1.A2;
DWW_A1:=DWW1.A1;
```
ON DELAY

The FALSE/TRUE edge of input $E$ is delayed by the time $T$ and is output as a FALSE/TRUE edge at output $A$.

If input $E$ returns to the FALSE level before the time $T$ is expired, output $A$ remains in the FALSE level.

**Block type**

Function block with historical

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>ESV</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>BOOL</td>
<td>Input signal</td>
</tr>
<tr>
<td>$T$</td>
<td>TIMER</td>
<td>Delay time</td>
</tr>
<tr>
<td>$A$</td>
<td>BOOL</td>
<td>Delayed signal</td>
</tr>
</tbody>
</table>

**Description**

The FALSE/TRUE edge of input $E$ (input $TH0$ in the figure) is delayed by the time $T$ and is output as a FALSE/TRUE edge at output $A$ ($Q$ in the figure).

If input $E$ returns to the FALSE level before the time $T$ is expired, output $A$ remains in the FALSE level.

Maximum time offset at the output: $< 1$ cycle time

Reasonable range for $T$: $> 1$ cycle time

The inputs and the output can neither be duplicated nor inverted.

**General behavior**

Started timers are processed by the PLC's operating system and are therefore completely independent of PLC program processing. The operating system does not issue a corresponding message to the corresponding timer block in the PLC program until the timer has elapsed.

**Function call in IL**

```
CAL ESV1(E := ESV_E, T := T#200ms)
LD ESV1.A
ST ESV_A
```

**Function call in ST**

```
ESV1(E := ESV_E, T := T#200ms);
ESV_A:=ESV1.A;
```
The function block realizes a stack memory for binary data from which the data written in first are again read out first (first in/first out).

### Parameters

- **Instance FIFOB Instance name**
- **LADE BOOL Load FIFO**
- **LESE BOOL Read FIFO**
- **RESET BOOL Reset FIFO, FALSE -> TRUE edge**
- **IE BOOL Input of data in the FIFO**
- **ANZ INT Number of memory locations (bytes)**
- **ANF DWORD Start address of the FIFO in the flag area**
- **A BOOL Output of data from the FIFO**
- **FST INT Filling level of the FIFO**
- **LEER BOOL FIFO empty**
- **VOLL BOOL FIFO full**

### Description

The function block realizes a stack memory for binary data from which the data written in first are again read out first (first in/first out).

The inputs and outputs can neither be duplicated nor negated/inverted.

**LADE**

The value present at input E is transferred to the next free position of the FIFO by means of a TRUE signal at input LADE. If the FIFO is "full" and a "load" signal is present, the new value will not be read in. A new value can only be read if a value has been read out first. It is then transferred to the first position of the FIFO.

**LESE**

A TRUE signal at input LESE results in the output of the current FIFO value through output A. If the FIFO is "empty", a pending read order is ignored and the value FALSE is applied at output A. Output A is always set to FALSE if there is no read order.

**LADE and LESE**

If load and read orders are present simultaneously and the FIFO is empty, the value to be loaded is forwarded directly from input E to output A. If the FIFO is not empty, the current FIFO value is output and the new load information is transferred to it. The FIFO does not change its filling level in this operating mode. The value output through output FST remains constant. If the FIFO is empty, output LEER remains TRUE and output FST is set to 0.

**RESET**

A FALSE -> TRUE edge at input RESET results in the reset of the block. Therefore, values read in before are no longer available.

Output LEER is set to TRUE and the outputs FST and A are set to 0 (FALSE).
RESET and LADE
If RESET and LADE signals are present simultaneously the reset is performed first and then directly afterwards the load operation. Therefore, the FIFO is reset and the new value to be read in is then immediately stored in the FIFO as the first value.

RESET and LADE and LESE
The value at input E is forwarded directly to output A. Output LEER is set permanently to TRUE and output FST is set permanently to 0.

E
The value to be transferred into the FIFO is specified at input E.

ANZ
The number of required memory locations (bytes) of the FIFO is specified at input ANZ. The number of memory locations corresponds to the number of binary values to be stored.

ANF
The FIFO memory start address is specified as a binary flag at input ANF. The block ADR provides the corresponding binary flag address. The FIFO begins with the specified flag.

A
When reading the FIFO, the current value is available at output A. If no read order is available, the value FALSE is output.

FST
Output FST indicates the current filling level of the FIFO at any time. The filling level is the number of binary values stored in the FIFO.

LEER
The output LEER indicates whether or not the FIFO is empty.

LEER = FALSE → FIFO is not empty
LEER = TRUE → FIFO is empty

VOLL
The output VOLL indicates whether or not the FIFO is full.

VOLL = FALSE → FIFO is not full
VOLL = TRUE → FIFO is full

No further values can be read in if the FIFO is full. A new value can only be read in after a value has been read out. The new value is positioned to the beginning of the FIFO.

Function call in IL
LD FIFO_ANF
ADR
ST FIFOB1.ANF
CAL FIFOB1(LADE := FIFO_LADE, LESE := FIFO_LESE, RESET := FIFO_RES, E := FIFO_E, ANZ := FIFO_ANZ)
LD FIFOB1.FST
ST FIFO_FST
LD FIFOB1.LEER
ST FIFO_LEER
LD FIFOB1.VOLL
ST FIFO_VOLL
LD FIFOB1.A
ST FIFO_A

Note: In IL, the function call has to be performed in one line.

Function call in ST
FIFOB1.ANF := ADR(FIFO_ANF);
FIFOB1(LADE := FIFO_LADE, LESE := FIFO_LESE, RESET := FIFO_RES, E := FIFO_E, ANZ := FIFO_ANZ);
FIFO_FST:=FIFOB1.FST;
FIFO_LEER:=FIFOB1.LEER;
FIFO_VOLL:=FIFOB1.VOLL;
FIFO_A:=FIFOB1.A;
STACK MEMORY FIRST IN FIRST OUT, WORD

The function block realizes a stack memory for word data from which the data written in first are again read out first (first in/first out).

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>FIFOW</td>
<td>Instance name</td>
</tr>
<tr>
<td>LADE</td>
<td>BOOL</td>
<td>Load FIFO</td>
</tr>
<tr>
<td>LESE</td>
<td>BOOL</td>
<td>Read FIFO</td>
</tr>
<tr>
<td>RESET</td>
<td>BOOL</td>
<td>Reset FIFO, FALSE→TRUE edge</td>
</tr>
<tr>
<td>IE</td>
<td>INT</td>
<td>Input of data in the FIFO</td>
</tr>
<tr>
<td>ANZ</td>
<td>INT</td>
<td>Number of memory locations (bytes)</td>
</tr>
<tr>
<td>ANF</td>
<td>INT</td>
<td>Start address of the FIFO in the flag area</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Output of data from the FIFO</td>
</tr>
<tr>
<td>FST</td>
<td>INT</td>
<td>Filling level of the FIFO</td>
</tr>
<tr>
<td>LEER</td>
<td>BOOL</td>
<td>FIFO empty</td>
</tr>
<tr>
<td>VOLL</td>
<td>BOOL</td>
<td>FIFO full</td>
</tr>
</tbody>
</table>

Description

The function block realizes a stack memory for word data from which the data written in first are again read out first (first in/first out).

The inputs and outputs can neither be duplicated nor negated/inverted.

**LADE**

The value present at input E is transferred to the next free position of the FIFO by means of a TRUE signal at input LADE. If the FIFO is "full" and a "load" signal is present, the new value will not be read in. A new value can only be read if a value has been read out first. It is then transferred to the first position of the FIFO.

**LESE**

A TRUE signal at input LESE results in the output of the current FIFO value through output A. If the FIFO is "empty", a pending read order is ignored and the value 0 is applied at output A. Output A is always set to 0 if there is no read order.

**LADE and LESE**

If load and read orders are present simultaneously and the FIFO is empty, the value to be loaded is forwarded directly from input E to output A. If the FIFO is not empty, the current FIFO value is output and the new load information is transferred to it. The FIFO does not change its filling level in this operating mode. The value output through output FST remains constant. If the FIFO is empty, output LEER remains TRUE and output FST is set to 0.

**RESET**

A FALSE → TRUE edge at input RESET results in the reset of the block. Therefore, values read in before are no longer available.

Output LEER is set to TRUE and the outputs FST and A are set to 0 (FALSE).
RESET and LADE
If RESET and LADE signals are present simultaneously the reset is performed first and then directly afterwards the load operation. Therefore, the FIFO is reset and the new value to be read in is then immediately stored in the FIFO as the first value.

RESET and LADE and LESE
The value at input E is forwarded directly to output A. Output LEER is set permanently to TRUE and output FST is set permanently to 0.

E
The value to be transferred into the FIFO is specified at input E.

ANZ
The number of required memory locations (words) of the FIFO is specified at input ANZ. The number of memory locations corresponds to the number of binary values to be stored.

ANF
The FIFO memory start address is specified as a word flag at input ANF. The FIFO begins with the specified flag word.

Function call in IL
LD FIFO_ANF
ADR
ST FIFOW1.ANF
CAL FIFOW1(LADE := FIFO_LADE, LESE := FIFO_LESE, RESET := FIFO_RES, E := FIFO_E, ANZ := FIFO_ANZ)
LD FIFOW1.FST
ST FIFO_FST
LD FIFOW1.LEER
ST FIFO_LEER
LD FIFOW1.VOLL
ST FIFO_VOLL
LD FIFOW1.A
ST FIFO_A

Note: In IL, the function call has to be performed in one line.

A
When reading the FIFO the current value is available at output A. If no read order is available, the value 0 is output.

FST
Output FST indicates the current filling level of the FIFO at any time. The filling level is the number of binary values stored in the FIFO.

LEER
The output LEER indicates whether or not the FIFO is empty.
LEER = FALSE  → FIFO is not empty
LEER = TRUE   → FIFO is empty

VOLL
The output VOLL indicates whether or not the FIFO is full.
VOLL = FALSE  → FIFO is not full
VOLL = TRUE   → FIFO is full

Function call in ST
FIFOW1.ANF := ADR(FIFO_ANF);
FIFOW1( LADE := FIFO_LADE,
         LESE := FIFO_LESE, RESET := FIFO_RES,
         E := FIFO_E, ANZ := FIFO_ANZ );
FIFO_FST :=FIFOW1.FST;
FIFO_LEER :=FIFOW1.LEER;
FIFO_VOLL :=FIFOW1.VOLL;
FIFO_A :=FIFOW1.A;
**FUNCTION GENERATOR**

**FKG(2..256)**

In an x/y coordinate system, a polygon is defined by n coordinate points X0/Y0...Xn-1/Yn-1. For each value at input x, the function block outputs the assigned y value of the polygon at output y.

The FKG number indicates the maximum number of points. The following function generators are available:

- FKG2 Function generator with max. 2 points
- FKG4 Function generator with max. 4 points
- FKG16 Function generator with max. 16 points
- FKG32 Function generator with max. 32 points
- FKG64 Function generator with max. 64 nodes
- FKG256 Function generator with max. 256 points

### Block type

Function block without historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>FKG(2..256)</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>INT</td>
<td>Input for the x value of the polygon</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Number of points</td>
</tr>
<tr>
<td>XC0..XCn-1</td>
<td>INT</td>
<td>Input for x-values of the points</td>
</tr>
<tr>
<td>YC0..YCn-1</td>
<td>INT</td>
<td>Input for y-values of the points</td>
</tr>
<tr>
<td>Y</td>
<td>INT</td>
<td>Output for the y value of the polygon</td>
</tr>
</tbody>
</table>

### Description

The FKG number indicates the maximum number of points. The following function generators are available:

- FKG2 Function generator with max. 2 points
- FKG4 Function generator with max. 4 points
- FKG16 Function generator with max. 16 points
- FKG32 Function generator with max. 32 points
- FKG64 Function generator with max. 64 points
- FKG256 Function generator with max. 256 points

In an x/y coordinate system, a polygon is defined by n coordinate points X0/Y0...Xn-1/Yn-1. For each value at input x, the function block outputs the assigned y value of the polygon at output y.

To the x-coordinate is valid:

\[ X_0 < X_1 < X_2 \ldots < X_{n-1} \]

\[ 2 \leq n \leq \text{FKG number} \]

Example:

For FKG16, the following applies: \( 2 \leq n \leq 16 \)

The block performs a linear interpolation between the points. The resulting polygon represents the connection between the input value x and the output value y.

For the interpolation between two points the following applies:

\[
\frac{(x - X_{i-1}) \ast (Y_i - Y_{i-1})}{X_i - X_{i-1}} + Y_{i-1}
\]

### Notes:

- At the division it is always rounded down, i.e. a remainder with the division is not considered.
- Die difference between two points may amount to max. 32767.
- For the range outside of the nodes the following applies:
  - for \( x < X_0 \), \( y = Y_0 \)
  - for \( x > X_{n-1} \), \( y = Y_{n-1} \)
x  
INT  
The current x coordinate is specified at input x. The block then defines the y coordinate assigned by the polygon.

n  
INT  
The number of nodes which are necessary to define the polygon is specified at input n.

Function call in IL  
CAL FKG21(X := FKG_X, n := 2,  
XC0 := FKG_XC0, YC0 := FKG_YC0,  
XC1 := FKG_XC1, YC1 := FKG_YC1)  
LD FKG21.Y  
ST FKG_Y  

Note: In IL, the function call has to be performed in one line.

Function call in ST  
FKG21(X := FKG_X,  
n := 2,  
XC0 := FKG_XC0,  
YC0 := FKG_YC0,  
XC1 := FKG_XC1,  
YC1 := FKG_YC1);  
FKG_Y:=FKG21.Y;  

y  
INT  
The y coordinate assigned by the polygon of the specified x coordinate is output at output y.

XC0----XCn-1  
INT  
The x coordinates of the n nodes are specified at the inputs XC0 ... XCn-1.

YC0----YCn-1  
INT  
The y coordinates of the n nodes are specified at the inputs YC0 ... YCn-1.
RAMP FUNCTION GENERATOR

The ramp function generator is used for the ramp-shaped adaption of the current actual value at the output to a specified set point.

The value at the HLG output is adapted linearly from the current actual value to the specified set point with the slope y'.

In doing so, the value at the output precisely runs through the amount of the set point during the time TH or TR. If the value at the HLG output has reached the set point, it no longer changes unless a new set point is specified.

### Block type

Function block with historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E01</td>
<td>BOOL Selection of the set point 0 or set point 1</td>
</tr>
<tr>
<td>S0</td>
<td>INT Set point 0</td>
</tr>
<tr>
<td>S1</td>
<td>INT Set point 1</td>
</tr>
<tr>
<td>TH_T</td>
<td>INT Start up time scaled to the cycle time</td>
</tr>
<tr>
<td>TR_T</td>
<td>INT Return time scaled to the cycle time</td>
</tr>
<tr>
<td>STOP</td>
<td>BOOL Latching the output at the current value</td>
</tr>
<tr>
<td>S</td>
<td>BOOL Setting the output to the INIT value</td>
</tr>
<tr>
<td>INIT</td>
<td>INT INIT value to which the output can be set</td>
</tr>
<tr>
<td>R</td>
<td>BOOL Reset the output to value 0</td>
</tr>
<tr>
<td>A</td>
<td>INT Output</td>
</tr>
</tbody>
</table>

### Description

The ramp function generator is used for the ramp-shaped adaption of the current actual value at the output to a specified set point.

The value at the HLG output is adapted linearly from the current actual value to the specified set point with the slope y'.

In doing so, the value at the output precisely runs through the amount of the set point during the time TH or TR. If the value at the HLG output has reached the set point, it no longer changes unless a new set point is specified.

The inputs and the output can neither be duplicated nor negated/inverted.

The slope y' of the ramp results from the specified time TH (start up time) or TR (return time) and the amount of the set point:

\[
\text{Slope } y' = \frac{\text{Set point amount}}{\text{TH or TR}}
\]

The slope is
- positive, if set point > actual value
- negative, if set point < actual value
- 0, if set point = 0

Therefore, the specified set point has two functions
- its amount defines the slope of the ramp in conjunction with the specified time TH or TR;
- it represents the value to which the current actual value must be adapted in a ramp shape.

The user can specify the start up time TH and the return time TR separately. The direction of the slope is defined on the basis of the set point. The direction of the slope then defines whether or not the running time TH or TR is used.

Slope y' positive → TH, i.e. the ramp runs upwards.
Slope y' negative → TR, i.e. the ramp runs downwards.

The start up time TH and the return time TR must be scaled to the program cycle time TZ, i.e. the following must be specified at the corresponding block inputs:
Start up time: TH/TZ
Return time: TR/TZ

The times are specified in milliseconds. The following applies to the time constants TH or TR:

\[ 0 < TH < 32767 \]
\[ 0 \leq TR \leq 32767 \]

Two set points can be planned (S0 and S1), whereby one of these set points is selected by the binary input E01 (set point selection).

The set points can assume the following values:

\[ -32767 \leq \text{set point} \leq +32767 \]

At any time, the output of the ramp function generator can be

- stopped at the current value
- set to an initial value
- reset (output = 0)

The STOP input has the highest priority and the R input has the lowest.

The values at the HLG inputs can be changed at any time in the user program. In this way, any (non-linear) adaption to the set point can be realized on the basis of the linear adaption of the actual value.

Caution:
Set point = 0 means that the slope of the ramp is also 0, i.e. the current actual value does not change. If it is intended to switch from an actual value unequal to 0 to an actual value of 0, a set point unequal to 0 must be specified and the output of the ramp function generator must be limited to 0 by a subsequent limiter. (On interpolation, the rounding transitions are based on calculation of integral numbers only).

**E01**

Boolean (BOOL)
One of the two set points is selected with input E01.

E01 = FALSE → set point S0
E01 = TRUE → set point

**S0**

Integer (INT)
The set point 0 is specified at input S0.

**S1**

Integer (INT)
The set point 0 is specified at input S0.

**TH_T**

Integer (INT)
The start up time is specified at input TH_T. At the same time, the start up time TH must be scaled to the cycle time T.

**STOP**

Boolean (BOOL)
The output can be latched to the current value by means of the STOP input.

STOP = FALSE → Output not latched
STOP = TRUE → Output is latched

The STOP input has higher priority than the inputs S and R.

**S**

Boolean (BOOL)
With input S, the output can be set to the initial value specified at input INIT.

S = FALSE → Output is not set to initial value.
S = TRUE → Output is set to initial value.

**INIT**

Integer (INT)
The initial value to which the output is to be set if required is specified at input INIT.

**R**

Boolean (BOOL)
The output can be set to the value 0 with the input R.

R = FALSE → Output is not reset
R = TRUE → Output is reset to value 0.

**A**

Integer (INT)
Output of the block.
Example 1:
The actual value is to be changed from 0 to the set point +500 (set point amount 500) and then from +500 to the set point -1000 (set point amount 1000).
The start up time is TH and the return time TR.
0 S: 500
1 S: -1000

TR(*) is the actual time until the actual value has reached -1000. During the time TH or TR, the actual value changes by the amount of the applied set point.

Example 2:
The actual value is to be changed from 0 to the set point +500 (set point amount 500) and then from +500 to the set point 1500 (set point amount 1500).
The start up time is TH and the return time TR.
0 S: 500
1 S: 1500

TR(*) is the actual time until the actual value has reached 1500. During the time TH or TR, the actual value changes by the amount of the applied set point.
Function call in IL

CAL HLG1(E01 := HLG_E01, S0 := HLG_S0, S1 := HLG_S1, TH_T := HLG_THT, TR_T := HLG_TRT, STOP := HLG_STOP, SET := HLG_SET, INIT := HLG_INIT, RESET := HLG_RES)

LD HLG1.A
ST HLG_A

Function call in ST

HLG1(E01 := HLG_E01, S0 := HLG_S0, S1 := HLG_S1, TH_T := HLG_THT, TR_T := HLG_TRT, STOP := HLG_STOP, SET := HLG_SET, INIT := HLG_INIT, RESET := HLG_RES);
HLG_A:=HLG1.A;

Note: In IL, the function call has to be performed in one line.
READ BINARY VARIABLE, INDEXED

The function block serves for the purpose of indexed reading of binary variables.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Block enabling</td>
</tr>
<tr>
<td>INDEX</td>
<td>INT</td>
<td>The index value is specified at input INDEX. The source variable results from the index and the basic variable. Value range: -16383 ≤ INDEX ≤ +16383 If INDEX is not within this range, the function block is not processed.</td>
</tr>
<tr>
<td>BASIS</td>
<td>DWORD</td>
<td>The basic variable is specified at input BASIS. The block ADR provides the corresponding binary flag address. The source variable results from the index and the basic variable.</td>
</tr>
<tr>
<td>ZIEL</td>
<td>BOOL</td>
<td>The target variable is specified at output ZIEL. The value of the selected source variable is assigned to the target variable ZIEL.</td>
</tr>
</tbody>
</table>

Description

The function block serves for the purpose of indexed reading of binary variables.

The source variable to be read is obtained from indexing the basic variable. The value of the source variable read is assigned to the target variable.

Note:
The basic variable must be either an addressable variable (%MX.., %IX.., %QX..) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

The inputs and outputs can neither be duplicated nor inverted nor negated.

FREI

Block enabling

FREI = FALSE: Block is not processed
FREI = TRUE: The value of the source variable is read and assigned to the target variable ZIEL.
Function call in IL

LD  IDLB_BAS
ADR
ST  IDLB1.BASIS
CAL  IDLB1(FREI := IDLB_FREI,
      INDEX := IDLB_INDEX)
LD  IDLB1.ZIEL
ST  IDLB_ZIEL

Note: In IL, the function call has to be performed in one line.

Function call in ST

IDLB1.BASIS := ADR(IDLB_BAS);
IDLB1(FREI := IDLB_FREI, INDEX := IDLB_INDEX);
IDLB_ZIEL := IDLB1.ZIEL;
READ WORD VARIABLE, INDEXED

The function block serves for the purpose of indexed reading of word variables.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>IDLm</td>
<td>Instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Block enabling</td>
</tr>
<tr>
<td>INDEX</td>
<td>INT</td>
<td>The index value is specified at input INDEX. The source variable results from the index and the basic variable. Value range: -32767 ≤ INDEX ≤ +32767</td>
</tr>
<tr>
<td>BASIS</td>
<td>INT</td>
<td>The basic variable is specified at input BASIS. The source variable results from the index and the basic variable.</td>
</tr>
<tr>
<td>ZIEL</td>
<td>INT</td>
<td>The target variable is specified at output ZIEL. The value of the selected source variable is assigned to the target variable ZIEL.</td>
</tr>
</tbody>
</table>

Description

The function block serves for the purpose of indexed reading of word variables.

The source variable to be read is obtained from indexing the basic variable. The value of the source variable read is assigned to the target variable.

Note:
The basic variable must be either an addressable variable (%MW.., %IW.., %QW..) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

The inputs and outputs can neither be duplicated nor inverted nor negated.

FREI

Block enabling

FREI = FALSE → Block is not processed
FREI = TRUE → The value of the source variable is read and assigned to the target variable ZIEL.

INDEX

The index value is specified at input INDEX. The source variable results from the index and the basic variable.

Value range: -32767 ≤ INDEX ≤ +32767

BASIS

The basic variable is specified at input BASIS. The source variable results from the index and the basic variable.

ZIEL

The target variable is specified at output ZIEL. The value of the selected source variable is assigned to the target variable ZIEL.
Function call in IL

CAL IDLM1(FREI := IDLM_FREI, INDEX := IDLM_INDEX, BASIS := IDLM_BAS)

LD IDLM1.ZIEL
ST IDLM_ZIEL

Note: In IL, the function call has to be performed in one line.

Function call in ST

IDLM1(FREI := IDLM_FREI, INDEX := IDLM_INDEX, BASIS := IDLM_BAS);
IDLM_ZIEL:=IDLM1.ZIEL;
WRITE BINARY VARIABLE, INDEXED

The function block serves for the purpose of indexed writing of binary variables.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>IDSB</td>
<td>Instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Block enabling</td>
</tr>
<tr>
<td>QUELL</td>
<td>BOOL</td>
<td>Source variable</td>
</tr>
<tr>
<td>INDEX</td>
<td>INT</td>
<td>The index and the basic variable result in the source variable</td>
</tr>
<tr>
<td>BASIS</td>
<td>DWORD</td>
<td>Basic address of the binary variable</td>
</tr>
</tbody>
</table>

Description

The function block serves for the purpose of indexed writing of binary variables.

When the block is enabled, the value of the source variable is read and assigned to the target variable. The target variable is defined by indexing the basic variable.

Note:
The basic variable must be either an addressable variable (%MX.., %IX.., %QX..) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

The inputs and outputs can neither be duplicated nor inverted nor negated.

### FREI

Block enabling

- FREI = FALSE → Block is not processed
- FREI = TRUE → The value of the source variable is read and assigned to the target variable.

### QUELL

The source variable is specified at input QUELL. The value of this variable is read and assigned to the target variable.

### INDEX

The index value is specified at input INDEX. The source variable results from the index and the basic variable.

Value range: \(-16383 \leq \text{INDEX} \leq +16383\)

If INDEX is not within this range, the function block is not processed.

### BASIS

The basic variable is specified at input BASIS. The block ADR provides the corresponding binary flag address. The source variable results from the index and the basic variable.
Function call IL

LD   IDSB_BAS
ADR
ST   IDSB1.BASIS
CAL  IDSB1(FREI := IDSB_FREI,
        QUELL := IDSB_QUELL,
        INDEX := IDSB_INDEX)

Function call in ST

IDSB1.BASIS := ADR(IDSB_BAS);
IDSB1(FREI := IDSB_FREI,
    QUELL := IDSB_QUELL, INDEX := IDSB_INDEX);

Note: In IL, the function call has to be performed in one line.
WRITE WORD VARIABLE, INDEXED

The function block serves for the purpose of indexed writing of word variables.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>IDSm</td>
<td>Instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Block enabling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frei = FALSE: Block is not processed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frei = TRUE: The value of the source variable is read and assigned to the target variable.</td>
</tr>
<tr>
<td>QUELL</td>
<td>INT</td>
<td>Source variable</td>
</tr>
<tr>
<td>INDEX</td>
<td>INT</td>
<td>The index and the basic variable result in the source variable</td>
</tr>
<tr>
<td>BASIS</td>
<td>INT</td>
<td>Basic variable</td>
</tr>
</tbody>
</table>

Description
The function block serves for the purpose of indexed writing of word variables.

When the block is enabled, the value of the source variable is read and assigned to the target variable. The target variable is defined by indexing the basic variable.

Note:
The basic variable must be either an addressable variable (%MW.., %IW.., %QW..) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

The inputs and outputs can neither be duplicated nor inverted nor negated.

FREI

Block enabling

FREI = FALSE
→ Block is not processed

FREI = TRUE
→ The value of the source variable is read and assigned to the target variable.

QUELL

The source variable is specified at input QUELL. The value of this variable is read and assigned to the target variable.

INDEX

The index value is specified at input INDEX. The source variable results from the index and the basic variable.

Value range: -32767 ≤ INDEX ≤ +32767

If INDEX is not within this range, the function block is not processed.

BASIS

The basic variable is specified at input BASIS. The source variable results from the index and the basic variable.
Function call in IL

CAL IDSM1(FREI := IDSM_FREI,
QUELL := IDSM_QUELL,
INDEX := IDSM_INDEX,
BASIS := IDSM_BAS)

Note: In IL, the function call has to be performed in one line.

Function call in ST

IDSM1(FREI := IDSM_FREI,
QUELL := IDSM_QUELL, INDEX := IDSM_INDEX,
BASIS := IDSM_BAS);
INITIALIZE MEMORY AREA IN THE
BINARY OPERAND MEMORY WITH ZERO

The function block serves for the purpose of bitwise initialization of memory areas in the operand memory with the value 0.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>INITSB</th>
<th>INITSB</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td></td>
<td>Enabling for non-recurring processing of the block (FALSE/TRUE edge)</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td></td>
<td>Number (n) of memory bits to be initialized</td>
</tr>
<tr>
<td>SVAR</td>
<td>DWORD</td>
<td></td>
<td>The n memory bits specified at input n are initialized as from this variable.</td>
</tr>
</tbody>
</table>

Description

The function block serves for the purpose of bit-wise initialization of memory areas in the operand memory with the value 0.

The block is run through precisely once in the event of a FALSE/TRUE edge at input FREI. In doing so, n memory bits are initialized with 0 starting from the variable specified at input VAR.

FREI BOOL

Block enabling. The block is run through precisely once in the event of a FALSE/TRUE edge at this input.

N INT

Number (n) of bit operands to be initialized (bit or BOOL). The number of bit operands to be initialized is specified at this input.

Value range: 0 ≤ n ≤ 32767

No initialization is performed if the value 0 is applied.

SVAR DWORD

Beginning from this variable address (inclusive), n memory bits are initialized with the value 0. The block ADR provides the corresponding binary flag address.

Function call in IL

| LD      | INSB_SVAR | ADR | ST | INITSB.SVAR | CAL | INITSB(FREI := INSB_FREI, N := INSB_N) |

Function call in ST

INITSB.SVAR := ADR(INSB_SVAR);
INITSB(FREI := INSB_FREI, N := INSB_N);
The function block serves for the purpose of double word wise initialization of memory areas in the operand memory with the value 0.

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling for non-recurring processing of the block (FALSE/TRUE edge)</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Number (n) of double words to be initialized</td>
</tr>
<tr>
<td>SVAR</td>
<td>DINT</td>
<td>The n memory double words specified at input n are initialized as from this variable address.</td>
</tr>
</tbody>
</table>

**Description**

The function block serves for the purpose of double word wise initialization of memory areas in the operand memory with the value 0.

The block is run through precisely once in the event of a FALSE/TRUE edge at input FREI. In doing so, n memory double words are initialized with 0 starting from the variable specified at input VAR.

**Function call in IL**

```
CAL INITSD1(FREI := INSD_FREI, N := INSD_N, SVAR := INSD_SVAR)
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
INITSD1(FREI := INSD_FREI, N := INSD_N, SVAR := INSD_SVAR);
```
INITIALIZE MEMORY AREA IN THE WORD OPERAND MEMORY WITH ZERO  

The function block serves for the purpose of word wise initialization of memory areas in the operand memory with the value 0.

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>FREI</th>
<th>BOOL</th>
<th>Enabling for non-recurring processing of the block (FALSE/TRUE edge)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>INT</td>
<td></td>
<td>Number ( n ) of memory words to be initialized</td>
</tr>
<tr>
<td>SVAR</td>
<td>INT</td>
<td></td>
<td>The ( n ) memory words specified at input ( n ) are initialized as from this variable address.</td>
</tr>
</tbody>
</table>

**Description**

The function block serves for the purpose of word wise initialization of memory areas in the operand memory with the value 0.

The block is run through precisely once in the event of a FALSE/TRUE edge at input FREI. In doing so, \( n \) memory words are initialized with 0 starting from the variable specified at input VAR.

**FREI**

Block enabling. The block is run through precisely once in the event of a FALSE/TRUE edge at this input.

**Function call in IL**

```
CAL INITSW1(FREI := INSW_FREI, 
             N := INSW_N, SVAR := INSW_SVAR)
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
INITSW1(FREI := INSW_FREI, N := INSW_N, 
        SVAR := INSW_SVAR);
```
INITIALIZE VARIABLES, BINARY

The block serves to initialize BINARY variables.

The INITVB number indicates the maximum number of variables to be initialized. The following blocks are available:

INITVB16 Variables initialization (binary) for a max. of 16 variables
INITVB32 Variables initialization (binary) for a max. of 32 variables

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>INITVB(16..32)</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling for non-recurring processing of the block (FALSE/TRUE edge)</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Number of variables to be initialized</td>
</tr>
<tr>
<td>W0..Wn-1</td>
<td>BOOL</td>
<td>Initialization value for the subsequent variable;</td>
</tr>
<tr>
<td>VR0..VRn-1</td>
<td>DWORD</td>
<td>Variable address to be initialized</td>
</tr>
</tbody>
</table>

Description

The block serves to initialize BINARY variables with numeric values.

The initialization values are specified at the block inputs W0..Wn-1. The block is run through precisely once in the event of a FALSE/TRUE edge at input FREI.

FREI | BOOL
The block is run through precisely once in the event of a FALSE/TRUE edge at input FREI.

n | INT
The number (n) of variables to be initialized is specified at input n.

W0..Wn-1 | BOOL
The initialization value for the subsequent variable is specified at input W0.

The inputs W0 and VR0 can be duplicated in pairs.

The following allocation applies:

Value 0 → Variable 0
:   :
Value n-1 → Variable n-1

VR0..VRn-1 | DWORD
The first variable to be initialized is specified at input VR0. The block ADR provides the corresponding binary flag address.

The inputs VR0 and W0 can be duplicated in pairs.
Function call in IL

LD
ADR
ST
LD
ADR
ST
CAL

INITVB_V0
INITVB_V1
INITVB_V2
INITVB161.V0
INITVB161.V1
INITVB161.V2
INITVB161(FREI := INITVB_FREI, N := 3,
W0 := INITVB_W0, W1 := INITVB_W1,
W2 := INITVB_W2)

Function call in ST

INITVB161.V0 := ADR(INITVB_V0);
INITVB161.V1 := ADR(INITVB_V1);
INITVB161.V2 := ADR(INITVB_V2);
INITVB161(FREI := INITVB_FREI, N := 3,
W0 := INITVB_W0, W1 := INITVB_W1,
W2 := INITVB_W2);

Note: In IL, the function call has to be performed in one line.
The block serves to initialize word variables with numeric values.

The INITVW number indicates the maximum number of variables to be initialized. The following blocks are available:

- **INITVW16**: Variables initialization (word) for a max. of 16 variables
- **INITVW32**: Variables initialization (word) for a max. of 32 variables

**Parameters**

- **Instance**: INITVW(16..32)
- **FREI**: BOOL Enabling for non-recurring processing of the block (FALSE/TRUE edge)
- **N**: INT Number of variables to be initialized
- **W0..Wn-1**: INT Initialization value for the subsequent variable;
- **VR0..VRn-1**: INT Variable to be initialized

**Description**

The block serves to initialize word variables with numeric values.

The initialization values are specified at the block inputs W0..Wn-1. The block is run through precisely once in the event of a FALSE/TRUE edge at input FREI.

**FREI**: BOOL The block is run through precisely once in the event of a FALSE/TRUE edge at input FREI.

**n**: INT The number (n) of variables to be initialized is specified at input n.

**W0..Wn-1**: INT

The initialization value for the subsequent variable is specified at input W0.

The inputs W0 and VR0 can be duplicated in pairs.

The following allocation applies:

<table>
<thead>
<tr>
<th>Value 0</th>
<th>→</th>
<th>Variable 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value n-1</td>
<td>→</td>
<td>Variable n-1</td>
</tr>
</tbody>
</table>

**VR0..VRn-1**: INT

The first variable to be initialized is specified at input VR0.

The inputs VR0 and W0 can be duplicated in pairs.
Function call in IL

CAL INITVW161(FREI := INITVW_FREI, N := 3,
W0 := INITVW_W0, W1 := INITVW_W0,
W2 := INITVW_W0, V0 := INITVW_V0,
V1 := INITVW_V0, V2 := INITVW_V0)

Note: In IL, the function call has to be performed in one line.

Function call in ST

INITVW161(FREI := INITVW_FREI, N := 3,
W0 := INITVW_W0, W1 := INITVW_W0,
W2 := INITVW_W0, V0 := INITVW_V0,
V1 := INITVW_V0, V2 := INITVW_V0);
INTEGRATOR (EXTENDED)

This block generates the integral of the controlled variable \( x \) multiplied by the proportional coefficient \( KP \).

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>INT</td>
<td>Instance name</td>
</tr>
<tr>
<td>X</td>
<td>INT</td>
<td>Controlled variable</td>
</tr>
<tr>
<td>T1_TZ</td>
<td>INT</td>
<td>Integration time scaled to the cycle time</td>
</tr>
<tr>
<td>RES</td>
<td>INT</td>
<td>Reset output ( Y ) to 0</td>
</tr>
<tr>
<td>STOP</td>
<td>BOOL</td>
<td>Integrator stop</td>
</tr>
<tr>
<td>SET</td>
<td>BOOL</td>
<td>Set output to ( INIT ) value</td>
</tr>
<tr>
<td>INIT</td>
<td>INT</td>
<td>Initial value</td>
</tr>
<tr>
<td>KP</td>
<td>INT</td>
<td>Proportional coefficient, output as a percentage value</td>
</tr>
<tr>
<td>OG</td>
<td>INT</td>
<td>High limit for the manipulated variable ( Y )</td>
</tr>
<tr>
<td>UG</td>
<td>INT</td>
<td>Low limit for the manipulated variable ( Y )</td>
</tr>
<tr>
<td>OG_MELD</td>
<td>BOOL</td>
<td>Output ( Y ) has reached top limit</td>
</tr>
<tr>
<td>UG_MELD</td>
<td>BOOL</td>
<td>Output ( Y ) has reached low limit</td>
</tr>
<tr>
<td>Y</td>
<td>INT</td>
<td>Manipulated variable</td>
</tr>
</tbody>
</table>

**Description**

This block generates the integral of the controlled variable \( x \) multiplied by the proportional coefficient \( KP \). The integrator output \( Y \) can be manipulated as follows:

- It can be set to the value 0 by a 1 signal at input RES (reset).
- It can be latched to the current value by a TRUE signal at input STOP.
- It can be set to the initial at input INIT by a TRUE signal at input SET.
- It can be limited to a maximum value specified at input OG (high limit).
- It can be limited to a minimum value specified at input UG (low limit).

The inputs and outputs can neither be duplicated nor negated/inverted.
Transfer function:

\[ F(s) = \frac{KP}{s \cdot T} \]

The operand for the controlled variable is specified at input X.

**X**  
INT

The integration time is specified at input T1_TZ. In this case it must be scaled to the cycle time. During the time T1 the output Y of the integrator changes by the value KP * X.

**T1_TZ**  
INT

Value range: 0 ≤ T1_TZ ≤ 328

- If values are specified which are beyond the admissible value range the PLC generally uses the value 328.
- A large integration time (T1) can be achieved by choosing a large cycle time T, too. If the block is used within a run number block, the cycle time of the run number block is valid for block INTK and not the cycle time (KD 0,0) of the PLC program.
RES

The output Y can be reset to the value 0 with the input RES. Integration then again begins as from the value 0.

STOP

Integration can be stopped with input STOP.

STOP = FALSE
→ integration is not stopped

STOP = TRUE
→ integration is stopped, i.e. output Y no longer changes.

SET

With the input SET, the manipulated value Y can be set to the initial value specified at input INIT. Integration then again begins as from the initial value.

S = FALSE → no setting
S = TRUE → output Y is set to the specified initial value

*) Priority sequence for the inputs STOP, SET and RES:
RES highest priority
STOP
SET lowest priority

INIT

The initial value to which output Y is to be set is specified at input INIT if required.

KP

The proportional coefficient is specified at input KP. It serves to weight the controlled variable at input X. Weighting is achieved by multiplying the controlled variable by the proportional coefficient. The proportional coefficient is specified as a percentage.

Example:
1 = 1 percent
55 = 55 percent
100 = 100 percent
1000 = 1000 percent
-100 = -100 percent

OG

The manipulated variable Y can be limited to a value range. The high limit for the manipulated variable Y is specified at input OG.

UG

The manipulated variable Y can be limited to a value range. The low limit for the manipulated variable Y is specified at input UG.

OG_MELD

Whether the value at output Y has reached the specified top limit is signalized at output OG_MELD. Integration is stopped automatically when the limit is reached.

OG_MELD = FALSE → Y has not reached the limit
OG_MELD = TRUE
→ Y has reached the limit

UG_MELD

Whether the value at output Y has reached the specified low limit is signalized at output UG_MELD. Integration is stopped automatically when the limit is reached.

UG_MELD = FALSE → Y has not reached the limit
UG_MELD = TRUE
→ Y has reached the limit

Y

The manipulated variable (output value of the integrator) is output through the output Y.
Function call in IL

CAL INTK1(X := INTK_X, T1_TZ := INTK_T1TZ,
RES := INTK_RES, STOP := INTK_STOP,
SET := INTK_SET, INIT := INTK_INIT,
KP := INTK_KP, OG := INTK_OG,
UG := INTK_UG)
LD INTK1.OG_MELD
ST INTK_MOG
LD INTK1.UG_MELD
ST INTK_MUG
LD INTK1.Y
ST INTK_Y

Note: In IL, the function call has to be performed in one line.

Function call in ST

INTK1(X := INTK_X, T1_TZ := INTK_T1TZ,
RES := INTK_RES, STOP := INTK_STOP,
SET := INTK_SET, INIT := INTK_INIT,
KP := INTK_KP, OG := INTK_OG, UG := INTK_UG);
INTK_MOG:=INTK1.OG_MELD;
INTK_MUG:=INTK1.UG_MELD;
INTK_Y:=INTK1.Y;
ILLUMINATION PUSHBUTTON CONTROL

This function block changes the polarity at output A with every FALSE/TRUE edge at input E (like an impulse relay).

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>LDT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIN</td>
<td>BOOL</td>
<td>Dynamic input for activating output A</td>
</tr>
<tr>
<td>AUS</td>
<td>BOOL</td>
<td>Dynamic input for deactivating output A</td>
</tr>
<tr>
<td>E</td>
<td>BOOL</td>
<td>Dynamic input for activating and deactivating output A</td>
</tr>
<tr>
<td>RES</td>
<td>BOOL</td>
<td>Static input for deactivating output A</td>
</tr>
<tr>
<td>A</td>
<td>BOOL</td>
<td>Output</td>
</tr>
<tr>
<td>EINI</td>
<td>BOOL</td>
<td>Pulse for the duration of one PLC program cycle when activating output A</td>
</tr>
<tr>
<td>AUSI</td>
<td>BOOL</td>
<td>Pulse for the duration of one PLC program cycle when deactivating output A</td>
</tr>
</tbody>
</table>

Description see next page
This function block changes the polarity at output A with every FALSE/TRUE edge at input E (like an impulse relay). However, this only applies if input RES is inactive. Output A can be influenced additionally with the inputs EIN, AUS and RES.

All inputs except RES are edge-controlled, i.e. they react only to a FALSE/TRUE edge. The input RES is active for as long as a static 1 signal is present at it. Brief pulses are available at output EINI and AUSI when a status change of output A occurs.

<table>
<thead>
<tr>
<th>EIN</th>
<th>AUS</th>
<th>E</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>H</td>
<td>L</td>
<td>NA</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>&gt;</td>
<td>L</td>
<td>NA</td>
</tr>
<tr>
<td>&gt;</td>
<td>&lt;</td>
<td>L</td>
<td>L</td>
<td>&lt;=</td>
</tr>
</tbody>
</table>

**EIN**

Output A is set with every FALSE/TRUE edge at input EIN. However, this only applies when the inputs AUS, E and R are inactive. Therefore, the input EIN has the lowest priority of all inputs.

**AUS**

Output A is reset with every FALSE/TRUE edge at input AUS. Therefore, input AUS has a higher priority than input EIN, but a lower priority than the inputs E and RES.

**Function call in IL**

```plaintext
CAL LDT1(EIN := LDT_EIN, AUS := LDT_AUS, E := LDT_E, RES := LDT_RES)
LD LDT1.EINI
ST LDT_EINI
LD LDT1.AUSI
ST LDT_AUSI
LD LDT1.A
ST LDT_A
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```plaintext
LDT1(EIN := LDT_EIN, AUS := LDT_AUS, E := LDT_E, RES := LDT_RES);
LDT_EINI:=LDT1.EINI;
LDT_AUSI:=LDT1.AUSI;
LDT_A:=LDT1.A;
```

**E**

With every FALSE/TRUE edge at input E, the block changes the priority at output A. However, this only applies if input R is inactive. Therefore, input E has a higher priority than the inputs EIN and AUS, but a lower priority than input RES.

**RES**

Output A is reset when a static TRUE signal is present at input RES.

The RES input has a higher priority than the inputs E and EIN. A priority conflict with the input AUS is not possible because both inputs have the same effect on the outputs.

**A**

Output A is changed corresponding to the definition of the inputs.

**EINI**

A pulse is generated at output EINI with every FALSE/TRUE edge at output A. The length of this pulse is 1 program cycle.

**AUSI**

A pulse is generated at output AUSI with every TRUE/FALSE edge at output A. The length of this pulse is 1 program cycle.
STACK MEMORY LAST IN FIRST OUT, BINARY

The function block realizes a stack memory for binary data from which the data written in last are again read out first (last in/first out).

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>LIFOB</td>
</tr>
<tr>
<td>LADE</td>
<td>BOOL</td>
</tr>
<tr>
<td>LESE</td>
<td>BOOL</td>
</tr>
<tr>
<td>RESET</td>
<td>BOOL</td>
</tr>
<tr>
<td>E</td>
<td>BOOL</td>
</tr>
<tr>
<td>ANZ</td>
<td>INT</td>
</tr>
<tr>
<td>ANF</td>
<td>DWORD</td>
</tr>
<tr>
<td>A</td>
<td>BOOL</td>
</tr>
<tr>
<td>FST</td>
<td>INT</td>
</tr>
<tr>
<td>LEER</td>
<td>BOOL</td>
</tr>
<tr>
<td>VOLL</td>
<td>BOOL</td>
</tr>
</tbody>
</table>

Description

The function block realizes a stack memory for binary data from which the data written in last are again read out first (last in/first out).

The inputs and outputs can neither be duplicated nor negated/inverted.

LADE

The value present at input E is transferred to the next free position of the LIFO by means of a TRUE signal at input LADE. If the LIFO is "full" and a "load" signal is present, the new value will not be read in. A new value can only be read after a value has been read out. It is then transferred to the first position of the LIFO.

LESE

A TRUE signal at input LESE results in the output of the current LIFO value through output A. If the LIFO is "empty", a pending read order is ignored and the value FALSE is output at output A. Output A is always set to FALSE if there is no read order.

LADE and LESE

If load and read orders are present simultaneously, the value to be loaded is forwarded directly from input E to output A. At the same time, the LIFO does not store the value. The LIFO does not change its filling level in this operating mode. The value output through output FST remains constant. If the LIFO is empty, output LEER remains TRUE and output FST is set to 0.

RESET

A FALSE → TRUE edge at input RESET results in the reset of the block. Therefore, values read in before are no longer available. Output LEER is set to TRUE and the outputs FST and A are set to 0 (FALSE).

RESET and LADE

If RESET and LADE signals are present simultaneously the reset is performed first and then directly afterwards the load operation. Therefore, the LIFO is reset and the new value to be read in is then immediately stored in the LIFO as the first value.
RESET and LADE and LESE
The value at input E is forwarded directly to output A. Output LEER is set permanently to TRUE and output FST is set permanently to 0.

E BOOL
The value to be transferred into the LIFO is specified at input E.

ANZ INT
The number of required memory locations (bytes) of the LIFO is specified at input ANZ. The number of memory locations corresponds to the number of binary values to be stored.

ANF DWORD
The LIFO memory start address is specified at input ANF. The block ADR provides the corresponding binary flag address. The LIFO begins with the specified flag.

A BOOL
When the LIFO is read, the current value is available at output A. If no read order is available, the value FALSE is output.

FST INT
Output FST indicates the current filling level of the LIFO at any time. The filling level is the number of binary values stored in the LIFO.

LEER BOOL
The output LEER indicates whether or not the LIFO is empty.

VOLL BOOL
The output VOLL indicates whether or not the LIFO is full.

No further values can be read in if the LIFO is full. A new value can only be read in after a value has been read out. The new value is positioned to the beginning of the LIFO.

Function call in IL
LD LIFO_ANF
ADR
ST LIFO_ANF

LD LIFO.FST
ST LIFO_FST
LD LIFO1.LEER
ST LIFO_LEER
LD LIFO1.VOLL
ST LIFO.VOLL
LD LIFO1.A
ST LIFO_A

Note: In IL, the function call has to be performed in one line.

Function call in ST
LIFO1.ANF := ADR(LIFO_ANF);
LIFO1(LADE := LIFO_LADE, LESE := LIFO_LESE,
RESET := LIFO_RES, E := LIFO_E,
ANZ := LIFO_ANZ);
LIFO_FST := LIFO1.FST;
LIFO_LEER := LIFO1.LEER;
LIFO.VOLL := LIFO1.VOLL;
LIFO_A := LIFO1.A;

STACK MEMORY LAST IN FIRST OUT, WORD

The function block realizes a stack memory for word data from which the data written in last are again read out first (last in/first out).

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>LIFOW</td>
<td>Instance name</td>
</tr>
<tr>
<td>LADE</td>
<td>BOOL</td>
<td>Load LIFO</td>
</tr>
<tr>
<td>LESE</td>
<td>BOOL</td>
<td>Read LIFO</td>
</tr>
<tr>
<td>RESET</td>
<td>BOOL</td>
<td>Reset LIFO, FALSE→TRUE edge</td>
</tr>
<tr>
<td>E</td>
<td>INT</td>
<td>Input of data in the LIFO</td>
</tr>
<tr>
<td>ANZ</td>
<td>INT</td>
<td>Number of memory locations (bytes)</td>
</tr>
<tr>
<td>ANF</td>
<td>INT</td>
<td>Start address of the LIFO in the flag area</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Output of data from the LIFO</td>
</tr>
<tr>
<td>FST</td>
<td>INT</td>
<td>Filling level of the LIFO</td>
</tr>
<tr>
<td>LEER</td>
<td>BOOL</td>
<td>LIFO empty</td>
</tr>
<tr>
<td>VOLL</td>
<td>BOOL</td>
<td>LIFO full</td>
</tr>
</tbody>
</table>

**Description**

The function block realizes a stack memory for word data from which the data written in last are again read out first (last in/first out).

The inputs and outputs can neither be duplicated nor negated/inverted.

**LADE**

The value present at input E is transferred to the next free position of the LIFO by means of a TRUE signal at input LADE. If the LIFO is “full” and a “load” signal is present, the new value will not be read in. A new value can only be read if a value has been read out first. It is then transferred to the first position of the LIFO.

**LESE**

A TRUE signal at input LESE results in the output of the current LIFO value through output A. If the LIFO is “empty”, a pending read order is ignored and the value 0 is output at output A. Output A is always set to 0 if there is no read order.

**LADE und LESE**

If load and read orders are present simultaneously, the value to be loaded is forwarded directly from input E to output A. At the same time, the LIFO does not store the value. The LIFO does not change its filling level in this operating mode. The value output through output FST remains constant. If the LIFO is empty, output LEER remains TRUE and output FST is set to 0.

**RESET**

A FALSE → TRUE edge at input RESET results in the reset of the block. Therefore, values read in before are no longer available. Output LEER is set to TRUE and the outputs FST and A are set to 0.

**RESET and LADE**

If RESET and LADE signals are present simultaneously the reset is performed first and then directly afterwards the load operation. Therefore, the LIFO is reset and the new value to be read in is then immediately stored in the LIFO as the first value.
RESET and LADE and LESE

The value at input E is forwarded directly to output A. Output LEER is set permanently to TRUE and output FST is set permanently to 0.

E

The value to be transferred into the LIFO is specified at input E.

ANZ

The number of required memory locations (words) of the LIFO is specified at input ANZ. The number of memory locations corresponds to the number of binary values to be stored.

ANF

The LIFO memory start address is specified as a binary flag at input ANF. The LIFO begins with the specified flag.

A

When the LIFO is read, the current value is available at output A. If no read order is available, the value 0 is output.

FST

Output FST indicates the current filling level of the LIFO at any time. The filling level is the number of binary values stored in the LIFO.

LEER

The output LEER indicates whether or not the LIFO is empty.

VOLL

The output VOLL indicates whether or not the LIFO is full.

Function call in IL

```
LD LIFO_ANF
ADR
ST LIFOW1.ANF
CAL LIFOW1(LADE := LIFO_LADE,
LESE := LIFO_LESE, RESET := LIFO_RES,
E := LIFO_E, ANZ := LIFO_ANZ)
LD LIFOW1.FST
ST LIFO_FST
LD LIFOW1.LEER
ST LIFO_LEER
LD LIFOW1.VOLL
ST LIFO_VOLL
LD LIFOW1.A
ST LIFO_A
```

Note: In IL, the function call has to be performed in one line.

Function call in ST

```
LIFOW1.ANF := ADR(LIFO_ANF);
LIFOW1(LADE := LIFO_LADE,
LESE := LIFO_LESE, RESET := LIFO_RES,
E := LIFO_E, ANZ := LIFO_ANZ);
LIFO_FST:=LIFOW1.FST;
LIFO_LEER:=LIFOW1.LEER;
LIFO_VOLL:=LIFOW1.VOLL;
LIFO_A:=LIFOW1.A;
```
LIST ALLOCATOR

The function block has a list of word data at its inputs. With a list pointer, it selects a value out of this list and applies it at its output.

The LIZU number indicates the maximum number of word data. The following list allocators are available:

- LIZU8: List allocator for max. 8 word data
- LIZU16: List allocator for max. 16 word data
- LIZU32: List allocator for max. 32 word data
- LIZU64: List allocator for max. 64 word data
- LIZU256: List allocator for max. 256 word data

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>LIZU(8..256)</td>
<td>Instance name</td>
</tr>
<tr>
<td>ZEIG</td>
<td>INT</td>
<td>Pointer to the list of word data</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Number of word data in the list</td>
</tr>
<tr>
<td>E0..En-1</td>
<td>INT</td>
<td>List of word data</td>
</tr>
<tr>
<td>A_E</td>
<td>BOOL</td>
<td>0 ≤ ZEIG &lt; n, i.e. pointer in the valid range</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Selected word value</td>
</tr>
</tbody>
</table>

**Description**

The function block has a list of word data at its inputs. With a list pointer, it selects a value out of this list and applies it at its output.

The inputs and outputs cannot be inverted. The input E0 can be duplicated.

**ZEIG**

The pointer to the word value to be selected from the list is specified at input ZEIG.

The following allocation applies:

- ZEIG = 0 → Word value at E0
- ZEIG = 1 → Word value at E1
- ...                                      
- ZEIG = n-1 → Word value at En-1

The value at input ZEIG is monitored for validity. The result of this range check is signalized at output A_E.

**Allowed range:**

- 0 ≤ ZEIG ≤ n-1

With n: Number of the inputs E0..En-1.

No allocation to output A takes place if the value at input ZEIG is outside the allowed range.

**E0-..-En-1**

The number of the word values planned at the inputs E0..En-1 is specified at input n.

**A_E**

Output A_E indicates whether the value of the list pointer (input ZEIG) is within the allowed range.

**Allowed range:**

- 0 ≤ ZEIG ≤ n-1

With n: Number of the inputs E0..En-1.

The following applies:

- ZEIG in the allowed range → A_E = TRUE
- ZEIG in the forbidden range → A_E = FALSE
If the list pointer has a forbidden value, no allocation of a value to output A takes place because no word value can be selected. In this case, output A is not updated.

The value of the selected word value is assigned to the operand at output A.

Function call in IL

```
CAL LIZU81(ZEIG := LIZU_ZEIG, n := 3,
            E0 := LIZU_E0, E1 := LIZU_E1,
            E2 := LIZU_E2)
LD LIZU81.A
ST LIZU_A
LD LIZU81.A_E
ST LIZU_AE
```

Note: In IL, the function call has to be performed in one line.

Function call in ST

```
LIZU81(ZEIG := LIZU_ZEIG,
        n := 3,
        E0 := LIZU_E0,
        E1 := LIZU_E1,
        E2 := LIZU_E2)
        LIZU_A := LIZU81.A
        LIZU_AE := LIZU81.AE
```
RUN NUMBER BLOCK

The block controls the processing of a program part. This program part is called run number block and begins with the function block LZB and ends with the appropriate target label specified at the block output MRK. This program part is processed as follows depending on the value of the operand at input E1.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>E1</th>
<th>MRK</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZB</td>
<td>INT</td>
<td>SPRUNG</td>
</tr>
</tbody>
</table>

Description

This program part is processed as follows depending on the value of the operand at input E1:

- **E1 = 0**: Program part is not processed
- **E1 = 1**: Program part is processed during every cycle
- **E1 = 2**: Program part is processed during every second cycle
- **E1 = n**: Program part is processed during every nth cycle

MRK BOOL

At this output, a jump instruction with corresponding jump destination must be specified. Output MRK only signalizes, whether the subsequent program part is processed or not.

The following applies:

- **MRK = FALSE**: Processing of program part
- **MRK = TRUE**: No processing of program part

Function call in IL

- **CAL** LZB1(E1 := LZB_E1)
- **LD** LZB1.MRK
- **JMPC** MARKE

Note: In IL, the function call has to be performed in one line.

Function call in ST

- **LZB1(E1 := LZB_E1);**
- **IF** (LZB1.MRK)
  - **THEN** ....;
- **END_IF**
This block realizes a majority element.

### Parameters

| E1 | BOOL | Operand 1 |
| E2 | BOOL | Operand 2 |
| E3 | BOOL | Operand 3 |
| A  | BOOL | Result    |

### Description

This block realizes a majority element. If at least 2 of the 3 binary operands at the inputs E1, E2 and E3 have the state TRUE, then the state TRUE is assigned to the binary operand at output A. The state FALSE is assigned if this is not the case.

The inputs and the output cannot be duplicated.

### Function call in IL

```
LD MAJ_E1
MAJ MAJ_E2,MAJ_E3
ST MAJ_A
```

### Function call in ST

```
MAJ_A := MAJ(MAJ_E1,MAJ_E2,MAJ_E3);
```
The individual bits of the operand at input E1 are compared to the bits of the operand at input MASKE. The result of the comparison is signalized at the outputs ALLE and KEIN.

**Block type**

Function block without historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>MASKE</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Input value</td>
</tr>
<tr>
<td>MASKE</td>
<td>INT</td>
<td>Mask</td>
</tr>
<tr>
<td>ALLE</td>
<td>BOOL</td>
<td>All bits agree</td>
</tr>
<tr>
<td>KEIN</td>
<td>BOOL</td>
<td>No bit agrees</td>
</tr>
</tbody>
</table>

**Description**

The individual bits of the operand at input E1 are compared to the bits of the operand at input MASKE. The result of the comparison is signalized at the outputs ALLE and KEIN.

If all bits are set in the operand at input E1 which are set in the operand at input MASKE, the following applies to the outputs:

ALLE = 1
KEIN = 0

If none of the bits are set in the operand at input E1 which are set in the operand at input MASKE, the following applies to the outputs:

ALLE = FALSE
KEIN = TRUE

If only some of the bits are set in the operand at input E1 which are set in the operand at input MASKE, the following applies to the outputs:

ALLE = FALSE
KEIN = FALSE

The inputs and outputs can neither be duplicated nor negated/inverted.

**Function call in IL**

```java
CAL MASKE1(E1 := MASK_E1,
            MASKE := MASK_MASKE)
LD MASKE1.KEIN
ST MASK_KEIN
LD MASKE1.ALLE
ST MASK_ALLE
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```java
MASKE1(E1 := MASK_E1,
       MASKE := MASK_MASKE);
MASK_KEIN:=MASKE1.KEIN;
MASK_ALLE:=MASKE1.ALLE;
```
The individual bits of the operand at input E1 are compared to the bits of the operand at input MASKE.

The result of the comparison is signaled at the outputs ALLE and KEIN.

**Block type**
Function block without historical values

**Parameters**

| Instance | MASKED | Instance name
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Input value</td>
</tr>
<tr>
<td>MASKE</td>
<td>DINT</td>
<td>Mask</td>
</tr>
<tr>
<td>ALLE</td>
<td>BOOL</td>
<td>All bits agree</td>
</tr>
<tr>
<td>KEIN</td>
<td>BOOL</td>
<td>No bit agrees</td>
</tr>
</tbody>
</table>

**Description**

The individual bits of the operand at input E1 are compared to the bits of the operand at input MASKE.

The result of the comparison is signaled at the outputs ALLE and KEIN.

If all bits are set in the operand at input E1 which are set in the operand at input MASKE, the following applies to the outputs:

ALLE = TRUE
KEIN = FALSE

If none of the bits are set in the operand at input E1 which are set in the operand at input MASKE, the following applies to the outputs:

ALLE = FALSE
KEIN = TRUE

If only some of the bits are set in the operand at input E1 which are set in the operand at input MASKE, the following applies to the outputs:

ALLE = FALSE
KEIN = FALSE

The inputs and outputs can neither be duplicated nor negated/inverted.

**Function call in IL**

CAL MASKED1(E1 := MASKD_E1, MASKED := MASKD_MASKE)
LD MASKED1.KEIN
ST MASKD_KERN
LD MASKED1.ALLE
ST MASKD_ALLE

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

MASKED1(E1 := MASKD_E1, MASKED := MASKD_MASKE); MASKD_KERN:=MASKED1.KEIN;
MASKD_ALLE:=MASKED1.ALLE;
MAXIMUM VALUE GENERATOR AS A FUNCTION OF TIME, WORD

This function block determines the maximum value of a signal up to the current point of time by evaluating the time behaviour of the signal.

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>MAZ</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Input value whose time maximum has to be determined</td>
</tr>
<tr>
<td>SET</td>
<td>BOOL</td>
<td>Set input</td>
</tr>
<tr>
<td>INIT</td>
<td>INT</td>
<td>Initial value</td>
</tr>
<tr>
<td>MAZ</td>
<td>INT</td>
<td>Maximum value</td>
</tr>
</tbody>
</table>

**Description**

This function block determines the maximum value of a signal up to the current point of time by evaluating the time behaviour of the signal.

The value of the operand at input E1 is compared to the previously occurred maximum value. If the input value E1 is higher than the previously occurred maximum, the input value is the new maximum value and is assigned to the operand at output MAZ. If the input value E1 is less than the previously occurred maximum value, the previous maximum value is assigned to the output.

Output MAZ is set to the value of the operand at input INIT (initial value) with the FALSE $\rightarrow$ TRUE edge at binary input SET.

The following applies:

- $E1 < MAZ \rightarrow MAZ = MAZ$
- $E1 > MAZ \rightarrow MAZ = E1$
- $SET = FALSE \rightarrow TRUE$ edge $\rightarrow MAZ = INIT$

The inputs and the output can neither be duplicated nor negated/inverted.

**Function call in IL**

```
CAL MAZ1(E1 := MAZ_E1, SET := MAZ_SET, INIT := MAZ_INIT)
LD MAZ1.MAZ
ST MAZ_MAZ
```

**Note:** In IL, the function call has to be performed in one line.

**Function call in ST**

```
MAZ1(E1 := MAZ_E1, SET := MAZ_SET, INIT := MAZ_INIT);
MAZ_MAZ:=MAZ1.MAZ;
```
MAXIMUM VALUE GENERATOR AS A FUNCTION OF TIME, DOUBLE WORD

This function block determines the maximum value of a signal up to the current point of time by evaluating the time behaviour of the signal.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Input value whose time maximum has to be determined</td>
</tr>
<tr>
<td>SET</td>
<td>BOOL</td>
<td>Set input</td>
</tr>
<tr>
<td>INIT</td>
<td>DINT</td>
<td>Initial value</td>
</tr>
<tr>
<td>MAZ</td>
<td>DINT</td>
<td>Maximum value</td>
</tr>
</tbody>
</table>

Description

Output MAZ is set to the value of the operand at input INIT (initial value) with the FALSE → TRUE edge at binary input SET.

The following applies:

- E1 < MAZ → MAZ = MAZ
- E1 > MAZ → MAZ = E1
- SET = FALSE → TRUE edge → MAZ = INIT

The inputs and the output can neither be duplicated nor negated/inverted.

Function call in IL

```
CAL MAZD1(E1 := MAZD_E1, 
SET := MAZD_SET, INIT := MAZD_INIT)
LD MAZD1.MAZ
ST MAZD_MAZ
```

Function call in ST

```
MAZD1(E1 := MAZD_E1, SET := MAZD_SET, 
INIT := MAZD_INIT);
MAZD_MAZ:=MAZD1.MAZ;
```

Note: In IL, the function call has to be performed in one line.
MONOSTABLE ELEMENT »ABORT«

A FALSE/TRUE edge at input E causes a FALSE/TRUE edge at output A. If input E remains at TRUE level, a TRUE/FALSE edge is applied at output A after the duration T has elapsed.

Output A is also immediately set back to FALSE level if input E returns to FALSE level before the time period T has expired.

Maximum time offset at the output: < 1 cycle time

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>MOA</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>BOOL</td>
<td>Input signal</td>
</tr>
<tr>
<td>T</td>
<td>TIMER</td>
<td>Pulse length</td>
</tr>
<tr>
<td>A</td>
<td>BOOL</td>
<td>Output signal</td>
</tr>
</tbody>
</table>

Description

A FALSE/TRUE edge at input E causes a FALSE/TRUE edge at output A. If input E remains at TRUE level, a TRUE/FALSE edge is applied at output A after the duration T has elapsed.

Output A is also immediately set back to FALSE level if input E returns to FALSE level before the time period T has expired.

Maximum time offset at the output: < 1 cycle time

Reasonable range for T: > 1 cycle time

The inputs and the output can neither be duplicated nor inverted.

Function call in IL

CAL MOA1(E := MOA_E, T := T#200ms)
LD MOA1.A
ST MOA_A

Function call in ST

MOA1(E := MOA_E, T := T#200ms);
MOA_A := MOA1.A;
A FALSE/TRUE edge at input E causes a FALSE/TRUE edge at output A. If input E remains at TRUE level, a TRUE/FALSE edge is applied at output A after the duration T has elapsed.

A possible second FALSE/TRUE edge of input E which occurs before the time period T has elapsed is ignored.

Maximum time offset at the output: < 1 cycle time

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>MOK</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>BOOL</td>
<td>Input signal</td>
</tr>
<tr>
<td>T</td>
<td>TIMER</td>
<td>Pulse length</td>
</tr>
<tr>
<td>A</td>
<td>BOOL</td>
<td>Output signal</td>
</tr>
</tbody>
</table>

### Description

A FALSE/TRUE edge at input E causes a FALSE/TRUE edge at output A. If input E remains at TRUE level, a TRUE/FALSE edge is applied at output A after the duration T has elapsed.

A possible second FALSE/TRUE edge of input E which occurs before the time period T has elapsed is ignored.

Maximum time offset at the output: < 1 cycle time

Reasonable range for T: > 1 cycle time

The inputs and the output can neither be duplicated nor inverted.

### General behavior

Started timers are processed by the PLC’s operating system and are therefore completely independent of PLC program processing. The operating system does not issue a corresponding message to the appropriate timer block in the PLC program until the timer has elapsed.

### Function call in IL

```il
CAL MOK1(E := MOK_E, T := T#200ms)
LD MOK1.A
ST MOK_A
```

### Function call in ST

```st
MOK1(E := MOK_E, T := T#200ms);
MOK_A:=MOK1.A;
```
MULTIPLICATION BY 2 TO THE POWER OF N, WORD

The value of the operand at input E1 is shifted bitwise N times.

If the value at input N is positive, the value is shifted to the left. Each shift by 1 bit position corresponds to a multiplication of the current value by 2.

If the value at input N is negative, the value is shifted to the right. Each shift by 1 bit position corresponds to a division of the current value by 2.

The result is assigned to the operand at output A1.

**Block type**

**Parameters**

<table>
<thead>
<tr>
<th>E1</th>
<th>INT</th>
<th>Input operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>INT</td>
<td>Quantity</td>
</tr>
<tr>
<td>A1</td>
<td>INT</td>
<td>Result</td>
</tr>
</tbody>
</table>

**Description**

The value of the operand at input E1 is shifted bitwise N times.

If the value at input N is positive, the value is shifted to the left. Each shift by 1 bit position corresponds to a multiplication of the current value by 2.

If the value at input N is negative, the value is shifted to the right. Each shift by 1 bit position corresponds to a division of the current value by 2.

The result is assigned to the operand at output A1.

The inputs and the output can neither be duplicated nor negated.

Reasonable range for N: -14 ≤ N ≤ +14

If N = 0, the value at input E1 is passed directly to output A1.

**Sign of the value at input E1:**

The sign of value E1 is not influenced by the shift operation. I.e. the sign of the output value is always identical with the sign of the input value.

**Shift to the left (Multiplication):**

When the value at the input is shifted to the left, the released bit 0 is filled with 0. The sign bit (bit 15) is not changed because a limiting to the limit of the number range is performed before.

Limiting the value at output A1 when shifting to the left:

- The following applies to positive values at input E1:
  - If bit 14 has a »1« and if shift operations still have to be carried out on the basis of the value at input N, these are no longer executed. Instead, the output is set to the limit of the positive number range. I.e. the limit has been reached in any case at the latest after 14 shifts.

  Limit value: Output A1 = +32767 (7FFFH).

- The following applies to negative values at input E1:
  - If bit 14 has a »0« and if shift operations still have to be carried out on the basis of the value at input N, these are no longer executed. Instead, the output is set to the limit of the negative number range. I.e. the limit has been reached in any case at the latest after 14 shifts.


**Shift to the right (Division):**

When shifting to the right, every bit moves to the right by one position. At the same time, the sign bit (bit 15) always retains its value. The released bit (bit 14) is filled in each case with the value of the sign bit.

Limiting the value at the output when shifting to the right:
The following applies to positive values at input E1:
If now only bit 0 has a »1« and shift operations still have to be carried out because of the value at input N, the output will be set to the value 0. I.e. value 0 has been reached in any case at the latest after 14 shifts.
Output A1 = 0.

The following applies to negative values at input E1:
If bit 0 ... bit 15 has a »1« as the result of the shift, the limit value (-1) has been reached. Further shifts have no effect. I.e. the value -1 has been reached at the latest after 15 shifts.
Output A1 = -1 (FFFFFFH).

The inputs and the output can neither be duplicated nor negated.

Function call in IL
LD MUL2N_E1
MUL2N MUL2N_N
ST MUL2N_A

Function call in ST
MUL2N_A:=MUL2N(MUL2N_E1, MUL2N_N);
MULTIPLICATION BY 2 TO THE POWER OF N, DOUBLE WORD

The value of the operand at input E1 is shifted bitwise N times.

If the value at input N is positive, the value is shifted to the left. Each shift by 1 bit position corresponds to a multiplication of the current value by 2.

If the value at input N is negative, the value is shifted to the right. Each shift by 1 bit position corresponds to a division of the current value by 2.

The result is assigned to the operand at output A1.

### Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Input operand</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Quantity</td>
</tr>
<tr>
<td>A1</td>
<td>DINT</td>
<td>Result</td>
</tr>
</tbody>
</table>

### Description

The value of the operand at input E1 is shifted bit-wise N times.

If the value at input N is positive, the value is shifted to the left. Each shift by 1 bit position corresponds to a multiplication of the current value by 2.

If the value at input N is negative, the value is shifted to the right. Each shift by 1 bit position corresponds to a division of the current value by 2.

The result is assigned to the operand at output A1.

The inputs and the output can neither be duplicated nor negated.

Reasonable range for N: \(-30 \leq N \leq +30\)

If N = 0, the value at input E1 is passed directly to output A1.

#### Sign of the value at input E1:

The sign of value E1 is not influenced by the shift operation. I.e. the sign of the output value is always identical with the sign of the input value.

#### Shift to the left (Multiplication):

When the value at the input is shifted to the left, the released bit 0 is filled with 0. The sign bit (bit 31) is not changed because a limiting to the limit of the number range is performed before.

#### Limiting the value at output A1 when shifting to the left:

- The following applies to positive values at input E1:
  - If bit 30 has a »1« and if shift operations still have to be carried out on the basis of the value at input N, these are no longer executed. Instead, the output is set to the limit of the positive number range. I.e. the limit has been reached in any case at the latest after 30 shifts.
  - Limit value: Output A1 = +2147483647 (7FFFFFFFH).

- The following applies to negative values at input E1:
  - If bit 30 has a »0« and if shift operations still have to be carried out on the basis of the value at input N, these are no longer executed. Instead, the output is set to the limit of the negative number range. I.e. the limit has been reached in any case at the latest after 30 shifts.

#### Shift to the right (Division):

When shifting to the right, every bit moves to the right by one position. At the same time, the sign bit (bit 31) always retains its value. The released bit (bit 30) is filled in each case with the value of the sign bit.
Limiting the value at the output when shifting to the right:

- The following applies to positive values at input E1: If now only bit 0 has a »1« and shift operations still have to be carried out because of the value at input N, the output will be set to the value 0. I.e. value 0 has been reached in any case at the latest after 30 shifts.
  
  Output A1 = 0.

- The following applies to negative values at input E1: If bit 0 ... bit 31 has a »1« as the result of the shift, the limit value (-1) has been reached. Further shifts have no effect. I.e. the value -1 has been reached at the latest after 31 shifts.
  
  Output A1 = -1 (FFFFFFFFH).

The inputs and the output can neither be duplicated nor negated.

### Examples:

1. Input value E1 = 58350926 (37A5D4Eh)  
   Exponent N = 4 \rightarrow 4 \times \text{Left shift}

   - 37A5D4EH
   - 0111 0111 1010 0101 1101 0100 1110 0000
   - Output A1 = 0

2. Input value E1 = 326786382 (137A5D4Eh)  
   Exponent N = -4 \rightarrow 4 \times \text{Right shift}

   - 137A5D4EH
   - 0001 0011 0111 1010 0101 1101 1110 0000
   - Output A1 = -1 (FFFFFFFFH)

3. Input value E1 = -326786382 (EC85A2B2h)  
   Exponent N = -4 \rightarrow 4 \times \text{Right shift}

   - EC85A2B2H
   - 1111 1100 1000 0101 1010 0010 1010 0010
   - FEC85A2BH
   - Output A1 = -1 (20424149)

### Function call in IL

- LD MUL2ND_E1
- MUL2ND MUL2ND_N
- ST MUL2ND_A

### Function call in ST

- MUL2ND_A:=MUL2ND(MUL2ND_E1, MUL2ND_N);
MULTIPLICATION, DOUBLE WORD

The value of the operand at input E1 is multiplied by the value of the operand at input E2 and the result is assigned to the operand at output A.

The result is limited to the maximum or minimum value of the number range. If limiting occurred, a TRUE signal is assigned to the binary operand at output Q. If no limiting has occurred, a FALSE signal is assigned to the binary operand at output Q.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>MULD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Multiplicand</td>
</tr>
<tr>
<td>E2</td>
<td>DINT</td>
<td>Multiplier</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Result (product)</td>
</tr>
<tr>
<td>Q</td>
<td>BOOL</td>
<td>Result limited</td>
</tr>
</tbody>
</table>

Description

The value of the operand at input E1 is multiplied by the value of the operand at input E2 and the result is assigned to the operand at output A.

The result is limited to the maximum or minimum value of the number range (Number range: -2147483647 … 2147483647). If limiting occurred, a TRUE signal is assigned to the binary operand at output Q. If no limiting occurred, a FALSE signal is assigned to the binary operand at output Q.

The inputs and outputs can neither be duplicated nor negated.

Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>MULD1(E1 := MULD_E1, E2 := MULD_E2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>MULD1.Q</td>
</tr>
<tr>
<td>ST</td>
<td>MULD.Q</td>
</tr>
<tr>
<td>LD</td>
<td>MULD1.A</td>
</tr>
<tr>
<td>ST</td>
<td>MULD.A</td>
</tr>
</tbody>
</table>

Function call in ST

MULD1(E1 := MULD_E1, E2 := MULD_E2);
MULD.Q:=MULD1.Q;
MULD.A:=MULD1.A;
MULTIPLICATION WITH DIVISION

The operand value at input E1 is multiplied by the operand value at input E2, the intermediate result is divided by the operand value at E3 and then the result is assigned to output A.

The result is limited to the maximum or minimum value of the number range.

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Multiplicand</td>
</tr>
<tr>
<td>E2</td>
<td>INT</td>
<td>Multiplier</td>
</tr>
<tr>
<td>E3</td>
<td>INT</td>
<td>Divisor</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Result</td>
</tr>
</tbody>
</table>

**Description**

The operand value at input E1 is multiplied by the operand value at input E2, the intermediate result is divided by the operand value at E3 and then the result is assigned to output A.

Internal, the block performs the multiplication and division with the accuracy of a double word (32 bit). Only when assigning the result to output A, the limiting to the accuracy of a word (16 bit) is carried out. If the remainder of the division is > 0.5, the result is rounded up. If a numerical overflow occurs during the division (e.g. division by zero), the correct signed limit of the number range is applied at output A.

The result is limited to the maximum value 32767 and the minimum value -32767.

The inputs and the output can neither be duplicated nor negated.

**Function call in IL**

```
LD MULDI_E1
MULDI MULDI_E2, MULDI_E3
ST MULDI_A
```

**Function call in ST**

```
MULDI_A := MULDI(MULDI_E1, MULDI_E2, MULDI_E3);
```
MULTIPLICATION, WORD

The values of the operands at the inputs of the function are multiplied and the result is assigned to the operand at the output.

The result is limited to the maximum or minimum value of the number range.

Block type

Function

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Multiplicand</td>
</tr>
<tr>
<td>E2</td>
<td>INT</td>
<td>Multiplier</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Result (product)</td>
</tr>
</tbody>
</table>

Description

The values of the operands at the inputs of the function are multiplied and the result is assigned to the operand at the output.

The result is limited to the maximum value 32767 and the minimum value -32767.

Function call in IL

LD MULW_E1
MULW MULW_E2,MULW_E3
ST MULW_A

Function call in ST

MULW_A := MULW(MULW_E1, MULW_E2);
MULTIPLEXER WITH RESET, WORD

This block connects input E to one of the outputs A0 ... An-1 depending on input INDEX.

The MUXR number indicates the maximum number of word outputs. The following multiplexers are available:

- MUXR8 Multiplexer with max. 8 word outputs
- MUXR16 Multiplexer with max. 16 word outputs
- MUXR32 Multiplexer with max. 32 word outputs
- MUXR64 Multiplexer with max. 64 word outputs
- MUXR256 Multiplexer with max. 256 word outputs

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>MUXR(8..256)</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>INT</td>
<td>Input</td>
</tr>
<tr>
<td>INDEX</td>
<td>INT</td>
<td>Index input</td>
</tr>
<tr>
<td>n</td>
<td>INT</td>
<td>Quantity (n) of word outputs A0 ... An-1</td>
</tr>
<tr>
<td>INOK</td>
<td>BOOL</td>
<td>Range monitoring for input INDEX</td>
</tr>
<tr>
<td>A0..An-1</td>
<td>INT</td>
<td>Word outputs A0 ... An-1</td>
</tr>
</tbody>
</table>

Description

This block connects input E to one of the outputs A0 ... An-1 depending on input INDEX.

The word outputs which are not connected are set to 0.

The value at input INDEX is monitored for validity.

The input INDEX is used to define with which of the outputs A0...An-1 the input E is connected.

The following applies:

\[
\text{INDEX} = 1: \quad E \rightarrow A0 \\
\text{INDEX} = 2: \quad E \rightarrow A1 \\
\text{INDEX} = 3: \quad E \rightarrow A2 \\
\vdots \quad \vdots \quad \vdots \\
\text{INDEX} = n: \quad E \rightarrow An-1
\]

with \(1 < \text{INDEX} < n < \text{MUXR number (8..256)}\)

Note:

INDEX = 0 can be used to initialize the outputs A0 ... An-1 (A0 ... An-1 = 0).

\[
n \quad \text{INT}
\]

Quantity (n) of word outputs A0 ... An-1.

\[
\text{INOK} \quad \text{BOOL}
\]

Range monitoring of the INDEX input.

The output indicates whether input INDEX is within the valid range or not.

Valid range: \(1 < \text{INDEX} < n\)

\[
\text{INOK} = \text{TRUE} \quad \rightarrow \text{Index input INDEX within the valid range}
\]

\[
\text{INOK} = \text{FALSE} \quad \rightarrow \text{Index input INDEX in the invalid range}
\]

\[\rightarrow A0 \ldots An-1 = 0\]

All word outputs A0 ... An-1 are set to 0 if the word input INDEX is not within the valid range. Thus, for example, INDEX = FALSE can be used to initialize the outputs (A0 ... An-1 = 0).
A0..An-1 \( \text{INT} \)

The output A0 can be duplicated (A0 ... An-1). The input E is assigned to one of the n outputs A0 ... An-1.

**Function call in IL**

```plaintext
CAL MUXR81(E := MUXR_E, INDEX := MUXR_IND, n := 3)
LD MUXR81.A0
ST MUXR_A0
LD MUXR81.A1
ST MUXR_A1
LD MUXR81.A2
ST MUXR_A2
LD MUXR81.INOK
ST MUXR_INOK
```

**Function call in ST**

```plaintext
MUXR81(E := MUXR_E, INDEX := MUXR_IND, n := 3);
MUXR_A0:=MUXR81.A0;
MUXR_A1:=MUXR81.A1;
MUXR_A2:=MUXR81.A2;
MUXR_INOK:=MUXR81.INOK;
```

Note: In IL, the function call has to be performed in one line.
MULTIPLEXER WITH RESET, DOUBLE WORD

This block connects input E to one of the outputs A0 ... An-1 depending on input INDEX.

The MUXRD number indicates the maximum number of word outputs. The following multiplexers are available:

- MUXRD8  Multiplexer with max. 8 double word outputs
- MUXRD16 Multiplexer with max. 16 double word outputs
- MUXRD32 Multiplexer with max. 32 double word outputs
- MUXRD64 Multiplexer with max. 64 double word outputs
- MUXRD256 Multiplexer with max. 256 double word outputs

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Instance name</th>
<th>E</th>
<th>DINT</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>INT</td>
<td>n</td>
<td>INT</td>
<td>Quantity (n) of word outputs A0 ... An-1</td>
</tr>
<tr>
<td>INOK</td>
<td>BOOL</td>
<td>A0..An-1</td>
<td>DINT</td>
<td>Double word outputs A0 ... An-1</td>
</tr>
</tbody>
</table>

Description

This block connects input E to one of the outputs A0 ... An-1 depending on input INDEX.

The word outputs which are not connected are set to 0. The value at input INDEX is monitored for validity.

Relationship between E, INDEX and A0...An-1:

The input INDEX is used to define with which of the outputs A0...An-1 the input E is connected.

The following applies:

INDEX = 1:  E \rightarrow A0
INDEX = 2:  E \rightarrow A1
INDEX = 3:  E \rightarrow A2
... 
INDEX = n:  E \rightarrow An-1

with 1 < INDEX < n < MUXR number (8..256)

E          | DINT
------------|------
Input which is switched through to one of the outputs A0...An-1.

INDEX      | INT
------------|------
Index input for selecting one of the outputs A0 ... An-1.

Value range:
1 < INDEX < n < MUXR number (8..256)

Note:
INDEX = 0 can be used to initialize the outputs A0 ... An-1 (A0 ... An-1 = 0).

n          | INT
------------|------
Quantity (n) of word outputs A0 ... An-1.
**INOK**

**BOOL**

Range monitoring of the INDEX input.

The output indicates whether input INDEX is within the valid range or not.

Valid range: $1 < INDEX < n$

INOK = TRUE  
→ Index input INDEX within the valid range

INOK = FALSE  
→ Index input INDEX in the invalid range

All word outputs $A0 \ldots An-1$ are set to 0 if the word input INDEX is not within the valid range. Thus, for example, INDEX = 0 can be used to initialize the outputs ($A0 \ldots An-1 = 0$).

**A0..An-1**  
**DINT**

The output $A0$ can be duplicated ($A0 \ldots An-1$). The input $E$ is assigned to one of the $n$ outputs $A0 \ldots An$.

**Function call in IL**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL MUXRD81(E := MUXRD_E, INDEX := MUXRD_IND, n := 3)</td>
<td>Function call</td>
</tr>
<tr>
<td>LD MUXRD81.A0</td>
<td>Load word output</td>
</tr>
<tr>
<td>ST MUXRD_A0</td>
<td>Store to word</td>
</tr>
<tr>
<td>LD MUXRD81.A1</td>
<td>Load word output</td>
</tr>
<tr>
<td>ST MUXRD_A1</td>
<td>Store to word</td>
</tr>
<tr>
<td>LD MUXRD81.A2</td>
<td>Load word output</td>
</tr>
<tr>
<td>ST MUXRD_A2</td>
<td>Store to word</td>
</tr>
<tr>
<td>LD MUXRD81.INOK</td>
<td>Load index validity</td>
</tr>
<tr>
<td>ST MUXRD_INOK</td>
<td>Store to index validity</td>
</tr>
</tbody>
</table>

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```plaintext
MUXRD81(E := MUXRD_E, INDEX := MUXRD_IND, n := 3);
MUXRD_A0 := MUXRD81.A0;
MUXRD_A1 := MUXRD81.A1;
MUXRD_A2 := MUXRD81.A2;
MUXRD_INOK := MUXRD81.INOK;
```
**NEGATION, DOUBLE WORD**

The value of the operand at input E is negated and the result is assigned to the operand at output A.

Block type

Function

**Parameters**

<table>
<thead>
<tr>
<th>E</th>
<th>DINT</th>
<th>Input value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>DINT</td>
<td>Negated value</td>
</tr>
</tbody>
</table>

**Description**

The value of the operand at input E is negated and the result is assigned to the operand at output A.

The result is limited to the maximum or minimum value of the number range. (Number range: -2147483647 ... 2147483647).

The input and the output can neither be duplicated nor negated.

**Function call in IL**

LD NEG_E1
NEGD
ST NEG_A

**Function call in ST**

NEG_A:=NEGD(NEG_E1);
The value of the operand at input E is negated and the result is assigned to the operand at output A.

Block type
Function

Parameters
E  INT  Input value
A  INT  Negated value

Description
The value of the operand at input E is negated and the result is assigned to the operand at output A.
The result is limited to the maximum or minimum value of the number range. (Number range: -2147483647 … 2147483647).
The input and the output can neither be duplicated nor negated.

Function call in IL
LD  NEGW_E1
NEGW
ST  NEGW_A

Function call in ST
NEGW_A:=NEGW(NEGW_E1);
This block packs \( n \) binary variables into one word variable.

### Block type

Function block without historical values

### Parameters

- **Instance** | **PACK** | Instance name
- **n** | **INT** | Number of binary variables to be packed
- **B0...B15** | **BOOL** | Binary variable to be packed
- **WORD** | **INT** | Word variable

### Description

This block packs \( n \) binary variables into one word variable.

\[
\begin{align*}
\text{At input } n, \text{ the number of binary variables to be packed is specified.} \\
\text{The following applies:} \\
1 \leq n \leq 16 \\
n = 0 \text{ is not allowed! (default } = 1) \\
\end{align*}
\]

The binary variables to be packed are specified at the inputs \( B0 \ldots Bn-1 \).

The value of each binary variable at the inputs \( B0 \ldots Bn-1 \) is loaded into the corresponding bit (bit 0 ... bit 15) of the variable at output \( \text{WORT} \).

Assignment:

\[
\begin{align*}
B0 & \rightarrow \text{bit}0 \text{ of the output variable} \\
B1 & \rightarrow \text{bit}1 \text{ of the output variable} \\
\ldots
\end{align*}
\]

\[
\begin{align*}
B15 & \rightarrow \text{bit}15 \text{ of the output variable}
\end{align*}
\]

Note:

If the user plans less than 16 binary input variables, the unnecessary bits of the output variable are set to the value 0.
Function call in IL

CAL PACK1(n := 3, B0 := PACK_B0, B1 := PACK_B1, B2 := PACK_B2)
LD PACK1.A
ST PACK_A

Note: In IL, the function call has to be performed in one line.

Function call in ST

PACK1(n := 3, B0 := PACK_B0, B1 := PACK_B1, B2 := PACK_B2);
PACK_A:=PACK1.A;
PACK BINARY VARIABLES IN DOUBLE WORD

This block packs n binary variables into one double word variable.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>PACKD Instance name</td>
</tr>
<tr>
<td>n</td>
<td>INT Number of binary variables to be packed</td>
</tr>
<tr>
<td>B0..B15</td>
<td>BOOL Binary variable to be packed</td>
</tr>
<tr>
<td>A</td>
<td>DINT Double word variable</td>
</tr>
</tbody>
</table>

Description
This block packs n binary variables into one double word variable.

n INT
At input n, the number of binary variables to be packed is specified.
The following applies:
1 < n < 32
n = 0 is not allowed! (default = 1)

B0...B31 BOOL
The binary variables to be packed are specified at the inputs B0 ... Bn-1.

A DINT
The value of each binary variable at the inputs B0 ... Bn-1 is loaded into the corresponding bit (bit 0 ... bit 13) of the variable at output A.

Assignment:
B0 → bit0 of the output variable
B1 → bit1 of the output variable
...
B31 → bit31 of the output variable

Note:
If the user plans less than 31 binary input variables, the unnecessary bits of the output variable are set to the value 0.
Function call in IL

CAL PACKD1(n := 3, B0 := PACKD_B0, B1 := PACKD_B1, B2 := PACKD_B2)
LD PACKD1.A
ST PACKD_A

Note: In IL, the function call has to be performed in one line.

Function call in ST

PACKD1(n := 3, B0 := PACKD_B0, B1 := PACKD_B1, B2 := PACKD_B2);
PACKD_A:=PACKD1.A;
This function block generates a pulse duration-modulated binary signal at its PULS output.

The pulse duty ratio is specified at the t_TA input and the period duration for the output signal is specified at the TA_T input.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>PDM</td>
<td>Instance name</td>
</tr>
<tr>
<td>t_TA</td>
<td>INT</td>
<td>Pulse duty ratio</td>
</tr>
<tr>
<td>TA_T</td>
<td>INT</td>
<td>Period duration referred to the cycle time</td>
</tr>
<tr>
<td>PULS</td>
<td>BOOL</td>
<td>Pulse duration modulated signal</td>
</tr>
</tbody>
</table>

The following relationship applies to the specification of the pulse duty ratio at input t_TA:

<table>
<thead>
<tr>
<th>Input t_TA</th>
<th>Output PULS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative value</td>
<td>0 (0 %)</td>
</tr>
<tr>
<td>0 (0 * 32767)</td>
<td>0 (0 %)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16384 (0,5 * 32767)</td>
<td>0,5 (50 %)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>32767 (1 * 32767)</td>
<td>1 (100 %)</td>
</tr>
</tbody>
</table>

The required period duration TA for the signal at output PULS is specified at input TA_T. At the same time, the period duration TA must be scaled to the cycle time T. The required period duration must be an integer multiple of T (TA = n * T). The higher TA is in relation to T, the more exactly the required pulse duty ratio is kept.

Example:

TA > 10 * T → inaccuracy of the pulse duty ratio at output PULS < 10%.

If a value TA_T < 0 is specified for TA_T, the function block automatically replaces this meaningless value by 32767.
The pulse duration modulated signal is available at output PULS.

Combination of the PDM block with a controller

If the PDM function block is connected to the output of a controller in order to realize a »switching« controller, the following boundary conditions apply:

– Period duration $T_A$ of the PDM = sampling time of the controller.

– Period of the pulse signal must be synchronous with the period of the controller sampling time.

These boundary conditions are fulfilled by planning the controller in the same PLC program as the PDM, but within one run number block. By means of the run number block, the sampling time of the controller is prolonged by an integer multiple of the cycle time. Therefore, the controller is processed less frequently within the run number block than the PDM outside of the run number block.

**Example**

Required:

– Pulse duty ratio: $t_{TA} = 0.25$ (25 %)

– Period duration: $T_A = 800$ ms (only an integer multiple of the PLC cycle time is possible)

– Cycle time: $T = 100$ ms

Block parameters to be specified:

– Value at input $t_{TA}$: 8192 (0.25 * 32767)

– Value at input $T_A_T$: 8 (800 ms/100 ms)

**Function call in IL**

```
CAL PDM1(t_TA := PDM_TTA, TA_T := PDM_TAT)
LD PDM1.PULS
ST PDM_PULS
```

**Note:** In IL, the function call has to be performed in one line.

**Function call in ST**

```
PDM1(t_TA := PDM_TTA, TA_T := PDM_TAT);
PDM_PULS := PDM1.PULS;
```
The PI controller changes the value at its output $Y$ (manipulated variable) until the value at input $X$ (controlled variable) is equal to the value at input $W$ (command variable).

Control algorithm: Simple rectangle rule

$$Y = \frac{KP}{100} \cdot \frac{W-X}{TN_{TZ}} + \frac{KP}{100} \cdot (W-X)$$

With: $Yl(z-1)$ is the integral component from the previous program cycle

Transfer function:

$$F(s) = KP \cdot \left(1 + \frac{1}{s \cdot TN}\right)$$

The inputs and outputs can neither be duplicated nor negated/inverted.
**PI controller**: Surge-free transition from the specified value to control operation

**PI controller**: Surging transition from the specified initial value to control operation
The command variable (set point) is specified at input \( W \).

The controlled variable (actual value) is specified at input \( X \).

The proportional coefficient is specified at input \( KP \).

This value is specified as a percentage value. Positive or negative values are possible.

Example:
- \( 1 = 1 \) percent
- \( 55 = 55 \) percent
- \( 100 = 100 \) percent
- \( 1000 = 1000 \) percent
- \( -500 = -500 \) percent

\( 1 \) percent means that the block multiplies the system deviation by the factor 0.01 (see also control algorithm)

\( 100 \) percent means that the block multiplies the system deviation by the factor 1 (see also control algorithm)

\( 1000 \) percent means that the block multiplies the system deviation by the factor 10 (see also control algorithm)

Generally, proportional coefficients of more than 1000 percent are not meaningful in control systems.

The integral action time \( TN \) is scaled to the PLC cycle time \( TZ \) and is specified at input \( TN_TZ \).

Value range: \( 0 < TN_TZ < 328 \)

If values are specified which are not within the admissible value range the PLC generally uses the value 328.

A large integral action time \( TN \) can be achieved by choosing a large cycle time \( T \), too. If the block is used within a run number block, the cycle time of the run number block is valid for block \( INTK \) and not the cycle time (KD 0,0) of the PLC program.

The controller output \( Y \) (manipulated variable) can be limited

- to a maximum value by specifying a limit at input \( OG \) (high limit)

The high and low limits also apply to the controller’s internal \( I \) component. I.e. the \( I \) component can only assume values between the high and low limits. If the manipulated variable \( Y \) reaches one of the two limits, the \( I \) component of the controller is no longer changed. This prevents the \( I \) component from drifting in the event of limiting of the controller output \( Y \), assuming meaningless values from the point of view of control and, in certain circumstances, not returning to the operating range until after a very long time. This response of a controller is also referred to as a »special anti-reset windup measure«.

Setting and resetting the controller

- The output \( Y \) of the controller is set to the initial value specified at the \( INIT \) input by means of a TRUE signal at input \( SET \).
- A TRUE signal at input \( RES \) (reset) is equivalent to specifying the initial value 0 (see above).

Surge-free setting/resetting

- The output \( Y \) of the controller is set to the initial value specified at the \( INIT \) input by means of a TRUE signal at the binary input \( SET \).
- A TRUE signal at input \( RES \) (reset) is equivalent to specifying the initial value 0.

In doing so, adjustment to the initial value is performed internal to the controller. This adjustment is a shift of the controller output from the momentary value to the required initial value. Now, the controller continues operating from this initial value precisely as it would have done at the old operating point before the shift, i.e. without surges. The \( I \) component of the controller is defined so that the sum of the \( P \) component and the \( I \) component just results in the initial value.

Advantage of surge-free setting:
- Control as from the new initial value is devoid of surges.

Disadvantage of surge-free setting:
- The equation applies: \( I_{\text{component}} = INIT - P_{\text{component}} \)

In certain circumstances, the \( I \) component is set to high values and may take very long before this »wrong« \( I \) component from the point of view of control is dissipated again.
The output $Y$ of the controller is set to the initial value specified at the INIT input by means of a TRUE signal at input SET.

A TRUE signal at input RES (reset) is equivalent to specifying the initial value 0.

In the event of surging setting or resetting of the controller, the I component is set equal to the initial value. To do this, the P component must be suppressed during the setting procedure.

The following applies: I component = INIT

Surging setting to an initial value is achieved by the following measure during the setting procedure:

- Specifying the value 0 at input KP.

This measure renders the controller’s P component inactive. The controller output $Y$ assumes the initial value during the set cycle. The P component is enabled again after the set cycle.

From the initial value, the controller output $Y$ jumps according to the P component of the controller.

Advantage of surging setting:
- The I component is not set to »wrong« values from the point of view of control.

Disadvantage of surging setting:
- Not surging-free

### Function call in IL

```
CAL PI1(W := PI_W, X := PI_X, KP := PI_KP, 
      TN_TZ := PI_TNTZ, OG := PI_OG, 
      UG := PI_UG, SET := PI_SET, 
      INIT := PI_INIT, RES := PI_RES)
```

LD PI1.OG_MELD
ST PI_MOG
LD PI1.UG_MELD
ST PI_MUG
LD PI1.Y
ST PI_Y

Note: In IL, the function call has to be performed in one line.

### Function call in ST

```
PI1(W := PI_W, X := PI_X, KP := PI_KP, 
      TN_TZ := PI_TNTZ, OG := PI_OG, UG := PI_UG, 
      SET := PI_SET, INIT := PI_INIT, RES := PI_RES);

PI_MOG:=PI1.OG_MELD;
PI_MUG:=PI1.UG_MELD;
PI_Y:=PI1.Y;
```

$Y$ INT

The manipulated variable $Y$ of the controller is applied at output $Y$.

OG_MELD BOOL

Output OG_MELD indicates whether the value at output $Y$ has reached the specified top limit.

OG_MELD = FALSE limit has not been reached.
OG_MELD = TRUE limit has been reached.

UG_MELD BOOL

Output UG_MELD indicates whether the value at output $Y$ has reached the specified low limit.

UG_MELD = FALSE limit has not been reached.
UG_MELD = TRUE limit has been reached.
**PIDT1 CONTROLLER**

The PI controller changes its output \( Y \) (manipulated variable) until input \( X \) (controlled variable) is equal to input \( W \) (command variable).

---

**Block type**

Function block with historical values

---

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIDT1</td>
<td>Instance name</td>
<td>( W ) Command variable (set point)</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Controlled variable (actual value)</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Proportional coefficient, specified as a percentage value</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Integral action time scaled to the PLC cycle time</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Derivative action time scaled to the PLC cycle time</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Returning time scaled to the PLC cycle time</td>
</tr>
<tr>
<td></td>
<td>BOOL</td>
<td>Enable DT component</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>High limit for the manipulated variable ( Y )</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Low limit for the manipulated variable ( Y )</td>
</tr>
<tr>
<td></td>
<td>BOOL</td>
<td>Enable for setting to initial value INIT</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Initial value for the manipulated variable ( Y )</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Reset of the manipulated variable ( Y ) to 0</td>
</tr>
<tr>
<td></td>
<td>BOOL</td>
<td>High limit has been reached</td>
</tr>
<tr>
<td></td>
<td>BOOL</td>
<td>Low limit has been reached</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Output for the manipulated variable ( Y )</td>
</tr>
</tbody>
</table>

**Description**

The PI controller changes its output \( Y \) (manipulated variable) until input \( X \) (controlled variable) is equal to input \( W \) (command variable).

Transfer function:

\[
F(s) = KP \times \left(1 + \frac{1}{s \times TN} + \frac{s \times TV}{1 + (s \times T1)}\right)
\]

Control algorithm: Simple rectangle rule:

\[
Y = \frac{KP \times XD}{100} + \frac{XD}{100} \times \frac{YI(z-1)}{TN_TZ} + \frac{T1_TZ}{1+(T1_TZ)} \times (YDTI(z-1) + \frac{TV}{T1_TZ} + \frac{TP}{TZ} + \frac{100}{100} \times (XD - XD(z-1)))
\]

Where:

- \( YI(z-1) \): The integral portion from the previous program cycle
- \( YDTI(z-1) \): The differential portion from the previous program cycle
- \( XD(z-1) \): Control system difference from the previous program cycle

---

Base_S90 Library 132

907 AC 1131/Stand: 09.03
The inputs and outputs can neither be duplicated nor negated/inverted.

**PIDT1 controller:** Surge-free transition from the specified value to control operation

**PIDT1 controller:** Surging transition from the specified initial value to control operation
The command variable (set point) is specified at input W.

The controlled variable (actual value) is specified at input X.

The proportional coefficient is specified at input KP. This value is specified as a percentage value. Positive or negative values are possible.

Example:

- 1 = 1 percent
- 55 = 55 percent
- 100 = 100 percent
- 1000 = 1000 percent
- -500 = -500 percent

1 percent means that the block multiplies the system deviation by the factor 0.01 (see also control algorithm).

100 percent means that the block multiplies the system deviation by the factor 1 (see also control algorithm).

1000 percent means that the block multiplies the system deviation by the factor 10 (see also control algorithm).

Generally, proportional coefficients of more than 1000 percent are not meaningful in control systems.

The integral action time TN is scaled to the PLC cycle time TZ and is specified at input TN_TZ.

Value range: 0 ≤ TN_TZ ≤ 328

If values are specified which are not within the admissible value range, the PLC generally uses the value 328.

A large integral action time TN can be achieved by choosing a large cycle time T, too. If the block is used within a run number block, the cycle time of the run number block is valid for block INTK and not the cycle time (KD 0,0) of the PLC program.

The derivative action time TV is scaled to the PLC cycle time TZ and is specified at input TV_TZ: (0 ≤ TN_TZ ≤ 32767).

The returning time T1 is scaled to the PLC cycle time TZ and is specified at input T1_TZ: (0 ≤ TN_TZ ≤ 32767).

The returning time is the time in which the DT1 component has decreased to approximately 37% of its initial value.

Invalid time parameters:
Every time value is set to the maximum positive value 32767 if the time value at the input is erroneously specified as less than or equal to «0».

The DT1 component of the controller can be activated or deactivated by means of the D-FR input.

In the following cases, from the control point of view it is often disturbing and not meaningful for the DT1 component to be active:
- During activations
- In the event of large system deviations
- When setting the controller to a specified initial value
- When resetting the controller to the value 0

The command and controlled variables can be compared outside of the controller. Depending on this comparison, the DT1 component can be activated or deactivated specifically using the D-FR input.

For example, the activation can be restricted to the condition that the system deviation is within a required bandwidth. i.e. the DT1 component is only active if the controlled variable fluctuates around the set point within a specific bandwidth. The DT1 component is deactivated if the controlled variable leaves this tolerance band.
Limiting the manipulated variable Y

OG, UG INT

The controller output Y (manipulated variable) can be limited

- to a maximum value by specifying a limit at input OG (high limit)
- to a minimum value by specifying a limit at input UG (low limit).

The high and low limits also apply to the controller’s internal I component. I.e. the I component can only assume values between the high and low limits. If the manipulated variable Y reaches one of the two limits, the controller’s I component is no longer changed.

This prevents the I component from drifting in the event of limiting of the controller output Y, assuming meaningless values from the point of view of control and, in certain circumstances, not returning to the operating range until after a very long time. This response of a controller is also referred to as a »special anti-reset windup measure«.

Setting and resetting the controller

SET BOOL
INIT INT
RES BOOL

Setting the controller to an initial value

- The output Y of the controller is set to the initial value specified at the INIT input by means of a TRUE signal at input SET.
- A 1 signal at input RES (reset) is equivalent to specifying the initial value 0 (see above).

Surge-free setting/resetting

- Output Y of the controller is set to the initial value specified at the INIT input by means of a TRUE signal at the binary input SET.
- A 1 signal at input RES (reset) is equivalent to specifying the initial value 0.

In doing so, adjustment to the initial value is performed internal to the controller. This adjustment is a shift of the controller output from the momentary value to the required initial value. Now, the controller continues operating from this initial value precisely as it would have done at the old operating point before the shift, i.e. without surges. The controller’s I component is defined in a way that the sum of the P, I and the DT1 component just results in the initial value.

Advantage of surge-free setting:
- Control as from the new initial value is devoid of surges.

Disadvantage of surge-free setting:
- The equation applies:
  \[ \text{I component} = \text{INIT} - \text{P component} - \text{DT1 comp.} \]

In certain circumstances, the I component is set to high values and may take very long before this »wrong« I component from the point of view of control is dissipated again.

Surging setting/resetting

- The output Y of the controller is set to the initial value specified at the INIT input by means of a TRUE signal at input SET.
- A 1 signal at input RES (reset) is equivalent to specifying the initial value 0.
- In the event of surging setting or resetting of the controller the I component is set equal to the initial value. To do this, the P component and the DT1 component must be suppressed during setting procedure.

The following applies: \[ \text{I component} = \text{INIT} \]

Surging setting to an initial value is achieved by the following measures during setting procedure:

- Deactivation of the DT1 component via the D-FR control input and
- Specifying the value 0 at the input KP.

These measures render the P component and the DT1 component inactive during setting of the controller.

The controller output assumes the initial value during the set cycle.

The P component and the DT1 component are enabled again after the set cycle. From the initial value, the controller output Y jumps according to the P component and the DT1 component of the controller.

Advantage of surging setting:
- The I component is not set to »wrong« values from the point of view of control.

Disadvantage of surging setting:
- Not surge-free

Y INT

The manipulated variable Y of the controller is applied at output Y.

OG_MELD BOOL

Output OG_MELD indicates whether the value at output Y has reached the specified top limit.

OG_MELD = FALSE limit has not been reached.
OG_MELD = TRUE limit has been reached.
UG_MELD

BOOL

Output UG_MELD indicates whether the value at output Y has reached the specified low limit.

UG_MELD = FALSE  limit has not been reached.
UG_MELD = TRUE   limit has been reached.

Function call in IL

CAL PIDT11(W := PIDT_W, X := PIDT_X,
       KP := PIDT_KP, TN_TZ := PIDT_TNTN,
       TV_TZ := PIDT_TVTZ,
       T1_TZ := PIDT_T1TZ,
       DT1_FR := PIDT_DTFR, OG := PIDT_OG,
       UG := PIDT_UG, SET := PIDT_SET,
       INIT := PIDT_INIT, RES := PIDT_RES)

LD PIDT11.OG_MELD
ST PIDT_MOG
LD PIDT11.UG_MELD
ST PIDT_MUG
LD PIDT11.Y
ST PIDT_Y

Note: In IL, the function call has to be performed in one line.

Function call in ST

PIDT11(W := PIDT_W, X := PIDT_X,
       KP := PIDT_KP, TN_TZ := PIDT_TNTN,
       TV_TZ := PIDT_TVTZ, T1_TZ := PIDT_T1TZ,
       DT1_FR := PIDT_DTFR, OG := PIDT_OG,
       UG := PIDT_UG, SET := PIDT_SET,
       INIT := PIDT_INIT, RES := PIDT_RES);

PIDT_MOG:=PIDT11.OG_MELD;
PIDT_MUG:=PIDT11.UG_MELD;
PIDT_Y:=PIDT11.Y;
This function block realizes a delay element of the first order.

**Transfer function:**

\[
F = \frac{1}{1 + T_1 \cdot s}
\]

The inputs and the output can neither be duplicated nor negated.
The controlled variable is specified at input X.

The delay time T1 is specified at input T1_TZ. To do this, the delay time T1 must be normalized to the cycle time TZ.

The following must apply: $T1_TZ > 0$

If an incorrect negative time value is specified, the PLC automatically sets $T1_TZ$ to the value 32767.

The result of the delay element (manipulated variable) is output at output Y.

Function call in IL

```
CAL PT11(X := PT1_X, T1_TZ := PT1_T1TZ)
LD PT11.Y
ST PT1_Y
```

Function call in ST

```
PT11(X := PT1_X, T1_TZ := PT1_T1TZ);
PT1_Y:=PT11.Y;
```
SET BIT IN DOUBLE WORD

The bit at input BIT is set or reset in the double word A depending on input E. Valid values at input BIT: 0…31.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>SETBD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>BOOL</td>
<td>Input value</td>
</tr>
<tr>
<td>BIT</td>
<td>INT</td>
<td>Bit number of the bit to be set</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Output value</td>
</tr>
</tbody>
</table>

Description

The bit at input BIT is set or reset in the double word A depending on input E. Valid values at input BIT: 0…31.

The inputs and the output can neither be duplicated nor negated.

Function call in IL

```
CAL SETBD1(E := SETD_E, BIT := SETD_BIT, A := SETD_A)
```

Function call in ST

```
SETBD1(E := SETD_E, BIT := SETD_BIT, A := SETD_A)
```

Note: In IL, the function call has to be performed in one line.
**SET BIT IN WORD**

The bit at input BIT is set or reset in the word A depending on input E. Valid values at input BIT: 0…15.

**Block type**

Function block without historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETBW</td>
<td>E</td>
<td>BOOL</td>
<td>Input value</td>
</tr>
<tr>
<td></td>
<td>BIT</td>
<td>INT</td>
<td>Bit number of the bit to be set</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>INT</td>
<td>Output value</td>
</tr>
</tbody>
</table>

**Description**

The bit at input BIT is set or reset in the word A depending on input E. Valid values at input BIT: 0…15.

The inputs and the output can neither be duplicated nor negated.

**Function call in IL**

```
CAL SETBW1(E := SETW_E, BIT := SETW_BIT, A := SETW_A)
```

**Function call in ST**

```
SETBW1(E := SETW_E, BIT := SETW_BIT, A := SETW_A)
```

Note: In IL, the function call has to be performed in one line.
SHIFT BLOCK, DOUBLE WORD

SHIFTD

This function block shifts the operand present at the input by a specified number of bit positions to the left or right.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>SHIFTD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>DINT</td>
<td>Operand to be shifted</td>
</tr>
<tr>
<td>ANZ</td>
<td>INT</td>
<td>Number of bit positions to be shifted</td>
</tr>
<tr>
<td>LKS</td>
<td>BOOL</td>
<td>Shift direction, left or right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LKS = FALSE → shift to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LKS = TRUE → shift to left</td>
</tr>
<tr>
<td>ROT</td>
<td>BOOL</td>
<td>Shift type: ROTATE</td>
</tr>
<tr>
<td>ROTC</td>
<td>BOOL</td>
<td>Shift type: ROTATE by the CARRY FLAG</td>
</tr>
<tr>
<td>SLOG</td>
<td>BOOL</td>
<td>Shift type: LOGICAL SHIFT</td>
</tr>
<tr>
<td>SARI</td>
<td>BOOL</td>
<td>Shift type: ARITHMETIC SHIFT</td>
</tr>
<tr>
<td>CY_E</td>
<td>BOOL</td>
<td>Initial value for the CARRY FLAG in case of shift type ROTC</td>
</tr>
<tr>
<td>CY_A</td>
<td>BOOL</td>
<td>CARRY FLAG state after the shift procedure</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Result of the shift procedure</td>
</tr>
</tbody>
</table>

Beschreibung

This function block shifts the operand present at the input by a specified number of bit positions to the left or right.

The result of the shift procedure and the CARRY flag are available at the block outputs. The operand at the input remains unchanged.

The required shift type is planned at the inputs:

- ROT
- ROTC
- SLOG
- SARI

If several shift types are specified simultaneously, the applying shift type is the one which is located furthest to the front in the sequence of the block inputs.

The block can only shift double words.

Caution:

In certain circumstances, the value 8000 0000H forbidden for arithmetic operations may be present at this block output.

The inputs and outputs can neither be duplicated nor inverted nor negated.
### SHIFT BLOCK, DOUBLE WORD

**E**
Operand to be shifted. The planned SHIFT operation is applied to the input operand. In doing so, the input operand is not changed.

**ANZ**
Number of bit positions to be shifted.
Sensible range for the quantity n: $0 < n < 32$

**LKS**
Direction in which shifting is performed.
$LKS = \text{FALSE} \rightarrow \text{shift to right}$
$LKS = \text{TRUE} \rightarrow \text{shift to left}$

**ROT**
Shift type: ROTATE
The bit position released by the shift procedure is replaced by the bit shifted out. The contents of the CARRY FLAG is additionally replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.

- ROT to the right: C MSB LSB
- ROT to the left: C MSB LSB

**SLOG**
Shift type: LOGICAL SHIFT
The bit position released by the shift procedure is replaced by the value 0.

- SLOG to the right: C 0 MSB LSB
- SLOG to the left: C MSB LSB 0

**ROTC**
Shift type: ROTATE by the CARRY FLAG.
The bit position released by the shift procedure is replaced by the contents of the CARRY FLAG. The CARRY FLAG is then replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.

- ROTC to the right: C MSB LSB
- ROTC to the left: C MSB LSB

**SOL**
Shift type: LOGICAL SHIFT
The bit position released by the shift procedure is replaced by the value 0.

- SLO to the right: C 0 MSB LSB
- SLO to the left: C MSB LSB 0
SHIFT BLOCK, DOUBLE WORD

SARI
Shift type: ARITHMETIC SHIFT

ARITHMETIC SHIFT to the right:
The bit position 15 (MSB, sign bit) released by the shift operation is replaced by itself. The contents of the CARRY FLAG is replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.

ARITHMETIC SHIFT to the left (identical with SLOG to the left):
The bit position 0 released by the shift procedure is replaced by the value 0. The contents of the CARRY FLAG is replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.

SARI to the right: C MSB LSB
SARI to the left (identical with SLOG to the left): C MSB LSB 0
LSB: Least significant bit
MSB: Most significant bit

Function call in IL
CAL SHIFTD1(E := SD_E, ANZ := SD_ANZ, LKS := SD_LKS, ROT := SD_ROT, ROTC := SD_ROTC, SLOG := SD_SLOG, SARI := SD_SARI, CY_E := SD_CYE)
LD SHIFTD1.A
ST SD_A
LD SHIFTD1.CY_A
ST SD_CYA

Note: In IL, the function call has to be performed in one line.

Function call in ST
SHIFTD1(E := SD_E, ANZ := SD_ANZ, LKS := SD_LKS, ROT := SD_ROT, ROTC := SD_ROTC, SLOG := SD_SLOG, SARI := SD_SARI, CY_E := SD_CYE);
SD_A:=SHIFTD1.A;
SD_CYA:=SHIFTD1.CY_A;

SARI to the right:

SARI to the left:

CY_E
Initial value for the CARRY FLAG in case of shift type ROTC. For the shift type »ROTATE by the CARRY FLAG«, an initial value is needed for the CARRY FLAG. This initial value is specified at the CY_E input.

CY_A
State of the CARRY FLAG after the shift procedure. After the shift procedure, the current value of the CARRY FLAG is available at this output.

A
Result of the shift procedure.
After the shift procedure, the result is available at this output.
SHIFT BLOCK, WORD

This function block shifts the operand present at the input by a specified number of bit positions to the left or right.

**Block type**

Function block without historical values

**Parameters**

- **Instance**  SHIFTW          Instance name
- **E**  INT          Operand to be shifted
- **ANZ**  INT         Number of bit positions to be shifted
- **LKS**  BOOL        Shift direction, left or right
  - LKS = FALSE → shift to right
  - LKS = TRUE → shift to left
- **ROT**  BOOL        Shift type: ROTATE
- **ROTC**  BOOL       Shift type: ROTATE by the CARRY FLAG
- **SLOG**  BOOL       Shift type: LOGICAL SHIFT
- **SARI**  BOOL       Shift type: ARITHMETIC SHIFT
- **CY_E**  BOOL       Initial value for the CARRY FLAG in case of shift type ROTC
- **CY_A**  BOOL       CARRY FLAG state after the shift procedure
- **A**  INT           Result of the shift procedure

**Description**

This function block shifts the operand present at the input by a specified number of bit positions to the left or right.

The result of the shift procedure and the CARRY flag are available at the block outputs. The operand at the input remains unchanged.

The required shift type is planned at the inputs:

- **ROT**
- **ROTC**
- **SLOG**
- **SARI**

If several shift types are specified simultaneously, the applying shift type is the one which is located furthest to the front in the sequence of the block inputs.

The block can only shift word operands.

Caution:

In certain circumstances, the value 8000H forbidden for arithmetic operations may be present at this block output.

The inputs and outputs can neither be duplicated nor inverted nor negated.
E  INT
Operand to be shifted. The planned SHIFT operation is applied to the input operand. In doing so, the input operand is not changed.

ANZ  INT
Number of bit positions to be shifted. Sensible range for the quantity n:
0 ≤ n ≤ 32

LKS  BOOL
Direction in which shifting is performed.
LKS = FALSE → shift to right
LKS = TRUE → shift to left

ROT  BOOL
Shift type: ROTATE
The bit position released by the shift procedure is replaced by the bit shifted out. The contents of the CARRY FLAG is additionally replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.
ROT to the right: C MSB LSB
ROT to the left: C MSB LSB
LSB: Least significant bit
MSB: Most significant bit

ROTC  BOOL
Shift type: ROTATE by the CARRY FLAG.
The bit position released by the shift procedure is replaced by the contents of the CARRY FLAG. The CARRY FLAG is then replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.
ROTC to the right: C MSB LSB
ROTC to the left: C MSB LSB
LSB: Least significant bit
MSB: Most significant bit

SLOG  BOOL
Shift type: LOGICAL SHIFT
The bit position released by the shift procedure is replaced by the value 0. The contents of the CARRY FLAG is replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.
SLOG to the right: C 0 MSB LSB
SLOG to the left: C MSB LSB 0
LSB: Least significant bit
MSB: Most significant bit
SHIFT BLOCK, WORD

SARI
Shift type: ARITHMETIC SHIFT

ARITHMETIC SHIFT to the right:
The bit position 15 (MSB, sign bit) released by the shift operation is replaced by itself. The contents of the CARRY FLAG is replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.

ARITHMETIC SHIFT to the left (identical with SLOG to the left):
The bit position 0 released by the shift procedure is replaced by the value 0. The contents of the CARRY FLAG is replaced by the bit shifted out. After the shift procedure, the result and the contents of the CARRY FLAG are available at the block outputs.

SARI to the right: C MSB LSB
SARI to the left: (identical with SLOG to the left) C MSB LSB 0
LSB: Least significant bit
MSB: Most significant

CY_E
Initial value for the CARRY FLAG in case of shift type ROTC. For the shift type »ROTATE by the CARRY FLAG«, an initial value is needed for the CARRY FLAG. This initial value is specified at the CY_E input.

CY_A
State of the CARRY FLAG after the shift procedure. After the shift procedure, the current value of the CARRY FLAG is available at this output.

A
Result of the shift procedure
After the shift procedure, the result is available at this output.

Function call in IL

CAL SHIFTW1(E := SW_E, ANZ := SW_ANZ,
LKS := SW_LKS, ROT := SW_ROT,
ROTC := SW_ROTC, SLOG := SW_SLOG,
SARI := SW_SARI, CY_E := SW_CYE)
LD SHIFTW1.A
ST SW_A
LD SHIFTW1.CY_A
ST SW_CYA

Note: In IL, the function call has to be performed in one line.

Function call in ST

SHIFTW1(E := SW_E, ANZ := SW_ANZ,
LKS := SW_LKS, ROT := SW_ROT,
ROTC := SW_ROTC, SLOG := SW_SLOG,
SARI := SW_SARI, CY_E := SW_CYE);
SW_A:=SHIFTW1.A;
SW_CYA:=SHIFTW1.CY_A;
The block calculates the sine value from input ANG and assigns it to output AD. The result is within the range of -100,000 to +100,000. If the value at ANG is negative or greater than 3600 (360°), output AD is set to 0 and output ERR is set to TRUE. The maximum deviation of the result is ±0.5.

**Block type**

Function block without historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>SIN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANG</td>
<td>INT</td>
</tr>
<tr>
<td>AD</td>
<td>DINT</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
</tr>
</tbody>
</table>

**Description**

<table>
<thead>
<tr>
<th>ANG</th>
<th>INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0 degrees</td>
</tr>
<tr>
<td>0001</td>
<td>0.1 degrees</td>
</tr>
<tr>
<td>0010</td>
<td>1.0 degrees</td>
</tr>
<tr>
<td>3600</td>
<td>360.0 degrees</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AD</th>
<th>DINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sine value of input ANG is available at output AD.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ERR</th>
<th>BOOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>The output ERR indicates whether the input value is within the correct range (0 ≤ ANG ≤ 3600).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input 0 ≤ ANG ≤ 3600 → ERR = 0 and AD = SIN(ANG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input ANG &lt; 0 or ANG &gt; 3600 → ERR = TRUE and AD = 0</td>
</tr>
</tbody>
</table>

**Examples for sine values**

<table>
<thead>
<tr>
<th>x angle</th>
<th>sin (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>100000</td>
</tr>
<tr>
<td>45</td>
<td>70711</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>180</td>
<td>0</td>
</tr>
<tr>
<td>360</td>
<td>0</td>
</tr>
<tr>
<td>236</td>
<td>-82904</td>
</tr>
<tr>
<td>270</td>
<td>-100000</td>
</tr>
</tbody>
</table>
Function call in IL
CAL  SIN11(ANG := SIN_ANG)
LD   SIN11.ERR
ST   SIN.ERR
LD   SIN11.AD
ST   SIN_AD

Function call in ST
SIN11(ANG := SIN_ANG);
SIN_ERR:=SIN11.ERR;
SIN_AD:=SIN11.AD;
SUBTRACTION, DOUBLE WORD

The value of the operand at input E2 is subtracted from the value of the operand at input E1 and the result is assigned to the operand at output A.

The result is limited to the maximum or minimum value of the number range (Number range: -2147483647 … 2147483647). If limiting occurred, a TRUE signal is assigned to the binary operand at output Q. If no limiting occurred, a FALSE signal is assigned to the binary operand at output Q.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>SUBD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Minuend</td>
</tr>
<tr>
<td>E2</td>
<td>DINT</td>
<td>Subtrahend</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Result (difference)</td>
</tr>
<tr>
<td>Q</td>
<td>BOOL</td>
<td>Result limited</td>
</tr>
</tbody>
</table>

Description

The value of the operand at input E2 is subtracted from the value of the operand at input E1 and the result is assigned to the operand at output A.

The result is limited to the maximum or minimum value of the number range (Number range: -2147483647 … 2147483647). If limiting occurred, a TRUE signal is assigned to the binary operand at output Q. If no limiting occurred, a FALSE signal is assigned to the binary operand at output Q.

The inputs and outputs can neither be duplicated nor negated.

Function call in IL

CAL SUBD1(E1 := SUBD_E1, E2 := SUBD_E2)
LD SUBD1.Q
ST SUBD_Q
LD SUBD1.A
ST SUBD_A

Function call in ST

SUBD1(E1 := SUBD_E1, E2 := SUBD_E2);
SUBD_Q:=SUBD1.Q;
SUBD_A:=SUBD1.A;
SUBTRACTION, WORD

The value of the operand at input E2 is subtracted from the value of the operand at input E1 and the result is assigned to the operand at output A.

Block type
Function

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Minuend</td>
</tr>
<tr>
<td>E2</td>
<td>INT</td>
<td>Subtrahend; input can be duplicated</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
<td>Result (difference)</td>
</tr>
</tbody>
</table>

Description

The value of the operand at input E2 is subtracted from the value of the operand at input E1 and the result is assigned to the operand at output A.

The result is limited to the maximum value 32767 and the minimum value -32767.

The inputs and the output can neither be duplicated nor negated.

Function call in IL

LD SUBW_E1
SUBW SUBW_E2
ST SUBW_A

Function call in ST

SUBW_A:=SUBW(SUBW_E1, SUBW_E2);
The bit at input BIT is tested in double word and assigned to the binary variable A. Valid values at input BIT: 0…31.

<table>
<thead>
<tr>
<th>Block type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Function</td>
</tr>
<tr>
<td>E  DINT</td>
<td>Input value</td>
</tr>
<tr>
<td>BIT INT</td>
<td>Bit number of the bit to be tested</td>
</tr>
<tr>
<td>A  BOOL</td>
<td>Output value</td>
</tr>
</tbody>
</table>

Description
The bit at input BIT is tested in double word and assigned to the binary variable A. Valid values at input BIT: 0…31.

The inputs and the output can neither be duplicated nor negated.

Function call in IL
LD  TESTD_E
TESTBD TESTD_BIT
ST  TESTD_A

Function call in ST
TESTD_A:=TESTBD(TESTD_E, TESTD_BIT);
**TEST BIT IN WORD**

The bit at input BIT is tested in word and assigned to the binary variable A. Valid values at input BIT: 0…15.

<table>
<thead>
<tr>
<th>Block type</th>
<th>Function</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E INT Input value</td>
<td>The bit at input BIT is tested in word and assigned to the binary variable A. Valid values at input BIT: 0…15.</td>
</tr>
<tr>
<td>BIT INT Bit number of the bit to be tested</td>
<td></td>
</tr>
<tr>
<td>A BOOL Output value</td>
<td>The inputs and the output can neither be duplicated nor negated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function call in IL</th>
<th>Function call in ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD TESTBW_E</td>
<td>TESTW_A:=TESTBW(TESTW_E, TESTW_BIT);</td>
</tr>
<tr>
<td>TESTBW</td>
<td></td>
</tr>
<tr>
<td>ST TESTW_A</td>
<td></td>
</tr>
</tbody>
</table>
DISPLAY AND SET CLOCK

This function block allows to set and display the current time and date.

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>UHR</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enable block processing</td>
</tr>
<tr>
<td>SET</td>
<td>BOOL</td>
<td>FALSE/TRUE edge sets the time and date</td>
</tr>
<tr>
<td>ESEC</td>
<td>INT</td>
<td>Set input for seconds</td>
</tr>
<tr>
<td>EMIN</td>
<td>INT</td>
<td>Set input for minutes</td>
</tr>
<tr>
<td>EH</td>
<td>INT</td>
<td>Set input for hours</td>
</tr>
<tr>
<td>ETAG</td>
<td>INT</td>
<td>Set input for days</td>
</tr>
<tr>
<td>EMON</td>
<td>INT</td>
<td>Set input for months</td>
</tr>
<tr>
<td>EJHR</td>
<td>INT</td>
<td>Set input for years</td>
</tr>
<tr>
<td>EWTG</td>
<td>INT</td>
<td>Set input for weekdays</td>
</tr>
<tr>
<td>AKT</td>
<td>BOOL</td>
<td>Topicality (usefulness) of the data at the outputs</td>
</tr>
<tr>
<td>ERR</td>
<td>INT</td>
<td>Error identifier</td>
</tr>
<tr>
<td>ASEc</td>
<td>INT</td>
<td>Output seconds</td>
</tr>
<tr>
<td>AMIN</td>
<td>INT</td>
<td>Output minutes</td>
</tr>
<tr>
<td>AH</td>
<td>INT</td>
<td>Output hours</td>
</tr>
<tr>
<td>ATAG</td>
<td>INT</td>
<td>Output days</td>
</tr>
<tr>
<td>AMON</td>
<td>INT</td>
<td>Output months</td>
</tr>
<tr>
<td>AJHR</td>
<td>INT</td>
<td>Output years</td>
</tr>
<tr>
<td>AWTG</td>
<td>INT</td>
<td>Output weekday no.</td>
</tr>
</tbody>
</table>

Description

This function block allows to set and display the current time and date.

The inputs and outputs can neither be duplicated nor inverted nor negated.

The clock is set by means of the set inputs for the time and date. The values present at the set inputs are adopted with the occurrence of a FALSE/TRUE edge at input SET. As long as a TRUE signal is present at the FREI input, the current date and time are indicated at the block's outputs.

FREI = FALSE:
The block is not processed. The AKT and ERR outputs are set to FALSE or 0. The time and date outputs are no longer changed by the block.

FREI = TRUE: Block is processed

SET BOOL
FALSE/TRUE edge
The clock is set to the values present at the time and date inputs.

During the setting procedure the time and date at the block outputs are invalid (output AKT = FALSE).
Set inputs for date and time
In the event of a FALSE/TRUE edge at input SET, the clock is set to the values present at the set inputs. If the specified set values are inadmissible, the AKT output is set to FALSE and an error message appears at the ERR output. In this case the values present at the time and date outputs are invalid. The clock has to be set again.

**ESEC**  
Set input for seconds.  
Value range: 0 … 59.

**EMIN**  
Set input for minutes.  
Value range: 0 … 59.

**EH**  
Set input for hours.  
The clock operates in 24 hour mode, i.e. it changes from 23:59:59 h to 0:0:0 h.  
Value range: 0 … 23.

**ETAG**  
Set input for days (which day of the month).  
The clock knows the number of days depending on the months and leap years. For the clock, a leap year exists when the year number is an integral multiple of 4. The maximum value for the days (28, 29, 30, 31) depends on the month.  
Value range: 1…28, 29, 30, 31.

**EMON**  
Set input for months.  
Value range: 1 … 12.

**EJHR**  
Set input for years.  
The clock only indicates the years and decades.  
Value range: 0 … 99.

**EWTG**  
Set input for the number of the weekday.  
This input specifies on which day of the week the input is made. That is to say, it is possible to determine which day of the week the day with the number 1 is to be (e.g. Sunday or Monday).  
Value range: 1 … 7.

Example:  
The clock is set on Friday, 01.07.88. If the value 6 is entered for the week day number, Friday is now the 6th day of the week and Sunday is defined as the 1st day of the week.

**AKT**  
Indication of the topicality (usefulness) of the outputs.  
AKT is 1 if:  
- The date and time outputs were updated in the current cycle.  
- The values at the outputs are consistent, i.e. none of the values at the date or time outputs has changed during updating. They all originate from the same clock pulse.  
- The clock was set correctly.  

**BOOL**  
AKT = TRUE → ERR = 0:  
Date/time are valid  
AKT = FALSE → ERR > 0:  
Date/time are invalid  
The reason is displayed at output ERR as an error identifier.

**ERR**  
In the event of an error, the relevant error identifier is available at output ERR.  
Meanings of the error identifiers:

- **No error has occurred:**  
  ERR=0: No error has occurred or FREI = FALSE, i.e. block disabled.

- **Error when setting the clock:**  
  ERR=1: 0 < SEC < 59 has not been met  
  ERR=2: 0 < MIN < 59 has not been met  
  ERR=3: 0 < H < 23 has not been met  
  ERR=4: 1 < TAG < 28, 29, 30, 31 (depending on the month) has not been met  
  ERR=5: 1 < MON < 12 has not been met  
  ERR=6: 0 < JHR < 99 has not been met  
  ERR=7: 1 < WTG < 7 has not been met  
  ERR=8: The transmission mailbox is currently occupied by another user. The block waits until the mailbox is free and afterwards date/time are set.  
  ERR=9: Date/time at the outputs are invalid.  
  ERR=10: Setting of date/time is currently performed; this may take several PLC cycles.  
  ERR=11: Setting was not successful, please repeat (unknown request code).  
  ERR=12: Setting was not successful, please repeat (invalid mail parameter).  
  ERR=13: Setting was not successful, please repeat (request code cannot be excexuted).
**Error when displaying date and time:**

ERR=9:  Date/time at the outputs are invalid.

**Outputs for date and time**

The outputs are updated whenever a TRUE signal is present at the FREI input and the clock has been set once. During the setting the outputs for date and time are invalid.

If the AKT output is equal to TRUE, the outputs for the date and time are valid. In case of an error, an error identifier is output at output ERR.

**ASEC**  
Output seconds. 
Value range: 0 … 59.

**AMIN**  
Output minutes. 
Value range: 0 … 59.

**AH**  
Output hours. 
Value range: 0 ... 23.

**ATAG**  
Output days. 
Value range: 1 ... 28, 29, 30, 31.

**AMON**  
Output months. 
Value range: 1 ... 12.

**AJHR**  
Output years. 
Value range: 0 ... 99.

**AWTG**  
Output weekday no. 
Value range: 1 ... 7.

---

**Function call in IL**

```
CAL UHR1(FREI := UHR_FREI, 
       SET := UHR_SET, ESEC := UHR_ESEC, 
       EMIN := UHR_EMIN, EH := UHR_EH, 
       ETAG := UHR_ETAG, 
       EMON := UHR_EMON, 
       EJHR := UHR_EJHR, EWO := UHR_EWO)
```

```
LD UHR1.ERR
ST UHR_ERR
LD UHR1.ASEC
ST UHR_ASEC
LD UHR1.AMIN
ST UHR_AMIN
LD UHR1.AH
ST UHR_AH
LD UHR1.ATAG
ST UHR_ATAG
LD UHR1.AMON
ST UHR_AMON
LD UHR1.AJHR
ST UHR_AJHR
LD UHR1.AWO
ST UHR_AWO
LD UHR1.AKT
ST UHR_AKT
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
UHR1(FREI := UHR_FREI, SET := UHR_SET, 
     ESEC := UHR_ESEC, EMIN := UHR_EMIN, 
     EH := UHR_EH, ETAG := UHR_ETAG, 
     EMON := UHR_EMON, EJHR := UHR_EJHR, 
     EWO := UHR_EWO);
```

```
UHR_ERR:=UHR1.ERR;
UHR_ASEC:=UHR1.ASEC;
UHR_AMIN:=UHR1.AMIN;
UHR_AH:=UHR1.AH;
UHR_ATAG:=UHR1.ATAG;
UHR_AMON:=UHR1.AMON;
UHR_AJHR:=UHR1.AJHR;
UHR_AWO:=UHR1.AWO;
UHR_AKT:=UHR1.AKT;
```
UNPACKING A WORD INTO BINARY VARIABLES

This function block unpacks the word variable at input E. Each bit of the input variable is assigned to one binary variable each (B0 ... Bn) at the output.

**Block type**
Function block without historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>UNPACK</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>INT</td>
<td>Word variable to be unpacked</td>
</tr>
<tr>
<td>n</td>
<td>INT</td>
<td>Number of output variables B0 ... Bn-1</td>
</tr>
<tr>
<td>B0..Bn</td>
<td>BOOL</td>
<td>Binary output variable</td>
</tr>
</tbody>
</table>

**Description**
This function block unpacks the word variable at input E. Each bit of the input variable is assigned to one binary variable each (B0 ... Bn-1) at the output.

**E**

The variable to be unpacked is specified at input E. Each bit (bit0 ... bit15) of this input variable is assigned to the assigned output variable (B0 ... Bn-1).

**n**

The number of planned binary outputs (B0 ... Bn-1) is specified at input n. The following applies: 1 ≤ n ≤ 16

**B0...B15**

The assigned bits of the variable at input E are assigned to the binary outputs B0 ... B15.

**Assignment:**

- Input variable Bit0 → B0
- Input variable Bit1 → B1
- ... 
- Input variable Bit15 → B15

**Function call in IL**

```
CAL UNPACK1(E := UNPACK_E, n := 3)
LD UNPACK1.B1
ST UNPACK_B1
LD UNPACK1.B2
ST UNPACK_B2
LD UNPACK1.B0
ST UNPACK_B0
```

**Function call in ST**

```
UNPACK1(E := UNPACK_E, n := 3);
UNPACK_B1:=UNPACK1.B1;
UNPACK_B2:=UNPACK1.B2;
UNPACK_B0:=UNPACK1.B0;
```
This function block unpacks the double word variable at input E. Each bit of the input variable is assigned to one binary variable each (B0 ... Bn) at the output.

**Description**

This function block unpacks the double word variable at input E. Each bit of the input variable is assigned to one binary variable each (B0 ... Bn-1) at the output.

**Parameters**

- **Instance**: UNPACKD
- **E**: DINT - Word variable to be unpacked
- **n**: INT - Number of output variables B0 ... Bn-1
- **B0..Bn**: BOOL - Binary output variable

**Assignment**

- Input variable Bit0 → B0
- Input variable Bit1 → B1
- ... ...
- Input variable Bit31 → B31

The allocated bits of the variable at input E are assigned to the binary outputs B0 ... B31.
UNPACKING A DOUBLE WORD INTO BINARY VARIABLES

Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>UNPACKD1(E := UNPACKD_E, n := 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>UNPACKD1.B1</td>
</tr>
<tr>
<td>ST</td>
<td>UNPACKD_B1</td>
</tr>
<tr>
<td>LD</td>
<td>UNPACKD1.B2</td>
</tr>
<tr>
<td>ST</td>
<td>UNPACKD_B2</td>
</tr>
<tr>
<td>LD</td>
<td>UNPACKD1.B0</td>
</tr>
<tr>
<td>ST</td>
<td>UNPACKD_B0</td>
</tr>
</tbody>
</table>

Function call in ST

UNPACKD1(E := UNPACKD_E, n := 3);
UNPACKD_B1:=UNPACKD1.B1;
UNPACKD_B2:=UNPACKD1.B2;
UNPACKD_B0:=UNPACKD1.B0;
This function block allocates the value from input E1 to an operand using the method of indirect addressing.

Note:
The USM block can only be used meaningfully in conjunction with the ADRWA block.

**Parameters**

<table>
<thead>
<tr>
<th>E1</th>
<th>INT</th>
<th>Value to be assigned to the operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>DWORD</td>
<td>Address of the operand to be written</td>
</tr>
</tbody>
</table>

**Description**

This function block allocates the value from input E1 to an operand using the method of indirect addressing.

Note:
The USM block can only be used meaningfully in conjunction with the ADRWA block.

The value of the operand at input ADR is interpreted as an address of the operand to be written (indirect addressing). This address is generated by the ADRWA block.

The inputs can neither be duplicated nor inverted.

Note:
An explanation of the indirect addressing method and the possibilities of using the USM block are described in the section dealing with the ADRWA block.

**Function call in IL**

LD USM_E1
USM USM_ADR

**Function call in ST**

USM(USM_E1, USM_ADR);
SWITCHOVER GATE

A FALSE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A1.

The respective output that is not assigned keeps its old value, but the old value is not updated.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0_1</td>
<td>BOOL</td>
<td>Switchover input</td>
</tr>
<tr>
<td>E1</td>
<td>INT</td>
<td>Input</td>
</tr>
<tr>
<td>A0</td>
<td>INT</td>
<td>Output for E0_1 = FALSE</td>
</tr>
<tr>
<td>A1</td>
<td>INT</td>
<td>Output for E0_1 = TRUE</td>
</tr>
</tbody>
</table>

Description

A FALSE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A1.

The respective output that is not assigned keeps its old value, but the old value is not updated.

The inputs and outputs can neither be duplicated nor negated/inverted.

Function call in IL

CAL UST1(E0_1 := UST_E01, E1 := UST_E1)
LD UST1.A1
ST UST_A1
LD UST1.A0
ST UST_A0

Function call in ST

UST1(E0_1 := UST_E01, E1 := UST_E1);
UST_A1:=UST1.A1;
UST_A0:=UST1.A0;
SWITCHOVER GATE / DOUBLE WORD

A FALSE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A1.

The respective output that is not assigned keeps its old value, but the old value is not updated.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>USTD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0_1</td>
<td>BOOL</td>
<td>Switchover input</td>
</tr>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Input</td>
</tr>
<tr>
<td>A0</td>
<td>DINT</td>
<td>Output for E0_1 = FALSE</td>
</tr>
<tr>
<td>A1</td>
<td>DINT</td>
<td>Output for E0_1 = TRUE</td>
</tr>
</tbody>
</table>

Description

A FALSE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A1.

The respective output that is not assigned keeps its old value, but the old value is not updated.

The inputs and outputs can neither be duplicated nor negated/inverted.

Function call in IL

```
CAL USTD1(E0_1 := USTD_E01,
          E1 := USTD_E1)
LD USTD1.A1
ST USTD_A1
LD USTD1.A0
ST USTD_A0
```

Note: In IL, the function call has to be performed in one line.

Function call in ST

```
USTD1(E0_1 := USTD_E01, E1 := USTD_E1);
USTD_A1:=USTD1.A1;
USTD_A0:=USTD1.A0;
```
SWITCHOVER GATE WITH RESET

A FALSE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A1.

The respective output that is not assigned is set to zero.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>USTR</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0_1</td>
<td>BOOL</td>
<td>Switchover input</td>
</tr>
<tr>
<td>E1</td>
<td>INT</td>
<td>Input</td>
</tr>
<tr>
<td>A0</td>
<td>INT</td>
<td>Output for E0_1 = FALSE</td>
</tr>
<tr>
<td>A1</td>
<td>INT</td>
<td>Output for E0_1 = TRUE</td>
</tr>
</tbody>
</table>

Description
A FALSE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the word operand at input E1 to the word operand at output A1.

The respective output that is not assigned is set to zero.

The inputs and outputs can neither be duplicated nor negated/inverted.

Function call in IL
CAL USTR1(E0_1 := USTR_E01, E1 := USTR_E1)
LD USTR1.A1
ST USTR_A1
LD USTR1.A0
ST USTR_A0

Note: In IL, the function call has to be performed in one line.
SWITCHOVER GATE WITH RESET / DOUBLE WORD

A FALSE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A1.

The respective output that is not assigned is set to zero.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>USTD</td>
<td>Instance name</td>
</tr>
<tr>
<td>E0_1</td>
<td>BOOL</td>
<td>Switchover input</td>
</tr>
<tr>
<td>E1</td>
<td>DINT</td>
<td>Input</td>
</tr>
<tr>
<td>A0</td>
<td>DINT</td>
<td>Output for E0_1 = FALSE</td>
</tr>
<tr>
<td>A1</td>
<td>DINT</td>
<td>Output for E0_1 = TRUE</td>
</tr>
</tbody>
</table>

Description

A FALSE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A0.

A TRUE signal at the binary input E0_1 assigns the value of the double word operand at input E1 to the word operand at output A1.

The respective output that is not assigned is set to zero.

The inputs and outputs can neither be duplicated nor negated/inverted.

Function call in IL

```
CAL USTRD1(E0_1 := USTRD_E01,
            E1 := USTRD_E1)
LD USTRD1.A1
ST USTRD_A1
LD USTRD1.A0
ST USTRD_A0
```

Note: In IL, the function call has to be performed in one line.

Function call in ST

```
USTRD1(E0_1 := USTRD_E01, E1 := USTRD_E1);
USTRD_A1:=USTRD1.A1;
USTRD_A0:=USTRD1.A0;
```
COMPARATOR WITH 3-POINT RESPONSE

The value of the operand at input E is compared to the values of the operands at the inputs OG and UG.

The possible results are signaled at the outputs E_OG, E_UG and Q.

The following applies:

\[ E < UG \]
\[ \rightarrow E_{OG} = \text{FALSE}, E_{UG} = \text{TRUE}, Q = \text{FALSE} \]
\[ UG \leq E \leq OG \]
\[ \rightarrow E_{OG} = \text{FALSE}, E_{UG} = \text{FALSE}, Q = \text{TRUE} \]
\[ E > OG \]
\[ \rightarrow E_{OG} = \text{TRUE}, E_{UG} = \text{FALSE}, Q = \text{FALSE} \]

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>VGL3P</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>INT</td>
<td>Input value</td>
</tr>
<tr>
<td>OG</td>
<td>INT</td>
<td>High limit</td>
</tr>
<tr>
<td>UG</td>
<td>INT</td>
<td>Low limit</td>
</tr>
<tr>
<td>E_OG</td>
<td>BOOL</td>
<td>Value &gt; high limit</td>
</tr>
<tr>
<td>E_UG</td>
<td>BOOL</td>
<td>Value &lt; low limit</td>
</tr>
<tr>
<td>Q</td>
<td>BOOL</td>
<td>Low limit \leq \text{input value} \leq \text{high limit}</td>
</tr>
</tbody>
</table>

Description

The value of the operand at input E is compared to the values of the operands at the inputs OG and UG.

The possible results are signaled at the outputs E_OG, E_UG and Q.

The following applies:

\[ E < UG \]
\[ \rightarrow E_{OG} = \text{FALSE}, E_{UG} = \text{TRUE}, Q = \text{FALSE} \]
\[ UG \leq E \leq OG \]
\[ \rightarrow E_{OG} = \text{FALSE}, E_{UG} = \text{FALSE}, Q = \text{TRUE} \]
\[ E > OG \]
\[ \rightarrow E_{OG} = \text{TRUE}, E_{UG} = \text{FALSE}, Q = \text{FALSE} \]

The inputs and outputs can neither be duplicated nor negated.
Function call in IL

**CAL** VGL3P1(E := V3P_E, OG := V3P_OG,
UG := V3P_UG)

LD VGL3P1.E_UG
ST V3P_EUG
LD VGL3P1.Q
ST V3P_Q
LD VGL3P1.E_OG
ST V3P_EOG

Note: In IL, the function call has to be performed in one line.

Function call in ST

VGL3P1(E := V3P_E, OG := V3P_OG,
UG := V3P_UG);
V3P_EUG:=VGL3P1.E_UG;
V3P_Q:=VGL3P1.Q;
V3P_EOG:=VGL3P1.E_OG;
The values of the operands at the inputs E1 and E2 are compared to each other. Taking the hysteresis at input HSY into account, the result is signaled at output Q.

The following applies:

\[ E1 > E2 \rightarrow Q = \text{TRUE} \]
\[ E1 < E2 - \text{HSY} \rightarrow Q = \text{FALSE} \]
\[ E2 - \text{HSY} \leq E1 < E2 \rightarrow Q \text{ as in the previous cycle} \]

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Input value 1</td>
</tr>
<tr>
<td>E2</td>
<td>INT</td>
<td>Input value 2</td>
</tr>
<tr>
<td>HYS</td>
<td>INT</td>
<td>Hysteresis</td>
</tr>
<tr>
<td>Q</td>
<td>BOOL</td>
<td>Result of the comparison</td>
</tr>
</tbody>
</table>

**Description** see next page
The values of the operands at the inputs E1 and E2 are compared to each other. Taking the hysteresis at input HSY into account, the result is signaled at output Q.

The following applies:

\[
\begin{align*}
E1 & \geq E2 \\
\rightarrow & \; Q = 1 \\
E1 & < E2 - HYS \\
\rightarrow & \; Q = \text{FALSE} \\
E2 - HYS & \leq E1 < E2 \\
\rightarrow & \; Q \text{ as in the previous cycle} \\
\end{align*}
\]

The inputs can neither be duplicated nor inverted. The output can neither be duplicated nor inverted.

**Function call in IL**

```il
CAL VGLEH1(E1 := VEH_E1, E2 := VEH_E2, HYS := VEH_HYS)
LD VGLEH1.Q
ST VEH_Q
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```st
VGLEH1(E1 := VEH_E1, E2 := VEH_E2, HYS := VEH_HYS);
VEH_Q:=VGLEH1.Q;
```

**Number range**

Integer word (16 bits)

- Low limit: 8001H 32767
- High limit: 7FFFH +32767
- Inadmissible value: 8000H ---

The following especially applies here to the specification for the left edge of the hysteresis:

\[
E2 - HYS \geq -32767 (8001H)
\]

In the two's complement arithmetic, the value 8000H (-32768) lies outside of the number range and is neither generated nor processed correctly by the PLC. If this forbidden value reaches the PLC

- by bit manipulations of the user or
- by reading from outside the PLC or
- by an indirect word constant

it is absolutely not allowed to carry out a negation or subtraction on this value.
The values of the operands at the inputs E1 and E2 are compared to each other. Taking the hystereses at the inputs OHYS (high hysteresis) and UHYS (low hysteresis) into account, the result is signaled at output Q.

The following applies:

E1 < E2 – UHYS
→ Q = FALSE

E1 ≥ E2 + OHYS
→ Q = TRUE

E2 - UHYS ≤ E1 ≤ E2 + OHYS
→ Q as in the previous cycle

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>INT</td>
<td>Input value 1</td>
</tr>
<tr>
<td>E2</td>
<td>INT</td>
<td>Input value 2</td>
</tr>
<tr>
<td>OHYS</td>
<td>INT</td>
<td>High hysteresis</td>
</tr>
<tr>
<td>UHYS</td>
<td>INT</td>
<td>Low hysteresis</td>
</tr>
<tr>
<td>Q</td>
<td>BOOL</td>
<td>Result of the comparison</td>
</tr>
</tbody>
</table>

**Description** see next page
The values of the operands at the inputs E1 and E2 are compared to each other. Taking the hystereses at the inputs OHYS (high hysteresis) and UHYS (low hysteresis) into account, the result is signaled at output Q.

The following applies:

\[ E1 < E2 - UHYS \rightarrow Q = \text{FALSE} \]
\[ E1 \geq E2 + OHYS \rightarrow Q = 1 \]
\[ E2 - UHYS \leq E1 \leq E2 + OHYS \rightarrow Q \text{ as in the previous cycle} \]

The inputs can neither be duplicated nor inverted. The output can neither be duplicated nor inverted.

### Number range

Integer word (16 bits)

- Low limit: 8001H 32767
- High limit: 7FFFH +32767
- Inadmissible value: 8000H ---

The following especially applies here to the specification for the left edge of the hysteresis:

\[ E2 - HYS \geq -32767 \text{ (8001H)} \]

In the two's complement arithmetic, the value 8000H (-32768) lies outside of the number range and is neither generated nor processed correctly by the PLC. If this forbidden value reaches the PLC

- by bit manipulations of the user or
- by reading from outside the PLC or
- by an indirect word constant

it is absolutely not allowed to carry out a negation or subtraction on this value.

### Function call in IL

```plaintext
CAL VGLUH1(E1 := VUH_E1, E2 := VUH_E2,
            OHYS := VUH_OHYS,
            UHYS := VUH_UHYS)
```

LD VGLUH1.Q

ST VUH_Q

Note: In IL, the function call has to be performed in one line.

### Function call in ST

```plaintext
VGLUH1(E1 := VUH_E1, E2 := VUH_E2,
        OHYS := VUH_OHYS, UHYS := VUH_UHYS);
VUH_Q:=VGLUH1.Q;
```
UP-DOWN COUNTER

This function block is used to count pulses. During counting, the positive edge of the pulse is evaluated in each case. The counter is capable of counting up and down and the counting increment can be specified. It is possible to preset the counter content to an intermediate value.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>VRZ</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>ZV</td>
<td>BOOL</td>
<td>Pulse input, up counting</td>
</tr>
<tr>
<td>ZR</td>
<td>BOOL</td>
<td>Pulse input, down counting</td>
</tr>
<tr>
<td>DIFF</td>
<td>INT</td>
<td>Counter content change per positive edge (increment)</td>
</tr>
<tr>
<td>SET</td>
<td>BOOL</td>
<td>Set counter to an intermediate value</td>
</tr>
<tr>
<td>ZW</td>
<td>INT</td>
<td>Intermediate value</td>
</tr>
<tr>
<td>RES</td>
<td>BOOL</td>
<td>Reset counter</td>
</tr>
<tr>
<td>Z</td>
<td>INT</td>
<td>Output for counter content</td>
</tr>
</tbody>
</table>

Description

This function block is used to count pulses. During counting, the positive edge of the pulse is evaluated in each case. The counter is capable of counting up and down and the counting increment can be specified. It is possible to preset the counter content to an intermediate value.

FREI

Counting is enabled or disabled by means of the FREI input. The following applies:

FREI = FALSE → Counting disabled
FREI = TRUE → Counting enabled

ZV

Each positive edge (FALSE → TRUE edge) at the input ZV increases the current counter content by the increment specified at input DIFF.

ZR

Each positive edge (FALSE → TRUE edge) at the input ZR decreases the current counter content by the increment specified at input DIFF.

DIFF

The increment for the counting operation is specified at input DIFF. The increment is the value by which the counter is changed at the input ZV or ZR with each positive edge.

SET

By means of a TRUE signal at input SET, the counter content is set to the value specified at input ZW. Counting is blocked as long as a TRUE signal is present at input SET. Setting is also effective when a TRUE signal is present at input FREI.

ZW

The value to which the counter content is set by a TRUE signal at input SET is specified at the input ZW.

RES

A TRUE signal at input RES sets the counter content to the value 0. The reset input RES has the highest priority of all inputs.

Z

The current counter content is available at output Z. If the counter reaches the positive or negative limit of the number range, the counter is limited to this value. The inputs and the output can neither be duplicated nor negated/inverted.
Function call in IL


LD VRZ1.A
ST VRZ_A

Function call in ST

VRZ1(FREI := VRZ_FREI, ZV := VRZ_ZV, ZR := VRZ_ZR, DIFF := VRZ_DIFF, SET := VRZ_SET, ZW := VRZ_ZW, RESET := VRZ_RES);

VRZ_A:=VRZ1.A;

Note: In IL, the function call has to be performed in one line.
UP-DOWN COUNTER / DOUBLE WORD

This function block is used to count pulses. During counting, the positive edge of the pulse is evaluated in each case. The counter is capable of counting up and down and the counting increment can be specified. It is possible to preset the counter content to an intermediate value.

### Block type
Function block with historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>VRZD</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>ZV</td>
<td>BOOL</td>
<td>Pulse input, up counting</td>
</tr>
<tr>
<td>ZR</td>
<td>BOOL</td>
<td>Pulse input, down counting</td>
</tr>
<tr>
<td>DIFF</td>
<td>DINT</td>
<td>Counter content change per positive edge (increment)</td>
</tr>
<tr>
<td>SET</td>
<td>BOOL</td>
<td>Set counter to an intermediate value</td>
</tr>
<tr>
<td>ZW</td>
<td>DINT</td>
<td>Intermediate value</td>
</tr>
<tr>
<td>RES</td>
<td>BOOL</td>
<td>Reset counter</td>
</tr>
<tr>
<td>Z</td>
<td>DINT</td>
<td>Output for counter content</td>
</tr>
</tbody>
</table>

### Description
This function block is used to count pulses. During counting, the positive edge of the pulse is evaluated in each case. The counter is capable of counting up and down and the counting increment can be specified. It is possible to preset the counter content to an intermediate value.

**FREI**
Enabling of the block processing
Counting is enabled or disabled by means of the FREI input.
The following applies:
FREI = FALSE → Counting disabled
FREI = TRUE → Counting enabled

**ZV**
Pulse input, up counting
Each positive edge (FALSE → TRUE edge) at input ZV increases the current counter content by the increment specified at input DIFF.

**ZR**
Pulse input, down counting
Each positive edge (FALSE → TRUE edge) at input ZR decreases the current counter content by the increment specified at input DIFF.

**DIFF**
Counter content change per positive edge (increment)
The increment for the counting operation is specified at input DIFF. The increment is the value by which the counter is changed at input ZV or ZR with each positive edge.

**SET**
Enabling of the block processing
By means of a TRUE signal at input SET, the counter content is set to the value specified at the input ZW. Counting is blocked as long as a TRUE signal is present at input SET. Setting is also effective when a TRUE signal is present at input FREI.

**ZW**
Intermediate value
The value to which the counter content is set by a TRUE signal at input SET is specified at the input ZW.

**RES**
Reset counter
A TRUE signal at input RES sets the counter content to the value 0. The reset input RES has the highest priority of all inputs.

**Z**
Output for counter content
The current counter content is available at output Z.
If the counter reaches the positive or negative limit of the number range, the counter is limited to this value. The inputs and the output can neither be duplicated nor negated/inverted.
Function call in IL

`CAL VRZD1(FREI := VRZD_FREI, ZV := VRZD_ZV, ZR := VRZD_ZR, DIFF := VRZD_DIFF, SET := VRZD_SET, ZW := VRZD_ZW, RESET := VRZD_RES)`

`LD VRZD1.A`

`ST VRZD_A`

Note: In IL, the function call has to be performed in one line.

Function call in ST

`VRZD1(FREI := VRZD_FREI, ZV := VRZD_ZV, ZR := VRZD_ZR, DIFF := VRZD_DIFF, SET := VRZD_SET, ZW := VRZD_ZW, RESET := VRZD_RES);`

`VRZD_A:=VRZD1.A;`
VARIABLE DELAY ELEMENT

A FALSE/TRUE edge at input E is delayed by the time T_EIN and output at output Q. A subsequent FALSE/TRUE edge at input E is delayed by the time T_AUS and output at output Q.

If input E returns to the FALSE level before the time T_EIN is expired, output A remains in the FALSE level. If input E returns to the TRUE level before the time T_AUS is expired, output A remains in the TRUE level.

Maximum time offset at the output: < 1 cycle time

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>E</th>
<th>T_EIN</th>
<th>T_AUS</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance name</td>
<td>BOOL</td>
<td>TIMER</td>
<td>TIMER</td>
<td>BOOL</td>
</tr>
</tbody>
</table>

Description
A FALSE/TRUE edge at input E is delayed by the time T_EIN and output at output Q. A subsequent FALSE/TRUE edge at input E is delayed by the time T_AUS and output at output Q.

If input E returns to the FALSE level before the time T_EIN is expired, output A remains in the FALSE level. If input E returns to the TRUE level before the time T_AUS is expired, output A remains in the TRUE level.

Maximum time offset at the output: < 1 cycle time

The inputs and the output can neither be duplicated nor inverted.

General behavior
- Started timers are processed by the PLC’s operating system and are therefore completely independent of the PLC program processing. The operating system does not issue a corresponding message to the corresponding timer block in the PLC program until the timer has elapsed.

Function call in IL
CAL VVZ1(E := VVZ_E, T_EIN := T#200ms, T_AUS := T#300ms)
LD VVZ1.A
ST VVZ_A

Note: In IL, the function call has to be performed in one line.

Function call in ST
VVZ1(E := VVZ_E, T_EIN := T#200ms, T_AUS := T#300ms);
VVZ_A := VVZ1.A;
TWO WORDS INTO DOUBLE WORD CONVERSION

This block packs two words into a double word.

**Block type**
- Function

**Parameters**
- **MSW** (INT) Input higher significant word
- **LSW** (INT) Input lower significant word
- **DW** (DINT) Output double word

**Description**
With this block it is possible to pack two 16 bit words into a 32 bit double word. For example, this is necessary to transfer double words or to evaluate event timing marks.

The flag words must be specified directly at the inputs MSW and LSW. Data errors can occur, if the assignment is performed via intermediate flags (correction of the value 8000H).

Please note that for data records sent by MasterPiece systems the higher significant word is assigned to the lower address.

**Function call in IL**
- `LD WDW_MSW`
- `W2WDW WDW_LSW`
- `ST WDW_DW`

**Function call ST**

```plaintext
WDW_DW := W2WDW(WDW_MSW, WDW_LSW);
```
WRITE WORD IN THE EVENT OF VALUE CHANGE

If the value of the operand at input E1 changes in comparison with the value during previous processing of the block, the value of the operand at input E1 is written to the specified physical address.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>WAES</td>
<td>Instance name</td>
</tr>
<tr>
<td>E1</td>
<td>INT</td>
<td>Input for the operand to be read</td>
</tr>
<tr>
<td>ADRESSE</td>
<td>DWORD</td>
<td>32 bit address of the memory location to which the value of E1 must be written in the event of a change.</td>
</tr>
</tbody>
</table>

Description

If the value of the operand at input E1 changes in comparison with the value during previous processing of the block, the value of the operand at input E1 is written to the specified physical address.

The inputs can neither be duplicated nor inverted.

E1        INT
If the operand at input E1 changes, its value is written to the address specified at input ADRESSE.

ADRESSE   DWORD
The 32 bit address to be written is specified at input ADRESSE.

Function call in IL

LD  WAES_MW
ADR
ST  WAES1.ADRESSE
CAL  WAES1(E1 := WAES_E1)

Function call in ST

WAES1.ADRESSE:=ADR(WAES_MW);
WAES1(E1 := WAES_E1);
WORD DECODER

This block compares the value of the operand at input E to the reference values of the operands at the inputs EC0 ... ECn-1. The result of the comparison is signaled at the outputs.

The WDEC number indicates the maximum number of reference values. The following word decoders are available:

- WDEC8: Word decoder with a maximum of 8 reference values
- WDEC16: Word decoder with a maximum of 16 reference values
- WDEC32: Word decoder with a maximum of 32 reference values
- WDEC64: Word decoder with a maximum of 64 reference values
- WDEC256: Word decoder with a maximum of 256 reference values

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>WDEC(8..256)</td>
<td>Instance name</td>
</tr>
<tr>
<td>E</td>
<td>INT</td>
<td>Input</td>
</tr>
<tr>
<td>N</td>
<td>INT</td>
<td>Number of reference values</td>
</tr>
<tr>
<td>EC0...ECn-1</td>
<td>INT</td>
<td>Reference values</td>
</tr>
<tr>
<td>E=EC</td>
<td>BOOL</td>
<td>Coincidence indication</td>
</tr>
<tr>
<td>NR</td>
<td>INT</td>
<td>Number of the reference value in the event of coincidence</td>
</tr>
</tbody>
</table>

Description

The WDEC number indicates the maximum number of reference values. The following word decoders are available:

- WDEC8: Word decoder with a maximum of 8 reference values
- WDEC16: Word decoder with a maximum of 16 reference values
- WDEC32: Word decoder with a maximum of 32 reference values
- WDEC64: Word decoder with a maximum of 64 reference values
- WDEC256: Word decoder with a maximum of 256 reference values

This block compares the value of the operand at input E to the reference values of the operands at the inputs EC0 ... ECn-1. The result of the comparison is signaled at the outputs.

If input E agrees with at least one of the n reference values EC, output E=EC is set to TRUE. The number of the 1st reference value EC agreeing with input E is assigned to the operand at output NR. Therefore, the number may assume a value from 1 to n.

The outputs E=EC and NR are set to 0 if no agreement between the input value E and the reference value EC is determined.

The number of inputs EC must be specified at input n.

E | INT

The operand whose value is to be compared to the reference values of the inputs EC0 ... ECn-1 is specified at input E.

n | INT

The number of planned reference values is specified at input n.
EC0 ... ECn-1 INT
The operands for the reference values are specified at the inputs EC0 ... ECn-1. The value of the operand at input E is compared to this reference values.

E=EC BOOL
Agreement (coincidence) between the operand value of input E and one of the reference values is signaled at output E=EC.

The following applies:
E=EC = FALSE → No agreement (coincidence)
E=EC = TRUE → Agreement (coincidence)

NR INT
The number of the reference value that agrees with the value of the operand at input E is signaled at output NR.

The following applies:
No agreement (coincidence) → NR = 0
Agreement (coincidence) → NR = n where n > 1
E=EC0 → NR = 1
E=EC1 → NR = 2
. . .
E=ECn-1 → NR = n

Function call in IL
CAL WDEC1(E := WDEC_E, n := 3, EC0 := WDEC_EC0, EC1 := WDEC_EC1, EC2 := WDEC_EC2)
LD WDEC1.NR
ST WDEC_NR
LD WDEC1.E_EC
ST WDEC_EEC

Note: In IL, the function call has to be performed in one line.

Function call in ST
WDEC1(E := WDEC_E, n := 3, EC0 := WDEC_EC0, EC1 := WDEC_EC1, EC2 := WDEC_EC2);
WDEC_NR:=WDEC1.NR;
WDEC_EEC:=WDEC1.E_EC;
The value of the word operand at input E is converted to a double word variable and the result is assigned to the double word operand at output A.

### Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>INT</td>
<td>Word variable to be converted</td>
</tr>
<tr>
<td>A</td>
<td>DINT</td>
<td>Result of the conversion, double word variable</td>
</tr>
</tbody>
</table>

### Value range for E:

- $8000H \leq E < 7FFFH$
- $-32768 \leq E \leq 32767$

The input and the output can neither be duplicated nor negated.

### Function call in IL

- LD  WDW_E
- WDW
- ST  WDW_DW

### Function call in ST

- WDW_DW := WDW(WDW_E);
If there is a TRUE signal at input FREI, the value of the specified physical address is read and assigned to the operand at output A.

<table>
<thead>
<tr>
<th>Block type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
</tr>
<tr>
<td>ADRESSE</td>
<td>DWORD</td>
</tr>
<tr>
<td>A</td>
<td>INT</td>
</tr>
</tbody>
</table>

Description

The following applies:

- FREI = FALSE → Processing disabled
- FREI = TRUE → Processing enabled

The address to be read is specified at input ADRESSE. This is specified as 32 bit address.

<table>
<thead>
<tr>
<th>FREI</th>
<th>BOOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Processing of the block is enabled or disabled with the operand at the FREI input.

The value read is assigned to the operand at output A.

<table>
<thead>
<tr>
<th>Function call in IL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
</tr>
<tr>
<td>WOL_FREI</td>
</tr>
<tr>
<td>WOL</td>
</tr>
<tr>
<td>WOL_ADR</td>
</tr>
<tr>
<td>ST</td>
</tr>
<tr>
<td>WOL_A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function call in ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>WOL_A:=WOL(WOL_FREI, WOL_ADR);</td>
</tr>
</tbody>
</table>
WRITE WORD WITH ENABLING

If there is a TRUE signal at input FREI, the value of the operand at input E1 is read and then written to the specified physical address.

### Block type
- Function

### Parameters
- **FREI** BOOL: Enabling the block
- **E1** INT: Input for the operand to be written
- **ADRESSE** DWORD: Address of the memory location to which the value of E1 must be written

### Description
- If there is a TRUE signal at input FREI, the value of the operand at input E1 is read and then written to the specified physical address.
- The block is not processed if there is a FALSE signal at input FREI.
- The inputs can neither be duplicated nor inverted.

**FREI** BOOL
- Processing of the block is enabled or disabled with the operand at input FREI.

**E1** INT
- The operand at input E1 is read and its value is written to the defined physical address.

**ADRESSE** DWORD
- The address to be written is specified at input ADRESSE.

### Function call in IL
```
LD  WOS_FREI
WOS  WOS_E1,WOS_ADR
```

Note: In IL, the function call has to be performed in one line.

### Function call in ST
```
WOS(WOS_FREI, WOS_E1, WOS_ADR);
```
This block compares the value of the operand at input E to the reference values of the operands at the inputs EC0 ... ECn-1. If input E agrees with at least one of the reference values EC, output E=EC is set to TRUE. Output A receives the value of the output code AC which is assigned to the found reference value EC.

The WUMC number indicates the maximum number of reference values. The following word recoders are available:

- **WUMC8** Word recoder with a maximum of 8 reference values
- **WUMC16** Word recoder with a maximum of 16 reference values
- **WUMC32** Word recoder with a maximum of 32 reference values
- **WUMC64** Word recoder with a maximum of 64 reference values
- **WUMC256** Word recoder with a maximum of 256 reference values

**Block type**

- Function block without historical values

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>WUMC(8..256) Instance name</td>
</tr>
<tr>
<td>E</td>
<td>INT Input</td>
</tr>
<tr>
<td>N</td>
<td>INT Number n of reference values</td>
</tr>
<tr>
<td>EC0..ECn-1</td>
<td>INT Reference value</td>
</tr>
<tr>
<td>AC0..ECn-1</td>
<td>INT Output code</td>
</tr>
<tr>
<td>E=EC</td>
<td>BOOL Coincidence indication</td>
</tr>
<tr>
<td>A</td>
<td>INT Output of the output code’s value</td>
</tr>
</tbody>
</table>

**Description**

The WUMC number indicates the maximum number of reference values. The following word recoders are available:

- **WUMC8** Word recoder with a maximum of 8 reference values
- **WUMC16** Word recoder with a maximum of 16 reference values
- **WUMC32** Word recoder with a maximum of 32 reference values
- **WUMC64** Word recoder with a maximum of 64 reference values
- **WUMC256** Word recoder with a maximum of 256 reference values

An operand for the output code ACi is allocated to each reference value at the inputs ECi. The allocation of EC to AC is recognizable by the index i. The index begins with 0.

The number of inputs EC and AC must be specified as a direct constant at input n.

The inputs and outputs cannot be negated/inverted.
The operand whose value is to be compared to the values of the n reference values (EC0 ... ECn-1) is specified at input E.

The number of reference values (EC0 ... ECn-1) and output codes (AC0 ... ACn-1) is specified at input n.

The operands for the reference values are specified at the inputs EC0 ... ECn-1. The value of the operand at input E1 is compared to the reference values.

The output codes are specified at the inputs AC0 ... ACn-1. The output code ACi is applied at output A if input E agrees with one of the reference values ECi.

Assignment between reference values and output codes:

\[
\begin{align*}
EC0 & \leftrightarrow AC0 \\
EC1 & \leftrightarrow AC1 \\
\cdots & \\
ECn-1 & \leftrightarrow ACn-1
\end{align*}
\]

Agreement (coincidence) between the operand value of input E and one of the reference values is signaled at output E=EC.

The following applies:

\[
\begin{align*}
E=EC = FALSE & \rightarrow \text{No agreement (coincidence)} \\
E=EC = TRUE & \rightarrow \text{Agreement (coincidence)}
\end{align*}
\]

The output code ACi is applied at output A if input E agrees with one of the reference values ECi.

The following applies:

\[
\begin{align*}
A = 0 & \rightarrow \text{No agreement (coincidence)} \\
A = ACi & \rightarrow \text{Agreement (coincidence)}
\end{align*}
\]

**Function call in IL**

```
LD WDW_E
WDW
ST WDW_DW
```

**Function call in ST**

```
WDW_DW := WDW(WDW_E)
```
Glossary

BOOL
Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT
DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

DWORD
DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

INT
INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

WORD
WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The
returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a
structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

Function blocks

Function blocks are subroutines which can have as much inputs, outputs and internal variables as required. They
are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be
considered as type. When assigning an individual data record (declaration) to the function block, a function block
instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next.
This allows for example to realize counters which may not forget their counter value. I.e. function blocks can have
an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters. A function has maximally one output parameter. Note that the
  output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

For function blocks with historical values it has to be observed that instance names may not be defined several
times if different data sets should be called.

Example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRZ1</td>
<td>VRZ</td>
</tr>
<tr>
<td>VRZ2</td>
<td>VRZ</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Function blocks without historical values (memory):

For function blocks without historical values only one instance has to be defined for the FB type. This instance can
be used for several calls of the FB (also with different I/O values).

Example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD1</td>
<td>ADDD (for all ADDD blocks used in the program)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The instance name can be defined without any restrictions. The type is preset and identical to the function block
name.
## Index

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD S90</td>
<td>11</td>
</tr>
<tr>
<td>ADDW S90</td>
<td>12</td>
</tr>
<tr>
<td>ADRWA S90</td>
<td>13</td>
</tr>
<tr>
<td>AMELD S90</td>
<td>16</td>
</tr>
<tr>
<td>AMELDD S90</td>
<td>18</td>
</tr>
<tr>
<td>ASV S90</td>
<td>20</td>
</tr>
<tr>
<td>AWM S90</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>DUALBCDD 54</td>
</tr>
<tr>
<td></td>
<td>DW2W 56</td>
</tr>
<tr>
<td></td>
<td>DWAES 57</td>
</tr>
<tr>
<td></td>
<td>DWOL 58</td>
</tr>
<tr>
<td></td>
<td>DWOS 59</td>
</tr>
<tr>
<td></td>
<td>DWUMC 60</td>
</tr>
<tr>
<td></td>
<td>DWW 62</td>
</tr>
<tr>
<td></td>
<td>ESV 63</td>
</tr>
<tr>
<td></td>
<td>FIFOB 64</td>
</tr>
<tr>
<td></td>
<td>FIFOW 66</td>
</tr>
<tr>
<td></td>
<td>FKG 68</td>
</tr>
<tr>
<td></td>
<td>HLG 70</td>
</tr>
<tr>
<td></td>
<td>IDLB 74</td>
</tr>
<tr>
<td></td>
<td>IDLM 76</td>
</tr>
<tr>
<td></td>
<td>IDSB 78</td>
</tr>
<tr>
<td></td>
<td>IDSM 80</td>
</tr>
<tr>
<td></td>
<td>INITSB 82</td>
</tr>
<tr>
<td></td>
<td>INITSD 83</td>
</tr>
<tr>
<td></td>
<td>INITSW 84</td>
</tr>
<tr>
<td></td>
<td>INITVB 85</td>
</tr>
<tr>
<td></td>
<td>INITVW 87</td>
</tr>
<tr>
<td></td>
<td>INTK 89</td>
</tr>
<tr>
<td></td>
<td>LDT 93</td>
</tr>
<tr>
<td></td>
<td>LIFOB 95</td>
</tr>
<tr>
<td></td>
<td>LIFOW 97</td>
</tr>
<tr>
<td></td>
<td>LIZU 99</td>
</tr>
<tr>
<td></td>
<td>LZB 101</td>
</tr>
<tr>
<td></td>
<td>MAJ 102</td>
</tr>
<tr>
<td></td>
<td>MASKE 103</td>
</tr>
<tr>
<td></td>
<td>MASKED 104</td>
</tr>
<tr>
<td></td>
<td>MAZ 105</td>
</tr>
<tr>
<td></td>
<td>MAZD 106</td>
</tr>
<tr>
<td></td>
<td>MOA 107</td>
</tr>
<tr>
<td></td>
<td>MOK 108</td>
</tr>
<tr>
<td></td>
<td>MUL2N 109</td>
</tr>
<tr>
<td></td>
<td>MUL2ND 111</td>
</tr>
<tr>
<td></td>
<td>MULD 113</td>
</tr>
<tr>
<td></td>
<td>BCDDUAL 22</td>
</tr>
<tr>
<td></td>
<td>BCDDUALD 24</td>
</tr>
<tr>
<td></td>
<td>BEG 26</td>
</tr>
<tr>
<td></td>
<td>BEGD 27</td>
</tr>
<tr>
<td></td>
<td>BITSU 28</td>
</tr>
<tr>
<td></td>
<td>BMELD8 31</td>
</tr>
<tr>
<td></td>
<td>COMPARE 33</td>
</tr>
<tr>
<td></td>
<td>COPY 34</td>
</tr>
<tr>
<td></td>
<td>COS1 35</td>
</tr>
<tr>
<td></td>
<td>COUNTBD 37</td>
</tr>
<tr>
<td></td>
<td>COUNTBW 38</td>
</tr>
<tr>
<td></td>
<td>COUNTW 39</td>
</tr>
<tr>
<td></td>
<td>DIVD 43</td>
</tr>
<tr>
<td></td>
<td>DIVW 45</td>
</tr>
<tr>
<td></td>
<td>DMUX 46</td>
</tr>
<tr>
<td></td>
<td>DMUXD 48</td>
</tr>
<tr>
<td></td>
<td>DT1 50</td>
</tr>
<tr>
<td></td>
<td>DUALBCD 52</td>
</tr>
</tbody>
</table>
MULDI 114
MULW 115
MUXR 116
MUXRD 118
NEGD 120
NEGW 121
PACK 122
PACKD 124
PDM 126
PI 128
PIDT1 132
PT1 137
SETBD 139
SETBW 140
SHIFTD 141
SHIFTW 144
SIN1 147
SUBD 149
SUBW 150
TESTBD 151
TESTBW 152
UHR 153
UNPACK 156
UNPACKD 158
USM 160
UST 161
USTD 162
USTR 163
USTRD 164
VGL3P 165
VGLEH 167
VGLUH 169
VRZ 171
VRZD 173
VVZ 175
W2WDW 176
WAES 177
WDEC 178
WDWW 180

WOL 181
WOS 182
WUMC 183
BCDDUAL S90 22
BCDDUALD S90 24
BEG S90 26
BEGD S90 27
BITSU S90 28
BMELD S90 31

COMPARE S90 33
COPY S90 34
COS1 S90 35
COUNTBD S90 37
COUNTBW S90 38
COUNTW S90 39

DIVD S90 43
DIVW S90 45
DMUX S90 46
DMUXD S90 48
DT1 S90 50
DUALBCD S90 52
DUALBCDD S90 54
DW2W S90 56
DWAES S90 57
DWOL S90 58
DWOS S90 59
DWUMC S90 60
DWW S90 62

ESV S90 63

FIFOB S90 64
FIFOW S90 66
FKG S90 68

H
HLG S90 70

I
IDLB S90 74
IDLM S90 76
IDSB S90 78
IDSM S90 80
INITSB S90 82
INITSD S90 83
INITSW S90 84
INITVB S90 85
INITVW S90 87
INTK S90 89

L
LDT S90 93
LIFOB S90 95
LIFOW S90 97
LIZU S90 99
LZB S90 101

M
MAJ S90 102
MASKE S90 103
MASKED S90 104
MAZ S90 105
MAZD S90 106
MOA S90 107
MOK S90 108
MUL2N S90 109
MUL2ND s90 111
MULD S90 113
MULDI S90 114
MULW S90 115
MUXR S90 116
MUXRD S90 118

N
NEG S90 120
NEG S90 121

P
PACK S90 122
PACKD S90 124
PDM S90 126
PI S90 128
PIDT1 S90 132
PT1 S90 137

S
SETBD S90 139
SETBW S90 140
SHIFTD S90 141
SHIFTW S90 144
SIN1 S90 147
SUBD S90 149
SUBW S90 150

T
TESTBD S90 151
TESTBW S90 152

U
UHR S90 153
UNPACK S90 156
UNPACKD S90 158
USM S90 160
UST S90 161
USTD S90 162
USTR S90 163
USTRD S90 164

V
VGL3P S90 165
VGLEH S90 167
VGLUH S90 169
VRZ S90 171
VRZD S90 173
VVZ S90 175

W
W2WDW S90 176
WAES S90 177
WDEC S90 178
WDWW S90 180
WOL S90 181
WOS S90 182
WUMC S90 183
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

RCOM/RCOM+
Function Block Library
90 Series

RCOM
90 Series
Contents

RCOM library 2

1.1 CONSIDERABLE CHANGES IN RCOM LIBRARY RCOM BIB COMPARED TO CE LIBRARY 907 KP 90 .......................... 2

Function blocks: .................................. 2

1.2 PARAMETERS .................................. 2

1.3 NOTE ON BLOCK DATA .................................. 2

Overview of blocks listed alphabetically according to their call names 3

SET CLOCK ........................................... 4
COLD START ........................................... 6
DIAL COMMUNICATION PARTNER ......................................................... 8
HANG UP PHONE ........................................... 10
NORMALIZATION ........................................... 12
CARRY OUT EVENT POLL ........................................... 14
INITIALIZE 07 KP 90 R202 ........................................... 16
INITIALIZE 07 KP 90 R202 ........................................... 16
READ DATA FROM RCOM SLAVE ......................................................... 24
PROVIDE DATA FOR READ JOB ......................................................... 26
RECEIVE DATA FROM RCOM PARTNER ......................................................... 28
RCOM SYSTEM SERVICES ......................................................... 30
SEND DATA TO RCOM PARTNER ......................................................... 32
WARM START ........................................... 34

Glossary 37

Index 39
This chapter contains the description of the blocks which are available for project engineering with the 07 KP 90 R202.

1.1 CONSIDERABLE CHANGES IN RCOM LIBRARY RCOM BIB COMPARED TO CE LIBRARY 907 KP 90

All blocks were converted to function blocks according to the IEC norm. (The corresponding definition can be found in the programming software manual).

In the RCOM library, all blocks for data remote transfer using the 07 KP 90 coupler (function blocks) are included.

Note:
The blocks defined in the RCOM library cannot be used in simulation mode.

In order to use these blocks the RCOM library (RCOM_S90_V41.LIB) must be included in the library window.

Function blocks:

A main characteristic of the function blocks is, that an instance has to be defined when calling them. In this case it has to be observed that instance names must not be defined several times.

Example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ1</td>
<td>READ</td>
</tr>
<tr>
<td>READ2</td>
<td>READ</td>
</tr>
</tbody>
</table>

The instance name can be defined without any restrictions. The type is preset and identical to the function block name.

1.2 PARAMETERS

In this section, the most important parameters which are relevant for the RCOM blocks are briefly explained.

NET

This parameter is reserved for future coupler expansions and must always be zero.

NOD

This parameter indicates the target and source address of a job. The RCOM master has address zero, the valid range for the slave addresses is 1..254. For broadcast jobs (only system services) NOD must be 255.

IDT

This parameter contains the number of the data record which should be called with the block. Valid range for the parameter is 1 ... 255.

LEN

This parameter indicates the length of the called data record. LEN must be even and the valid range is 2 ... 16. For events LEN may be max. 14, because in addition to the user data the timing mark is transmitted.

1.3 NOTE ON BLOCK DATA

Programming system

The programming software 907 AC 1131 is required for block usage.
Overview of blocks listed alphabetically according to their call names

Used abbreviations:

- FBmV ... Function block with historical values
- FBoV ... Function block without historical values
- F ... Function

<table>
<thead>
<tr>
<th>CE name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK</td>
<td>FB</td>
<td>Set clock</td>
<td>4</td>
</tr>
<tr>
<td>COLDST</td>
<td>FB</td>
<td>Cold start</td>
<td>6</td>
</tr>
<tr>
<td>DIAL</td>
<td>FB</td>
<td>Dial communication partner</td>
<td>8</td>
</tr>
<tr>
<td>HANGUP</td>
<td>FB</td>
<td>Hang up phone</td>
<td>10</td>
</tr>
<tr>
<td>NORMAL</td>
<td>FB</td>
<td>Normalization</td>
<td>12</td>
</tr>
<tr>
<td>POLL</td>
<td>FB</td>
<td>Perform event poll</td>
<td>14</td>
</tr>
<tr>
<td>RCOM</td>
<td>FB</td>
<td>Initialize 07 KP 90 R202 for RCOM protocol</td>
<td>16</td>
</tr>
<tr>
<td>RCOM_PL</td>
<td>FB</td>
<td>Initialize 07 KP 90 R202 for RCOM+ protocol</td>
<td>20</td>
</tr>
<tr>
<td>READ</td>
<td>FB</td>
<td>Read data from RCOM slave</td>
<td>24</td>
</tr>
<tr>
<td>READ_S</td>
<td>FB</td>
<td>Provide data for READ job</td>
<td>26</td>
</tr>
<tr>
<td>RECV</td>
<td>FB</td>
<td>Receive data from RCOM partner</td>
<td>28</td>
</tr>
<tr>
<td>SYS_S</td>
<td>FB</td>
<td>RCOM system services</td>
<td>30</td>
</tr>
<tr>
<td>TRANSM</td>
<td>FB</td>
<td>Send data to RCOM partner</td>
<td>32</td>
</tr>
<tr>
<td>WARMST</td>
<td>FB</td>
<td>Warm start</td>
<td>34</td>
</tr>
</tbody>
</table>
SET CLOCK

With this block, the RCOM master sets his own clock and the clocks of the called slaves to a specified time.

The clocks are used to build timing marks during the event-controlled transfer. Additional to this, the RCOM system time is available for the user at the RCOM block.

Block type

Function block

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>CLOCK</th>
<th>instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>FRIE edge at this input sets the clocks</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>number of the called slave</td>
</tr>
<tr>
<td>Std</td>
<td>INT</td>
<td>time, hours</td>
</tr>
<tr>
<td>Minu</td>
<td>INT</td>
<td>time, minutes</td>
</tr>
<tr>
<td>Sek</td>
<td>INT</td>
<td>time, seconds</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>BOOL</td>
<td>error number</td>
</tr>
</tbody>
</table>

Description

The function block transmits a telegram which sets the time in the called slave to the specified value. The master clock is set to the same value.

FREI BOOL

A FALSE/TRUE edge at input FREI starts the clock setting. Further edges are ignored until RDY has become “TRUE”.

NET INT

Must always be zero.

NOD INT

Number of the slave. In order to set all slaves to the same time, here should be entered 255 (all slaves are called).

Std INT

Time, hours.

Minu INT

Time, minutes.

Sek INT

Time, seconds.

RDY BOOL

Output RDY indicates that the clock setting has been completed. This output has always to be considered together with the output ERR. The following applies:

RDY = TRUE and ERR = FALSE: The clocks have been set successfully.

RDY = TRUE and ERR = TRUE: While setting the clocks an error occurred. In this case, the error message at output ERR_NR can be evaluated.

After one attempt, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the clock setting has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.
Output ERR indicates whether an error occurred while setting the clock. Output ERR has always to be considered together with the output RDY.

If an error occurred, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.

At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are described in detail in appendix "Error codes".

**Function call in IL**

```
CAL   CLOCK1(FREI := FREI, NET := NET,
            NOD := NOD, Std := STD, Minu := MINU,
            Sek := SEK)
LDCLOCK1.ERR
STD   ERR
LD    CLOCK1.ERR_NR
STD   ERR_NR
LDCLOCK1.RDY
STD   RDY
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
CLOCK1(FREI := FREI, NET := NET,
       NOD := NOD, Std := STD, Minu := MINU,
       Sek := SEK);
ERR:=CLOCK1.ERR;
ERR_NR:=CLOCK1.ERR_NR;
RDY:=CLOCK1.RDY;
```
With this block, the RCOM master carries out a cold start. It can either start all slaves or one individual slave.

The slave is partially reinitialized after the cold start.

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLDST</td>
<td>instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>FALSE/TRUE edge at this input initiates a cold start</td>
</tr>
<tr>
<td>NET</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>number of the slave to be started</td>
</tr>
<tr>
<td>RDY</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>error number</td>
</tr>
</tbody>
</table>

### Description

The function block carries out a cold start at the called slave. The slave coupler is partially reinitialized.

At the called slave the sequence marks are reset (RCOM specific protocol parameter) and the transmission of data records is disabled. All entries in the event queue are deleted and after this a cold start event is generated.

After a cold start, always a normalization must occur.

The block has to be used for the initialization of the RCOM network, e.g. after the initialization 07 KP 90 R202 at the master.

A FREI edge at input FREI initiates a cold start. Further edges are ignored until RDY has become "TRUE".

Must always be zero.

Number of the slave to be started. For NOD=255 all slaves are called.

### Output RDY

Output RDY indicates that the cold start has been completed. This output has always to be considered together with the output ERR. The following applies:

- RDY = TRUE and ERR = FALSE: The cold start has been terminated. The planned slaves have been reset.
- RDY = TRUE and ERR = FALSE: An error occurred during the cold start. In this case, the error message at output ERR_NR can be evaluated.

After a cold start, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the cold start has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

### Output ERR

Output ERR indicates whether an error occurred during the cold start. Output ERR has always to be considered together with the output RDY.

If an error occurred during the cold start, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.
ERR_NR

At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are described in detail in appendix "Error codes".

Function call in IL

CAL COLDST1(FREI := FREI, NET := NET, NOD := NOD)
LD COLDST1.ERR
ST ERR
LD COLDST1.ERR_NR
ST ERR_NR
LD COLDST1.RDY
ST RDY

Function call in ST

COLDST1(FREI := FREI, NET := NET, NOD := NOD);
ERR:=COLDST1.ERR;
ERR_NR:=COLDST1.ERR_NR;
RDY:=COLDST1.RDY;

Note: In IL, the function call has to be performed in one line.
DIAL COMMUNICATION PARTNER

With this block a RCOM subscriber can call another subscriber.

A master can call any slave. Using this block, a slave can call the RCOM master to request event polling.

The block can only be used if the RCOM network is realized with switched connections (Mdm=1 for RCOM block).

Block type

Function block

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>DIAL</td>
<td>instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>FALSE/TRUE edge at this input starts the dialling procedure</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>number of the slave to be called</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
<td>error number</td>
</tr>
</tbody>
</table>

Description

For dial lines, the function block calls the specified subscriber. In the RCOM master, the block must be used before transmitting system services or data records to a slave. Using this block, a slave can call the master to request event polling.

The telephone line is terminated using the HANGUP block after transmission is completed.

**FREI**

A FALSE/TRUE edge at input FREI starts the dialling procedure. Further edges are ignored until RDY has become “TRUE”.

**NET**

Must always be zero.

**NOD**

Number of the subscriber to be called. Here, the entry in the telephone list is specified which should be called. Example: NOD=5: The 5. entry in the telephone list is called.

**RDY**

Output RDY indicates that the dialling has been completed. This output has always to be considered together with the output ERR. The following applies:

- RDY = TRUE and ERR = FALSE: The telephone line is ready for data transfer.
- RDY = TRUE and ERR = TRUE: An error occurred during the dialling procedure. In this case, the error message at output FeNR can be evaluated.
- After a dialling operation, a falling edge at input FREI sets the outputs RDY, ERR and FeNR to zero.
- If the dialling has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

**ERR**

Output ERR indicates whether an error occurred during the dialling. Output ERR has always to be considered together with the output RDY.

- If an error occurred during the dialling, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output FeNR.
ERR_NR INT
At output FeNR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are described in detail in appendix "Error codes".

Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>DIAL1(FREI := FREI, NET := NET, NOD := NOD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>DIAL1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>ERR</td>
</tr>
<tr>
<td>LD</td>
<td>DIAL1.ERR_NR</td>
</tr>
<tr>
<td>ST</td>
<td>ERR_NR</td>
</tr>
<tr>
<td>LD</td>
<td>DIAL1.RDY</td>
</tr>
<tr>
<td>ST</td>
<td>RDY</td>
</tr>
</tbody>
</table>

Note: In IL, the function call has to be performed in one line.

Function call in ST

DIAL1(FREI := FREI, NET := NET, NOD := NOD);
ERR:=DIAL1.ERR;
ERR_NR:=DIAL1.ERR_NR;
RDY:=DIAL1.RDY;
HANG UP PHONE

With this block, the master can hang up the phone in a RCOM network with switched connections.

The block cannot be used until all jobs for a slave are processed completely.

The block can only be used if the RCOM network is realized with switched connections (Mdm=1 for RCOM block).

Block type

Function block

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td></td>
<td>FALSE/TRUE edge at this input hangs up the phone</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td></td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td></td>
<td>number of the slave</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td></td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td></td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
<td></td>
<td>error number</td>
</tr>
</tbody>
</table>

Description

For dial lines, the function block hangs up the phone.

The block must be used to terminate a connection after transmitting data records to or from a slave.

FREI BOOL

A FALSE/TRUE edge at input FREI starts the hanging up operation. Further edges are ignored until RDY has become "TRUE".

NET INT

Must always be zero.

NOD INT

Number of the slave which has been called before. The 07 KP 90 R202 ignores this parameter, it is always hang up.

RDY BOOL

Output RDY indicates that the hanging up operation has been completed. This output has always to be considered together with the output ERR. The following applies:

RDY = TRUE and ERR = FALSE: The telephone line was successful interrupted.

RDY = TRUE and ERR = TRUE: An error occurred during the hang up operation. In this case, the error message at output ERR_NR can be evaluated.

After the hang up operation, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the hang up has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

ERR BOOL

Output ERR indicates whether an error occurred during the hang up operation. Output ERR has always to be considered together with the output RDY.

If an error occurred during the hang up operation, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.

ERR_NR INT

At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are described in detail in appendix "Error codes".
Function call in IL

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>HANGUP1(FREI := FREI, NET := NET, NOD := NOD)</td>
</tr>
<tr>
<td>LD</td>
<td>HANGUP1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>ERR</td>
</tr>
<tr>
<td>LD</td>
<td>HANGUP1.ERR_NR</td>
</tr>
<tr>
<td>ST</td>
<td>ERR_NR</td>
</tr>
<tr>
<td>LD</td>
<td>HANGUP1.RDY</td>
</tr>
<tr>
<td>ST</td>
<td>RDY</td>
</tr>
</tbody>
</table>

Note: In IL, the function call has to be performed in one line.

Function call in ST

HANGUP1(FREI := FREI, NET := NET, NOD := NOD);
ERR:=HANGUP1.ERR;
ERR_NR:=HANGUP1.ERR_NR;
RDY:=HANGUP1.RDY;
With this block, the master enables the data record transfer.

A normalization must be performed after a cold or warm start.

**Block type**

Function block

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>NORMAL</td>
<td>instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>FALSE/TRUE edge at this input initiates a normalization</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>number of the slave to be normalized</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
<td>error number</td>
</tr>
</tbody>
</table>

**Description**

The function block carries out a normalization at the called slave. Due to this the data record transfer is enabled.

Before the data record transfer, after the switch-on and after a cold or warm start, the block has to be processed once.

**FREI**

A FALSE/TRUE edge at input FREI starts the normalization. Further edges are ignored until RDY has become "TRUE".

**NET**

Must always be zero.

**NOD**

Number of the slave to be normalized. For NOD=255 all slaves are called.

**RDY**

Output RDY indicates that the normalization has been completed. This output has always to be considered together with the output ERR. The following applies:

RDY = TRUE and ERR = FALSE: The normalization has been completed.

RDY = TRUE and ERR = TRUE: An error occurred during the normalization. In this case, the error message at output ERR_NR can be evaluated.

After a normalization, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the normalization has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

**ERR**

Output ERR indicates whether an error occurred during the normalization. Output ERR has always to be considered together with the output RDY.

If an error occurred during the normalization, RDY = TRUE and ERR = TRUE are set. The type of error can be evaluated using the error number at output ERR_NR.

**ERR_NR**

At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are described in detail in appendix “Error codes”.
**Function call in IL**

CAL NORMAL1(FREI := FREI, NET := NET, NOD := NOD)

LD NORMAL1.ERR
ST ERR
LD NORMAL1.ERR_NR
ST ERR_NR
LD NORMAL1.RDY
ST RDY

**Function call in ST**

NORMAL1(FREI := FREI, NET := NET, NOD := NOD);
ERR:=NORMAL1.ERR;
ERR_NR:=NORMAL1.ERR_NR;
RDY:=NORMAL1.RDY;

Note: In IL, the function call has to be performed in one line.
CARRY OUT EVENT POLL

With this block the master performs an event poll on a
slave.

If the slave sends back data records, these appear in a
correspondingly configured RECV block in the master.

The block polls the slave until the number MaxP (block
RCOM) is reached or the slave has no further data
records.

Block type

Function block

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>POLL</th>
<th>instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>FREI edge at this input initiates event poll</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>number of the slave to be polled</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
<td>error number</td>
</tr>
</tbody>
</table>

Description

The block performs an event poll at the called slave,
i.e. the slave is asked whether it wants to transmit data
records to the master.

The slave data records appear at a correspondingly
configured RECV block in the master.

FREI BOOL
A FALSE/TRUE edge at input FREI starts the event
poll. Further edges are ignored until RDY has become
“TRUE”.

NET INT
Must always be zero.

NOD INT
Number of the slave to be polled (1...254).

RDY BOOL
Output RDY indicates that the poll process has been
completed. This output has always to be considered
together with the output ERR. The following applies:

RDY = TRUE and ERR = FALSE: The poll event has been completed.

RDY = TRUE and ERR = TRUE: An error occurred
during the polling. In this case, the error message at
output ERR_NR can be evaluated.

After a poll, a falling edge at input FREI sets the
outputs RDY, ERR and ERR_NR to zero.

If an error occurred during the polling, RDY = TRUE
and ERR = TRUE are set. The type of error can be
evaluated using the error number at output ERR_NR.

ERR BOOL
Output ERR indicates whether an error occurred during
the polling. Output ERR has always to be considered
together with the output RDY.

If an error occurred during the polling, RDY = TRUE
and ERR = TRUE are set. The type of error can be
evaluated using the error number at output ERR_NR.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”.

ERR_NR INT
At output ERR_NR an error number is output. This
output has always to be considered together with the
outputs RDY and ERR.

The error numbers are described in detail in appendix
“Error codes”. 
Function call in IL

CAL POLL1(FREI := FREI, NET := NET,
NOD := NOD)
LD POLL1.ERR
ST ERR
LD POLL1.ERR_NR
ST ERR_NR
LD POLL1.RDY
ST RDY

Note: In IL, the function call has to be performed in one line.

Function call in ST

POLL1(FREI := FREI, NET := NET,
NOD := NOD);
ERR:=POLL1.ERR;
ERR_NR:=POLL1.ERR_NR;
RDY:=POLL1.RDY;
This block is used for initializing the 07 KP 90 R202. All necessary network parameters are defined here. At the RCOM master and every RCOM slave, the initialization must be performed before starting other blocks.

The block enters the network parameters and then starts the coupler initialization.

The block also supplies the 07 KP 90 R202 cycle counter value (life identifier) and the RCOM system time until input FREI is “TRUE”. Additionally, the current event queue level (i.e. the number of data records which are still queued) is displayed at output EvtP of RCOM slaves.

**Block type**

Function block

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>RCOM</td>
<td>instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>enabling of the block processing with FALSE/TRUE edge</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always 0</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>RCOM master: 0, RCOM slave: number of the slave</td>
</tr>
<tr>
<td>Baud</td>
<td>INT</td>
<td>baud rate for RCOM interface</td>
</tr>
<tr>
<td>Pari</td>
<td>INT</td>
<td>parity: 0: none, 1: odd, 2: even</td>
</tr>
<tr>
<td>Dupl</td>
<td>INT</td>
<td>0: half duplex, 1: full duplex</td>
</tr>
<tr>
<td>TLS</td>
<td>INT</td>
<td>time for carrier stabilization</td>
</tr>
<tr>
<td>Cdly</td>
<td>INT</td>
<td>waiting time for carrier stop</td>
</tr>
<tr>
<td>CTO</td>
<td>INT</td>
<td>character delay time</td>
</tr>
<tr>
<td>TAT</td>
<td>INT</td>
<td>reaction time for response</td>
</tr>
<tr>
<td>Mdm</td>
<td>INT</td>
<td>modem type</td>
</tr>
<tr>
<td>Retr</td>
<td>INT</td>
<td>number of repetitive attempts</td>
</tr>
<tr>
<td>MaxP</td>
<td>INT</td>
<td>maximum number of event polls for a slave in a cycle</td>
</tr>
<tr>
<td>NoPr</td>
<td>INT</td>
<td>number of preambles in the protocol</td>
</tr>
<tr>
<td>Dbg</td>
<td>INT</td>
<td>debug step</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
<td>error number</td>
</tr>
<tr>
<td>Zykl</td>
<td>INT</td>
<td>cycle counter value</td>
</tr>
<tr>
<td>NT</td>
<td>BOOL</td>
<td>new time available</td>
</tr>
<tr>
<td>Std</td>
<td>INT</td>
<td>time, hours</td>
</tr>
<tr>
<td>Minu</td>
<td>INT</td>
<td>time, minutes</td>
</tr>
<tr>
<td>Sek</td>
<td>INT</td>
<td>time, seconds</td>
</tr>
<tr>
<td>EvtP</td>
<td>INT</td>
<td>number of events in the event queue</td>
</tr>
</tbody>
</table>
Description

The block is used to initialize the 07 KP 90 R202. The necessary network parameters are set and the initialization is started.

After the block has been started, the input parameters must not be changed. Therefore the parameters should be defined as constants.

With a FALSE/TRUE edge at input FREI the initialization is performed once.

If the initialization has not yet been completed, further FALSE/TRUE edges at input FREI are ignored. After the initialisation has been completed, the outputs RDY, ERR and ERR_NR are set.

The initialization was successful if RDY = TRUE and ERR = FALSE. For RDY = TRUE and ERR = TRUE, an error occurred. The type of error can be determined using the output ERR_NR.

After initialization, the outputs ERR, ERR_NR, Zyk, NT, Std, Min, Sek and EvtP are updated in every PLC cycle, if input FREI is “TRUE”. Therefore the 07 KP 90 R202 reports the cycle counter value or error reports to the PLC. Due to this, input FREI should always remain “TRUE” during running operation of the coupler, except when the coupler has to be reinitialized with new parameters.

ERR and ERR_NR are reset with a falling edge at input FREI.

Zyk contains the current cycle counter value in the 07 KP 90 R202. The outputs NT, Std, Min and Sek contain the RCOM system time which can also be used in the PLC program. Time starts at switch-on with 00:00.00. If the command ‘set clock’ is received, the time is set to the new value, NT then is set for approx. 5 seconds (e.g. suitable for setting a real time clock).

For a RCOM slave, output EvtP contains the number of events which are saved in the event queue. For the RCOM master, this output is always zero.

Note: For the RCOM parameters some English names in brackets are listed. They correspond to the master piece names in the ‘data set for line characteristics’.

FREI

With a FALSE/TRUE edge at input FREI the initialization is started. With this FALSE/TRUE edge, the outputs RDY, ERR and ERR_NR are also set to zero. If the writing operation has not yet been completed (RDY = FALSE), a further FALSE/TRUE edge at input FREI is ignored. During FREI is set to one, the outputs ERR, ERR_NR and Zyk are updated constantly.

NET

Must always be zero.

NOD

Indicates the RCOM network address.
0: RCOM master
otherwise: slave number

A RCOM network can contain only one master.

Baud

(bitrate) transmission rate, possible values are 300, 600, 1200, 2400, 4800, 9600, 19.200.

Pari

parity, possible values:
0 -> no parity
1 -> odd parity
2 -> even parity

Dupl

(duplex)
0 -> half duplex
1 -> full duplex

This parameter depends on the used transmission medium. Refer to chapter ‘Planning’.

TLS

(line stab. time) Number of characters used as waiting time before sending a telegram (e.g. for carrier stabilization). Recommended values for 1200 baud: 3 characters for dial lines, 2 characters for multidrop lines. Refer to section ‘Hardware Handshake’.

CDly

(carrier delay) Number of characters used as waiting time after sending a telegram. Recommended values for 1200 baud: 2 characters for dial lines, 0 characters for multidrop lines. Refer to section ‘Hardware Handshake’.

CTO

(char. timeout) character delay time. Number of characters which are waited, before a telegram is rejected as interrupted. Recommended value: 30.
**TAT**  INT
(turnaround time) Maximum time in msec for which the RCOM master waits for a slave response (maximum slave reaction time). Must always be specified greater than 2000. Recommended value: 4000.

**Mdm**  INT
(type of modem) Used modem. The following modem types are permissible:
0: direct connection or fixed line modem
1: Hayes compatible dial modem
2: installed RS485 interface
3: modem 23 WT 90 half duplex

**Retr**  INT
(retransmissions) Maximum number of repetitive attempts before the line is considered as interrupted. Recommended value: 2.

**MaxP**  INT
(max no poll) If during the event poll an event is sent by the slave, the master can perform further event polls. This parameter specifies the maximum numbers of event polls which are carried out. Recommended value: 3.

**NoPr**  INT
(no of preambles) Number of preambles in the RCOM protocol. Recommended values: 1 for multidrop, 3 for dial lines.

**RDY**  BOOL
The output RDY indicates that the initialization has been completed. This output has always to be considered together with the output ERR. The following applies:
RDY = TRUE and ERR = FALSE: Initialization has been completed. The parameters were read and accepted by the 07 KP 90.
RDY = TRUE and ERR = TRUE: An error occurred during the initialization. In this case, the error message at output ERR_NR can be evaluated.

After an initialization, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.
If the initialization has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

**ERR**  BOOL
Output ERR indicates whether an error occurred during the initialization. Output ERR has always to be considered together with the output RDY.
If an error occurred, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.

ERR is also set if FREI is still “TRUE” and the 07 KP 90 R202 reports a fatal error.

**ERR_NR**  INT
At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

ERR_NR can also report an error in running operation, if input FREI is “TRUE”.
The error numbers are described in detail in appendix “Error codes”.

**Zykl**  INT
At output Zykl, the current value of the 07 KP 90 R202 cycle counter is updated as long as FREI is “TRUE”. The cycle counter should be used for monitoring the 07 KP 90 R202 (life identifier).

**NT**  BOOL
Output NT is set for approx. 5 seconds to “TRUE” if a new RCOM time was set (with block CLOCK). It can be used for setting a real time clock with the current values of Std, Min and Sek.

**Std**  INT
The output contains the RCOM system time in hours. The master can set the time using the CLOCK block.

**Minu**  INT
The output contains the RCOM system time in minutes.

**Sek**  INT
The output contains the RCOM system time in seconds.

**EvtP**  INT
For a RCOM slave, the output contains the number of events which are saved in the event queue. For the RCOM master, this output is always zero.
Function call in IL

LD RCOM1.ERR
ST ERR
LD RCOM1.ERR_NR
ST ERR_NR
LD RCOM1.Zykl
ST ZYKL
LD RCOM1.NT
ST INT
LD RCOM1.Std
ST STD
LD RCOM1.Minu
ST MINU
LD RCOM1.Sek
ST SEK
LD RCOM1.EvtP
ST EVTP
LD RCOM1.RDY
ST RDY

Note: In IL, the function call has to be performed in one line.

Function call in ST

ERR:=RCOM1.ERR;
ERR_NR:=RCOM1.ERR_NR;
ZYKL:=RCOM1.Zykl;
INT:=RCOM1.INT;
STD:=RCOM1.Std;
MINU:=RCOM1.Minu;
SEK:=RCOM1.Sek;
EVTP:=RCOM1.EvtP;
RDY:=RCOM1.RDY;
**INITIALIZE 07 KP 90 R202**

This block is used for initializing the 07 KP 90 R202. All necessary network parameters are defined here. At the RCOM master and every RCOM slave, the initialization must be performed before starting other blocks.

The block enters the network parameters and then starts the coupler initialization.

The block also supplies the 07 KP 90 R202 cycle counter value (life identifier) and the RCOM system time until input FREI is "TRUE". Additionally, the current event queue level (i.e. the number of data records which are still queued) is displayed at output EvtP of RCOM slaves.

The RCOM+ block can be used with the communication module 07 KP 90 beginning with index b.

---

**Block type**

Function block

**Parameters**

- **Instance**: RCOM_PL instance name
- **FREI**: BOOL enabling of the block processing with FALSE/TRUE edge
- **NET**: INT always 0
- **NOD**: INT RCOM master: 0, RCOM slave: number of the slave
- **Baud**: INT baud rate for RCOM interface
- **Pari**: INT parity: 0: none, 1: odd, 2: even
- **Dupl**: INT 0: half duplex, 1: full duplex
- **TLS**: INT time for carrier stabilization
- **Cdly**: INT waiting time for carrier stop
- **CTO**: INT character delay time
- **TAT**: INT reaction time for response
- **Mdm**: INT modem type
- **Retr**: INT number of repetitive attempts
- **MaxP**: INT maximum number of event polls for a slave in a cycle
- **NoPr**: INT number of preambles in the protocol
- **Dbg**: INT debug step
- **RDY**: BOOL ready message
- **ERR**: BOOL error output
- **ERR_NR**: INT error number
- **ZykI**: INT cycle counter value
- **NT**: BOOL new time available
- **Std**: INT time, hours
- **Minu**: INT time, minutes
- **Sek**: INT time, seconds
- **EvtP**: INT number of events in the event queue
Description

In the RCOM+ block the internal variable RCOM_typ is set to 1. At RCOM, RCOM_typ = 0.

Note: All block parameters (except CTO) are identical to the RCOM block parameters.

Switching between RCOM and RCOM+

Switching between RCOM and RCOM+ is performed by planning the blocks RCOM or RCOM+.

Setting the “intermission” for RCOM+

The “intermission” is set with the CTO parameter (character timeout) at the RCOM+ block. The CTO parameter is transferred as a number of characters. In such a way, the following calculation results for intermission duration:

\[
\text{time} [\text{ms}] = CTO [\text{char}] \times D 10 \times [\text{bit/char}] \times 1000 \times [\text{ms/s}] / \text{baud} [\text{bit/s}]
\]

The minimum ascertainable time in the RCOM coupler is 10 ms. Therefore the following calculation for the parameter CTO results:

\[
CTO [\text{char}] = \frac{\text{time} [\text{ms}] \times \text{baud} [\text{bit/s}]}{10 \times [\text{bit/char}] \times 1000 \times [\text{ms/s}]}
\]

Example:

- baud = 1200
  \[\text{CTO} = 30 \times 1200/10000 = 3.6 \rightarrow 4\]
- baud = 2400
  \[\text{CTO} = 30 \times 2400/10000 = 7.2 \rightarrow 8\]
- baud = 9600
  \[\text{CTO} = 30 \times 9600/10000 = 28.8 \rightarrow 29\]
- baud = 19200
  \[\text{CTO} = 30 \times 19200/10000 = 57.6 \rightarrow 58\]

Representation in the operator

When showing the RCOM parameters in the terminal (enter rcom<CR>), the parameters RCOM_typ and DIGI_time are displayed.

rcom typ: 1 - RCOM+ / 0 - RCOM
digi time: 0 - Standard

Error messages

While checking the parameters transferred with the RCOM or RCOM+ blocks, the parameters RCOM_typ and DIGI_time are checked.

If the parameter RCOM_typ is <0 or >1, an error is reported:

\[\text{E_NET_PARAM} + 20 \quad 2014_{\text{hex}} = 8212_{\text{dez}}\]

Initialization error: wrong RCOM type

If the parameter DIGI_time for RCOM+ is <0, the following error is reported:

\[\text{E_NET_PARAM} + 21 \quad 2015_{\text{hex}} = 8213_{\text{dez}}\]

Initialization error: wrong value for DIGI_time

Transmission delay using the DIGI_time parameter

Using the DIGI_time parameter, the transmission can be delayed, i.e. it is waited briefly between the individual telegram characters. Internal, the waiting time is realized using a "FOR loop with a variable number of runs".

\[
\text{for} \ (i=0; \ i<\text{DIGI_time}; \ i++)
\]

The following times and delays resulted during the test:

\[
\begin{align*}
\text{t_char} & : \text{time for 1 character [ms]} \\
\text{nt} & : \text{delay [ms]}
\end{align*}
\]

Using COMWATCH, the times have been measured for the direct connection of a T200 / 07 KP 64 (MASTER) and a AC31 / 07 KT 93 / 07 KP 90 R2 with pari=1 (11 bit frame).

The minimum time for 1 character can be calculated with:

\[
\text{t} [\text{ms}] = 11 \times [\text{bit/s}] \times 1000 \times [\text{ms/s}] / \text{baud} [\text{bit/s}]
\]

for:

- 1200 baud \(\rightarrow t = 11 \times 1000/1200 = 9.167 \text{ ms}\)
- 2400 baud \(\rightarrow t = 11 \times 1000/2400 = 4.583 \text{ ms}\)
- 9600 baud \(\rightarrow t = 11 \times 1000/9600 = 1.146 \text{ ms}\)
- 19200 baud \(\rightarrow t = 11 \times 1000/19200 = 0.573 \text{ ms}\)

Referring to the table, the delay does not have an effect until the minimum time for a character is reached.

Changing the DIGI_time parameter by 50 causes a delay of approx. 0,5-0,6 ms.
<table>
<thead>
<tr>
<th>DIGI_time</th>
<th>19200 baud</th>
<th>9600 baud</th>
<th>2400 baud</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t_char [ms]</td>
<td>nt [ms]</td>
<td>t_char [ms]</td>
</tr>
<tr>
<td>0</td>
<td>0.6</td>
<td>0.0</td>
<td>1.2</td>
</tr>
<tr>
<td>50</td>
<td>0.7</td>
<td>0.1</td>
<td>1.2</td>
</tr>
<tr>
<td>100</td>
<td>1.2</td>
<td>0.6</td>
<td>1.3</td>
</tr>
<tr>
<td>150</td>
<td>1.8</td>
<td>1.2</td>
<td>1.8</td>
</tr>
<tr>
<td>200</td>
<td>2.3</td>
<td>1.7</td>
<td>2.4</td>
</tr>
<tr>
<td>250</td>
<td>2.9</td>
<td>2.3</td>
<td>2.9</td>
</tr>
<tr>
<td>300</td>
<td>3.4</td>
<td>2.8</td>
<td>3.4</td>
</tr>
<tr>
<td>350</td>
<td>4.0</td>
<td>3.4</td>
<td>4.0</td>
</tr>
<tr>
<td>400</td>
<td>4.6</td>
<td>4.0</td>
<td>4.5</td>
</tr>
<tr>
<td>450</td>
<td>5.1</td>
<td>4.5</td>
<td>5.1</td>
</tr>
<tr>
<td>500</td>
<td>5.7</td>
<td>5.1</td>
<td>5.7</td>
</tr>
<tr>
<td>550</td>
<td>6.2</td>
<td>5.6</td>
<td>6.2</td>
</tr>
<tr>
<td>600</td>
<td>6.7</td>
<td>6.1</td>
<td>6.8</td>
</tr>
<tr>
<td>650</td>
<td>7.3</td>
<td>6.7</td>
<td>7.3</td>
</tr>
<tr>
<td>700</td>
<td>7.8</td>
<td>7.2</td>
<td>7.8</td>
</tr>
<tr>
<td>750</td>
<td>8.4</td>
<td>7.8</td>
<td>8.4</td>
</tr>
<tr>
<td>800</td>
<td>8.9</td>
<td>8.3</td>
<td>8.9</td>
</tr>
<tr>
<td>850</td>
<td>9.5</td>
<td>8.9</td>
<td>9.5</td>
</tr>
<tr>
<td>900</td>
<td>10.1</td>
<td>9.5</td>
<td>10.1</td>
</tr>
<tr>
<td>950</td>
<td>10.6</td>
<td>10.1</td>
<td>10.6</td>
</tr>
<tr>
<td>1000</td>
<td>11.2</td>
<td>10.6</td>
<td>11.2</td>
</tr>
</tbody>
</table>

With the primary formula for the calculation of a FOR loop time duration it was assumed, that one run takes approx. 20 processor pulses. The measuring results show, that one run takes approx. 180 processor pulses. For a 16 MHz processor a processor pulse takes 62,5 ns. Therefore one run takes approx. 62,5 D 180 = 11250 ns (0,01125 ms).

Transfer of the DIGI_time parameter WOS

At the Advant Controller 31 the transfer is performed with the WOS block (write word).

At input E1 the corresponding DIGI_time is specified.

The address specifications must be observed under all circumstances. In the PLC program, the WOS block has to be used directly after the RCOM+ block.
Function call in IL

CAL RCOM_PL1(FREI := FREI, NET := NET, NOD := NOD, Baud := BAUD, Pari := PARI,
Dupl := DUPL, TLS := TLS, CDly := CDLY, CTO := CTO, TAT := TAT, Mdm := MDM,
Retr := RETR, MaxP := MAXP, NoPr := NOPR, Dbg := DBG)

LD RCOM_PL1.ERR
ST ERR
LD RCOM_PL1.ERR_NR
ST ERR_NR
LD RCOM_PL1.Zykl
ST ZYKL
LD RCOM_PL1.NT
ST INT
LD RCOM_PL1.Std
ST STD
LD RCOM_PL1.Minu
ST MINU
LD RCOM_PL1.Sek
ST SEK
LD RCOM_PL1.EvtP
ST EVTP
LD RCOM_PL1.RDY
ST RDY

Note: In IL, the function call has to be performed in one line.

Function call in ST

RCOM_PL1(FREI := FREI, NET := NET, NOD := NOD,
Baud := BAUD, Pari := PARI, Dupl := DUPL,
TLS := TLS, CDly := CDLY, CTO := CTO,
TAT := TAT, Mdm := MDM, Retr := RETR,
MaxP := MAXP, NoPr := NOPR, Dbg := DBG);
ERR:=RCOM_PL1.ERR;
ERR_NR:=RCOM_PL1.ERR_NR;
ZYKL:=RCOM_PL1.Zykl;
INT:=RCOM_PL1.NT;
STD:=RCOM_PL1.Std;
MINU:=RCOM_PL1.Minu;
SEK:=RCOM_PL1.Sek;
EVTP:=RCOM_PL1.EvtP;
RDY:=RCOM_PL1.RDY;
READ DATA FROM RCOM SLAVE

With this block it is possible to read data records from a RCOM slave. The block can only be used by a master; slaves can not read data from the master.

The READ_S block must be implemented in the called slave.

### Block type

Function block

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>READ</td>
<td>instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>enabling of the block processing with FALSE/TRUE edge</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>number of the called RCOM slave</td>
</tr>
<tr>
<td>IDT</td>
<td>INT</td>
<td>number of the data record to be read</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>number of the words to be read</td>
</tr>
<tr>
<td>AMW</td>
<td>INT</td>
<td>start flag of the flag area where the read words are stored</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>BOOL</td>
<td>error number</td>
</tr>
</tbody>
</table>

### Description

The function block READ is used to read data from a RCOM slave. For reading, a READ job is sent to the communication partner via the 07 KP 90 R202 communication module. As a result, the communication partner sends an acknowledgement including the desired data.

The read data are stored in a flag area to be specified.

Before data can be transmitted using the READ function block, the 07 KP 90 R202 communication module must have been initialized (RCOM function block). Additionally, a normalization is necessary before transmitting.

If a job is started (FALSE/TRUE edge at input FREI), the data at the inputs NET, NOD, IDT, LEN and AMW must not be changed until the job is completed (RDY = TRUE).

With a FALSE/TRUE edge at input FREI the reading operation is performed once.

If the reading operation has not yet been completed, further FALSE/TRUE edges at input FREI are ignored. After the reading operation has been completed, the outputs RDY, ERR and ERR_NR are set.

Using the inputs NET and NOD, it is specified which RCOM subscriber should provide the data. IDT specifies the data record number within this subscriber, LEN specifies the number of transmitted words.

The reading operation was successful, if RDY = TRUE and ERR = FALSE. For RDY = TRUE and ERR = TRUE, an error occurred. The error type can be determined using the output ERR_NR. RDY, ERR and ERR_NR are reset with a falling edge at input FREI.

**FREI**

**BOOL**

With a FALSE/TRUE edge at input FREI the reading operation is performed once. With this FALSE/TRUE edge, the outputs RDY, ERR and ERR_NR are also set to zero. If the reading operation has not yet been completed (RDY = FALSE), a further FALSE/TRUE edge at input FREI is ignored.

**NET**

**INT**

Must always be zero.

**NOD**

**INT**

At input NOD, the number of the called RCOM subscriber is specified. Only slaves can be called (NOD=1...254).
READ DATA FROM RCOM SLAVE

IDT   INT
Number of the data record to be read

LEN   INT
At the input, the number of words to be read is specified. The number must be even-numbered.

AMW   INT
At input AMW, the first flag of the flag area is specified where the data should be stored.

RDY   BOOL
Output RDY indicates that the reading operation has been completed. This output has always to be considered together with the output ERR. The following applies:
RDY = TRUE and ERR = FALSE: The reading operation has been completed. The variable areas have been read by the RCOM slave and transferred to the data record.
RDY = TRUE and ERR = TRUE: An error occurred during the reading operation. In this case, the error message at output ERR_NR can be evaluated. No data were transferred to the data record.

Function call in IL
CAL READ1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, LEN := LEN, AWM := AWM)
LD READ1.ERR
ST ERR
LD READ1.ERR_NR
ST ERR_NR
LD READ1.RDY
ST RDY

Note: In IL, the function call has to be performed in one line.

After a reading operation, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.
If the reading operation has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

ERR   BOOL
Output ERR indicates whether an error occurred during the reading operation. Output ERR has always to be considered together with the output RDY.
If an error occurred during the reading operation, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.

ERR_NR   INT
At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.
The error numbers are further explained in appendix "Error codes".

Function call in ST
READ1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, LEN := LEN, AWM := AWM);
ERR:=READ1.ERR;
ERR_NR:=READ1.ERR_NR;
RDY:=READ1.RDY;
PROVIDE DATA FOR READ JOB

This block is the partner in the slave for a READ block in the master. The block provides the data which the master wants to read.

The block can only be used in a slave.

Block type

Function block

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>READ_S instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL enabling of the block processing</td>
</tr>
<tr>
<td>NET</td>
<td>INT always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT always zero</td>
</tr>
<tr>
<td>IDT</td>
<td>INT number of the called data record</td>
</tr>
<tr>
<td>AMW</td>
<td>INT start flag of the flag area to be transmitted</td>
</tr>
<tr>
<td>NEW</td>
<td>BOOL data have been read from master</td>
</tr>
</tbody>
</table>

Description

The function block provides the data which are read from the master using a READ block.

The data are provided to the slave coupler with an incoming READ job from the specified flag area.

If FREI = zero or the slave was not normalized, the block rejects reading jobs (answer 'application part not ready').

The data record number must match with the number in the READ block.

The sender of the reading job is specified via the inputs NET and NOD. This is always the master of a RCOM network, therefore NET and NOD must always be zero.

FREI

At input FREI, the data record reading is enabled. If FREI=FALSE, the incoming reading jobs are returned to the master with an error message. Doing so, the RCOM slave is able to prevent data reading.

NET

Must always be zero.

NOD

Must always be zero.

IDT

At input IDT, the data record number is specified.

AMW

At input AMW, the first flag where data are read out is specified.

NEW

This output is set to “1”, if the master has read the data record. Due to this, the PLC is able to detect the data transferring by the master. During the next PLC cycle, the input is reset.
Function call in IL

CAL READ_S1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, AWM := AWM)

LD READ_S1.NEW

ST NEW

Note: In IL, the function call has to be performed in one line.

Function call in ST

READ_S1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, AWM := AWM);
NEW:=READ_S1.NEW;
This block is used to receive a data record sent with TRANSM. The block can be used in the slave as well as in the master (reception of events).

**Block type**

- Function block

**Parameters**

- **Instance**: RECV (instance name)
- **FREI**: BOOL (enabling of the block processing)
- **NET**: INT (always zero)
- **NOD**: INT (number of the sending RCOM subscriber (0 for the master))
- **IDT**: INT (number of the called data record)
- **AMW**: INT (start flag of the flag area to be transmitted)
- **NEW**: INT (new values available)

**Description**

The function block is used to receive data which were sent with TRANSM.

As soon as new data are available for the called data record, NEW becomes "TRUE" for one PLC cycle duration. Then, the new data are provided in the specified flag area.

The data record number (IDT) must match with the number in the TRANSM block.

If FREI = zero or the slave was not normalized, the block rejects writing jobs (answer 'application part not ready').

The sender of the writing job is specified via the inputs NET and NOD. Data records for a slave are always sent by the master, therefore NET and NOD must be zero. Data records for the master (events) can be sent by different slaves, therefore NOD must be specified here. Due to this, the 07 KP 90 R202 in the master is able to distribute data records correctly.

**FREI**: BOOL

At input FREI, the data record reception is enabled. If FREI=FALSE, the incoming data records are returned with an error message. If the processing of new data takes quite long time, the PLC program is able to set FREI to zero.

**NET**: INT

Must always be zero.

**NOD**: INT

At input NOD, the number of the sending RCOM subscriber is specified. For the usage in a slave, NOD is always 0 (data records can only be sent by the master). For the usage in the master, the number of the sending slave must be specified.

**IDT**: INT

At input IDT, the data record number is specified.

**AMW**: INT

At input AMW, the first flag is specified where data are stored. Beginning with this flag, LEN flags are written.

**NEW**: BOOL

This output becomes "TRUE", as soon as a new data record is available. Now the PLC program is able to process the values. The RECV block resets NEW with the next PLC cycle.
RECEIVE DATA FROM RCOM PARTNER

**Function call in IL**

CAL RECV1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, AWM := AWM)

LD RECV1.NEW

ST NEW

**Function call in ST**

RECV1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, AWM := AWM);

NEW:=RECV1.NEW;

Note: In IL, the function call has to be performed in one line.
This block sends system services directly to the RCOM coupler.

### Diagram

```
  _____________  
SYS_S         |  SYS_S
  |__FREI_______|  |__RDY_______|
     0___NET___|  1___NOD___|  ERR______|  ERR_NR____|
      _____________  
SYS_S         |  SYS_S
  |__SYS_S1_____|
```

### Block type

**Function block**

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>SYS_S</td>
<td>instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>enabling of the block processing</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>number of the sending RCOM subscriber (0 for the master)</td>
</tr>
<tr>
<td>ID</td>
<td>INT</td>
<td>job identifier</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
<td>error number</td>
</tr>
</tbody>
</table>

### Description

This block sends system services directly to the RCOM coupler.

**FREI**

A FALSE/TRUE edge at input FREI starts a system service. Further edges are ignored until RDY has become "TRUE".

**NET**

Must always be zero.

**NOD**

Number of the slave to be started. For NOD=255 all slaves are called.

**ID**

At this input, the service identifier is specified which has the following significances:

- ID = 3D (3H): cold start (COLDST)
- ID = 4D (4H): warm start (WARMST)
- ID = 8D (8H): normalization (NORMAL)
- ID = 13D (D1H): perform event poll (POLL)
- ID = 241D (F1H): dial (DIAL) communication partner
- ID = 242D (F2H): hang up phone (HANGUP)

**RDY**

Output RDY indicates that the warm start has been completed. This output has always to be considered together with the output ERR. The following applies:

RDY = TRUE and ERR = FALSE: The warm start has been completed. The desired slaves have been reset.

RDY = TRUE and ERR = TRUE: An error occurred during the warm start. In this case, the error message at output ERR_NR can be evaluated.

After a warm start, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the warm start has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.
ERR
BOOL
Output ERR indicates whether an error occurred during the warm start. Output ERR has always to be considered together with the output RDY.

If an error occurred during the warm start, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.

ERR_NR
INT
At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are further explained in appendix “Error codes”.

Function call in IL

CAL SYS_S1(FREI := FREI, NET := NET, NOD := NOD)
LD SYS_S1.ERR
ST ERR
LD SYS_S1.ERR_NR
ST ERR_NR
LD SYS_S1.RDY
ST RDY

Note: In IL, the function call has to be performed in one line.

Function call in ST

SYS_S1(FREI := FREI, NET := NET, NOD := NOD);
ERR:=SYS_S1.ERR;
ERR_NR:=SYS_S1.ERR_NR;
RDY:=SYS_S1.RDY;
SEND DATA TO RCOM PARTNER

This block sends a data record to the RCOM partner. The block can be used in the master as well as in the slave (for event-controlled transmission).

In the receiver, for the same data record number, the RECV block must be implemented.

In slaves, the block can be used to initiate events. The data record is stored with a timing mark in the 07 KP 90 R202 and sent to the master with the next event poll. The timing mark is transmitted in the last two data words. Therefore, the block TRANSM can transmit only 14 data words at once, if used in a slave.

Block type

Function block

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>TRANSM</th>
<th>instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>enabling of the block processing with FALSE/TRUE edge</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
<td>always zero</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
<td>number of the called RCOM subscriber</td>
</tr>
<tr>
<td>IDT</td>
<td>INT</td>
<td>number of the called data record</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>number of the words to be transmitted</td>
</tr>
<tr>
<td>AMW</td>
<td>INT</td>
<td>start flag of the flag area to be transmitted</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>error output</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
<td>error number</td>
</tr>
</tbody>
</table>

Description

The function block is used to send a data record to a RCOM communication partner.

If used in a RCOM master, a send job including the user data is sent to the called slave. As a result, the slave sends an acknowledgement.

If the block is used in a slave, the data record is transferred to the event queue and transmitted to the master with the next event poll.

At the receiver, the transmitted data appear at an accordingly programmed RECV block combination element.

If a job is started (FALSE/TRUE edge at input FREI), the data at the inputs NET, NOD, IDT, LEN and AMW must not be changed until the job is completed (RDY = TRUE).

Before data can be transmitted using the TRANSM function block, the 07 KP 90 R202 communication module must have been initialized (RCOM function block). Additionally, a normalization is necessary before transmitting.

With a FALSE/TRUE edge at input FREI the writing operation is performed once. With this edge, the outputs RDY, ERR and ERR_NR are also set to zero.

If the writing operation has not yet been completed, further FALSE/TRUE edges at input FREI are ignored. After the writing operation has been completed, the outputs RDY, ERR and ERR_NR are set. RDY, ERR and ERR_NR are reset with a falling edge at input FREI.

Using the inputs NET and NOD, it is specified which RCOM subscriber should receive the data. IDT specifies the data record number within this subscriber, LEN specifies the number of transmitted words. LEN must be even. If the block is used in a slave (event transmission), NET and NOD have to be set to zero. LEN may be max. 14, because in addition to the user data the timing mark is stored.

The writing operation was successful, if RDY = TRUE and ERR = FALSE. For RDY = TRUE and ERR = TRUE, an error occurred. The type of error can be determined using the output ERR_NR.
**FREI**

**BOOL**

With a FALSE/TRUE edge at input FREI a writing operation is performed once. With this FALSE/TRUE edge, the outputs RDY, ERR and ERR_NR are also set to zero. If the writing operation has not yet been completed (RDY = FALSE), a further FALSE/TRUE edge at input FREI is ignored.

**NET**

**INT**

Must always be zero.

**NOD**

**INT**

At input NOD, the number of the called RCOM subscriber is specified. For a transmission from master to slave: slave number for event transmission: zero.

**IDT**

**INT**

At input IDT, the number of the data record to be transmitted is specified.

**LEN**

**INT**

At the input, the number of words to be transmitted is specified. For using TRANSM in slaves (event transmission), LEN must be < 14. LEN must be even.

**AMW**

**INT**

At input AMW, the first flag of the flag area to be transmitted is specified. Beginning with this flag, LEN flags are transmitted.

**RDY**

**BOOL**

Output RDY indicates that the writing operation has been completed. This output has always to be considered together with the output ERR. The following applies:

RDY = TRUE and ERR = FALSE: The writing operation has been completed. The desired variable areas have been sent to the RCOM partner.

RDY = TRUE and ERR = TRUE: An error occurred during the writing operation. In this case, the error message at output ERR_NR can be evaluated.

After a writing operation, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the writing operation has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

**ERR**

**BOOL**

Output ERR indicates whether an error occurred during the writing. Output ERR has always to be considered together with the output RDY. If an error occurred during the writing operation, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.

**ERR_NR**

**INT**

At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are described in detail in appendix “Error codes”.

### Function call in IL

```
CAL TRANSM1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, LEN := LEN, AWM := AWM)
LD TRANSM1.ERR
ST ERR
LD TRANSM1.ERR_NR
ST ERR_NR
LD TRANSM1.RDY
ST RDY
```

Note: In IL, the function call has to be performed in one line.

### Function call in ST

```
TRANSM1(FREI := FREI, NET := NET, NOD := NOD, IDT := IDT, LEN := LEN, AWM := AWM)
ERR:=TRANSM1.ERR
ERR_NR:=TRANSM1.ERR_NR
RDY:=TRANSM1.RDY
```
With this block, the RCOM master performs a warm start, i.e. all data records are disabled for the event-controlled transmission and data records which exist in the event queue are deleted.

**Block type**

Function block

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WARMST</td>
<td>instance name</td>
</tr>
<tr>
<td>FREI</td>
<td>BOOL</td>
</tr>
<tr>
<td>NET</td>
<td>INT</td>
</tr>
<tr>
<td>NOD</td>
<td>INT</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
</tr>
<tr>
<td>ERR_NR</td>
<td>INT</td>
</tr>
</tbody>
</table>

**FREI**

A FALSE/TRUE edge at input FREI initiates a warm start. Further edges are ignored until RDY has become "TRUE".

**NET**

Must always be zero.

**NOD**

Number of the slave to be started. For NOD=255 all slaves are called.

**RDY**

Output RDY indicates that the warm start has been completed. This output has always to be considered together with the output ERR. The following applies:

RDY = TRUE and ERR = FALSE: The warm start has been completed. The desired slaves have been reset.

RDY = TRUE and ERR = TRUE: An error occurred during the warm start. In this case, the error message at output ERR_NR can be evaluated.

After a warm start, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the warm start has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.

**ERR**

Output ERR indicates whether an error occurred during the warm start. Output ERR has always to be considered together with the output RDY.

If an error occurred during the warm start, RDY = TRUE and ERR = TRUE are set. The error type can be evaluated using the error number at output ERR_NR.

**ERR_NR**

At output ERR_NR an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error numbers are described in detail in appendix "Error codes".

RDY = TRUE: An error occurred during the warm start. In this case, the error message at output ERR_NR can be evaluated.

After a warm start, a falling edge at input FREI sets the outputs RDY, ERR and ERR_NR to zero.

If the warm start has not yet been completed (RDY = FALSE) after a FALSE/TRUE edge at input FREI, a new FALSE/TRUE edge at input FREI is ignored.
### Function call in IL

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>WARMST1(FREI := FREI, NET := NET, NOD := NOD)</td>
</tr>
<tr>
<td>LD</td>
<td>WARMST1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>ERR</td>
</tr>
<tr>
<td>LD</td>
<td>WARMST1.ERR_NR</td>
</tr>
<tr>
<td>ST</td>
<td>ERR_NR</td>
</tr>
<tr>
<td>LD</td>
<td>WARMST1.RDY</td>
</tr>
<tr>
<td>ST</td>
<td>RDY</td>
</tr>
</tbody>
</table>

Note: In IL, the function call has to be performed in one line.

### Function call in ST

```
WARMST1(FREI := FREI, NET := NET, NOD := NOD);
ERR:=WARMST1.ERR;
ERR_NR:=WARMST1.ERR_NR;
RDY:=WARMST1.RDY;
```
Glossary

BOOL
Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT
DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT:</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

DWORD
DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

INT
INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT:</td>
<td>-32768</td>
<td>32767</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

WORD
WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules are to be derived:

- Within functions, global variables can neither be read nor wrote.
- Within functions, absolute operands can neither be read nor wrote.
- Within functions, function blocks must not be called.

Function blocks

Function blocks are subroutines which can have as much inputs, outputs and internal variables as required. They are called from a program or from another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. Therefore, e.g. counters can be realized which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters, a function maximum one, but the output parameters differ syntactically.
- In difference to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
CLOCK S90 4
COLDST S90 6

D
DIAL S90 8

H
HANGUP S90 10

N
NORMAL S90 12

P
POLL S90 14

R
RCOM S90 16
CLOCK 4
COLDST 6
DIAL 8
HANGUP 10
NORMAL 12
POLL 14
RCOM 16
RCOM_PL 20
READ 24
READ_S 26
RECV 28
SYS_S 30
TRANSM 32
WARMST 34

S
SYS_S S90 30

T
TRANSM S90 32

W
WARMST S90 34
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

ARCNET
Function Block Library
90 Series

ARCNET
90 Series
# Contents

The ARCNET Library  .......................................................... 2
  SPECIAL FEATURES OF THE ARCNET LIBRARY .......................... 2
  COMPONENTS OF THE ARCNET LIBRARY ................................ 2

Overview of blocks arranged according to their call names  ............. 3
  INITIALIZATION OF THE ARCNET CONTROLLER  AINIT .................. 4
  RECEIVE ARCNET DATA PACKAGES  AREC ................................. 7
  SEND ARCNET DATA PACKAGES  ASEND(1..16) ......................... 9
  INITIALIZATION OF THE ARCNET COUPLER, VARIABLE  AINIT_V ........ 11
  RECEIVE ARCNET DATA PACKAGES  AREC_V ............................. 16
  SEND ARCNET DATA PACKAGES TO ONE SUBSCRIBER  ASEND_V ........ 19
  READ ARCNET TIMEOUT DATA PACKAGES  ASTO_V ....................... 22
  DRIVER FOR THE 5F PROTOCOL VIA ARCNET  _5F_ARC97 ............. 24

Glossary  ............................................................................... 29

Index  .................................................................................. 31
The ARCNET Library

SPECIAL FEATURES OF THE ARCNET LIBRARY

Note:
The ARCNET communication only runs in the RUN mode of the PLC, but not in the simulation mode.

Alterations compared with the blocks of the EBS operating system of the 07 KT 94 basic unit:

Block: AREC

Instead of n duplications AREC is projected n times. In this case it is important that the blocks are projected directly one behind the other, i.e. not at different positions in the program.

Blocks: ASEND1, ASEND4, ASEND16

The ASEND+ block is replaced by the corresponding ASEND block. Sending the data packages to one (ASEND1), four (ASEND4) or sixteen (ASEND16) subscribers is possible.

COMPONENTS OF THE ARCNET LIBRARY

The following function blocks are contained in the ARCNET library:

<table>
<thead>
<tr>
<th>Group: ARCNET</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AINIT</td>
<td>- Initialization of the ARCNET controller</td>
</tr>
<tr>
<td>AREC</td>
<td>- Receive ARCNET data packages</td>
</tr>
<tr>
<td>ASEND1</td>
<td>- Send ARCNET data packages to one subscriber</td>
</tr>
<tr>
<td>ASEND4</td>
<td>- Send ARCNET data packages to maximum of 4 subscribers</td>
</tr>
<tr>
<td>ASEND16</td>
<td>- Send ARCNET data packages to maximum of 16 subscribers</td>
</tr>
<tr>
<td>ASEND8</td>
<td>- ARCNET sending block for bytes for a maximum of 1 receiver</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group: ARCNET new</th>
</tr>
</thead>
<tbody>
<tr>
<td>AINIT_V</td>
</tr>
<tr>
<td>ASEND_V</td>
</tr>
<tr>
<td>AREC_V</td>
</tr>
<tr>
<td>ASTO_V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gruppe</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>_5F_ARC97</td>
<td>- Driver for the 5F protocol via ARCNET</td>
</tr>
</tbody>
</table>

Detailed descriptions of the blocks are in the following sections.
Overview of blocks arranged according to their call names

Character descriptions:

FBmV … Function block with historical values
FBoV … Function block without historical values

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>_5F_ARC97</td>
<td>FBmV</td>
<td>Driver for the 5F protocol via ARCNET</td>
<td>25</td>
</tr>
<tr>
<td>AINIT</td>
<td>FBmV</td>
<td>Initialization of the ARCNET controller</td>
<td>4</td>
</tr>
<tr>
<td>AINIT_V</td>
<td>FBmV</td>
<td>Initialisierung des ARCNET-Kopplers variabel</td>
<td>12</td>
</tr>
<tr>
<td>AREC</td>
<td>FBmV</td>
<td>Receive ARCNET data packages</td>
<td>8</td>
</tr>
<tr>
<td>AREC_V</td>
<td>FBoV</td>
<td>Receive ARCNET data packages</td>
<td>17</td>
</tr>
<tr>
<td>ASEND(1..16)</td>
<td>FBmV</td>
<td>Send ARCNET data packages</td>
<td>10</td>
</tr>
<tr>
<td>ASEND_V</td>
<td>FBoV</td>
<td>Send ARCNET data packages to one subscriber</td>
<td>20</td>
</tr>
<tr>
<td>ASTO_V</td>
<td>FBoV</td>
<td>Read ARCNET timeout data packages</td>
<td>23</td>
</tr>
</tbody>
</table>
INITIALIZATION OF THE ARCNET CONTROLLER

The operating system initializes the ARCNET controller in the following way:

- Interrupt after receiving or sending a data package
- Only short packages (a short package = 256 bytes)
- Reception of broadcasts

Available as of PLC runtime system: V4.0
Included in library: ABB-BIB4.LIB, ARCNET_S90_V41.LIB

Block data

Parameter

<table>
<thead>
<tr>
<th>Instance</th>
<th>AINIT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>Tout</td>
<td>INT</td>
<td>Timeout in ms when sending data packages</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Initialization terminated</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error has occurred</td>
</tr>
<tr>
<td>NODE</td>
<td>INT</td>
<td>Own node number (station address)</td>
</tr>
<tr>
<td>STAT</td>
<td>INT</td>
<td>Status register of the ARCNET controller</td>
</tr>
<tr>
<td>DIAG</td>
<td>INT</td>
<td>Diagnosis register of the ARCNET controller</td>
</tr>
<tr>
<td>TOS</td>
<td>BOOL</td>
<td>Timeout has occurred during send operation</td>
</tr>
<tr>
<td>TOND</td>
<td>INT</td>
<td>Node number, timeout data packages</td>
</tr>
<tr>
<td>TOJN</td>
<td>INT</td>
<td>Job number, timeout data packages</td>
</tr>
<tr>
<td>LEV</td>
<td>INT</td>
<td>Level of the send buffer</td>
</tr>
<tr>
<td>RECO</td>
<td>BOOL</td>
<td>Network reconfiguration is running (after loss of token)</td>
</tr>
</tbody>
</table>

The outputs STAT, DIAG, TOS, TOND, TOJN, LEV and RECO are updated after a successful initialization with each block call.

Description

The operating system initializes the ARCNET controller in the following way:

- Interrupt after receiving or sending a data package
- Only short packages (a short package = 256 bytes)
- Reception of broadcasts

FREI  BOOL
A TRUE signal at input FREI causes the processing of the block.

TO  INT
The timeout waiting time for sending data packages is specified in ms at input TO. If a data package cannot be sent within this period, the sending operation of this data package is aborted and the package is lost. The output TOS indicates the loss of the package.

DONE  BOOL
The output DONE indicates that the initialization has been terminated. This output has always to be considered together with output ERR.

The following applies:

DONE = TRUE and ERR = FALSE:
The initialization has been terminated. No error has occurred.
DONE = TRUE and ERR = TRUE:

An error has occurred during initialization. The ARCNET controller has not responded within a period of 100 ms.

ERR BOOL
The output ERR indicates that the ARCNET controller has not responded after an initialization command within a period of 100 ms. This output has always to be considered together with output DONE. If an error has occurred, the following applies:
DONE = TRUE and ERR = TRUE.

NODE INT
The output NODE indicates the own node number (station address) after a successful initialization.

STAT INT
The output STAT indicates the content of the status register of the ARCNET controller after a successful initialization.

DIAG INT
The output DIAG indicates the content of the diagnosis register of the ARCNET controller after a successful initialization.

TOS BOOL
The output TOS indicates that sending of a data package was not possible within the timeout period (input TO) and that the data package is lost. Both the node number and the job number of the lost data package are available at the outputs TOND and TOJN, respectively.

TOND INT
The output TOND indicates the node number of the lost data package after the timeout period has elapsed.

TOJN INT
The output TOJN indicates the job number of the lost data package after the timeout period has elapsed.

LEV INT
The output LEV indicates the level of the sending buffer after a successful initialization.

RECO BOOL
The output RECO indicates that the network is reconfiguring itself (RECO = TRUE) after a loss of token. The value RECO = FALSE indicates that the reconfiguring procedure has been terminated.

The outputs STAT, DIAG, TOS, TOND, TOJN, LEV and RECO are updated after a successful initialization with each block call.

The inputs and outputs can neither be duplicated nor inverted.
INITIALIZATION OF THE ARCNET CONTROLLER

Function call in IL

CAL AINIT1(FREI := AINIT_FREI,
           TOut := AINIT_TOut)
LD AINIT1.ERR
ST AINIT_ERR
LD AINIT1.NODE
ST AINIT_NODE
LD AINIT1.STAT
ST AINIT_STAT
LD AINIT1.DIAG
ST AINIT_DIAG
LD AINIT1.TOS
ST AINIT_TOS
LD AINIT1.TOND
ST AINIT_TOND
LD AINIT1.TOJN
ST AINIT_TOJN
LD AINIT1.LEV
ST AINIT_LEV
LD AINIT1.RECO
ST AINIT_RECO
LD AINIT1.DONE
ST AINIT_DONE

Note: The function call in IL has to be performed in one line.

Function call in ST

AINIT1(FREI := AINIT_FREI, TOut := AINIT_TOut);
AINIT_ERR:=AINIT1.ERR;
AINIT_NODE:=AINIT1.NODE;
AINIT_STAT:=AINIT1.STAT;
AINIT_DIAG:=AINIT1.DIAG;
AINIT_TOS:=AINIT1.TOS;
AINIT_TOND:=AINIT1.TOND;
AINIT_TOJN:=AINIT1.TOJN;
AINIT_LEV:=AINIT1.LEV;
AINIT_RECO:=AINIT1.RECO;
AINIT_DONE:=AINIT1.DONE;
RECEIVE ARCNET DATA PACKAGES

The block reads the receiving buffer in the correct order. The first data package which agrees with its planned inputs is taken from the receiving buffer and stored in the user data range.

Block data

Available as of PLC runtime system: V4.0
as of V4.15
Remarks:
NO := -1 is possible
Included in library: ABB-BIB4.LIB
ARCNET_S90_V41.LIB

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>AREC</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0</td>
<td>INT</td>
<td>Node number (station address) of the sender</td>
</tr>
<tr>
<td>D0</td>
<td>INT</td>
<td>DIN identification</td>
</tr>
<tr>
<td>J0</td>
<td>INT</td>
<td>Job number</td>
</tr>
<tr>
<td>L0</td>
<td>INT</td>
<td>Number of words of the user data to be received</td>
</tr>
<tr>
<td>MW</td>
<td>INT</td>
<td>First word variable, as of the received user data are stored</td>
</tr>
<tr>
<td>UJOB</td>
<td>BOOL</td>
<td>Unknown job received</td>
</tr>
<tr>
<td>JR0</td>
<td>BOOL</td>
<td>Job received</td>
</tr>
</tbody>
</table>

Description

The operating system reads the received data packages from the ARCNET controller interrupt-controlled and stores them into a storage buffer. The size of the storage buffer is 63 data packages.

The block reads the receiving buffer in the correct order. The first data package which agrees with its planned inputs is taken from the receiving buffer and stored in the user data range.

With the same job the receiver gets user data which belong together logically. For an unique identification of a job the following parameters are used:

- Node number of the sender,
- DIN identification and
- Job number.

The parameter values of the received data packages are compared to the parameter values of the planned job. If both values are equal, the user data of the data package beginning from the word variable MW are stored continuously and output JR0 is set to TRUE.

If no data package from the receiving buffer can be assigned to a planned job, the output JR0 which belongs to this planned job is set to FALSE.

The output UJOB indicates that a data package existing in the receiving buffer could not be assigned to a job.

For each received data package one call of the block AREC must be planned in the user program.

All blocks has to be called in each cycle. Either one job or one UJOB or none of both is indicated per cycle.
N0 \hspace{1cm} \text{INT}

The node number (station address) of the sender is specified for the planned job at input NO. The following applies: 0 ≤ N0…Nn-1 ≤ 255

As of PLC runtime system V4.15 the entry NO:=-1 is possible. With this setting, a telegram characterized by JO and DIN, is accepted by all station addresses.

D0 \hspace{1cm} \text{INT}

The DIN identification is specified for the planned job at input D0. The following applies: D0 = 127 (7FH) not allowed: 111 (6FH)/95 (5FH)/79 (4FH)/63 (3FH)

J0 \hspace{1cm} \text{INT}

The job number is specified for the planned job at input J0. The following applies: -32767 ≤ J0 ≤ +32767

L0 \hspace{1cm} \text{INT}

The number of user data words is specified for the planned job at input L0. The following applies: 0 ≤ L0 ≤ 125

MW \hspace{1cm} \text{INT}

The starting word flag of the user data is specified for the planned job at input MW.

UJOB \hspace{1cm} \text{BOOL}

Output UJOB indicates that a data package was stored in the receiving buffer which could not be assigned to any planned job. The following applies:

UJOB = TRUE: Unknown job received
UJOB = FALSE: No unknown job received

JR0 \hspace{1cm} \text{BOOL}

Output JR0 (job received) indicates that a data package from the receiving buffer has been assigned to this job. The user data of the data package are stored continuously beginning from the word variable MW. The following applies:

JR0 = TRUE: Job received
JR0 = FALSE: No job received

Note: The inputs and outputs can neither be duplicated nor inverted.

Function call in IL

CAL AREC1(N0 := AREC_N0, D0 := AREC_D0, J0 := AREC_J0, L0 := AREC_L0, MW0 := MW0)

LD AREC1.JR0
ST AREC_JR0
LD AREC1.UJOB
ST AREC_UJOB

Note: In IL, the function call has to be performed in one line.

Function call in ST

AREC1(N0 := AREC_N0, D0 := AREC_D0, J0 := AREC_J0, L0 := AREC_L0, MW0 := MW0);
AREC_JR0:=AREC1.JR0;
AREC_UJOB:=AREC1.UJOB;
SEND ARCNET DATA PACKAGES

The `ASEND(1..16)` block is used to send off jobs (data packages) via the ARCNET network.

The planned jobs are stored in a sending buffer. From here, they are transferred to the ARCNET controller for sending off.

The `ASEND` number indicates the maximum number of node numbers (receiver). The following `ASEND` blocks are available:

- `ASEND1`: ARCNET sending block for a maximum of 1 receiver
- `ASEND4`: ARCNET sending block for a maximum of 4 receivers
- `ASEND16`: ARCNET sending block for a maximum of 16 receivers
- `ASENDB`: ARCNET sending block for bytes for a maximum of 1 receiver

Block data

Available as of PLC runtime system: V4.0
Included in library:
- `ABB-BIB4.LIB`
- `ARCNET_S90_V41.LIB`

Remarks:

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>ASEND(1..16,B)</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enable: Send job</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE: do not send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE/TRUE edge: send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TRUE: send only if user data have changed</td>
</tr>
<tr>
<td>D0</td>
<td>INT</td>
<td>DIN identification</td>
</tr>
<tr>
<td>J0</td>
<td>INT</td>
<td>Job number</td>
</tr>
<tr>
<td>L0</td>
<td>INT</td>
<td>Number of user data words to be sent</td>
</tr>
<tr>
<td>n</td>
<td>INT</td>
<td>Number of node numbers</td>
</tr>
<tr>
<td>N0..Nn-1</td>
<td>INT</td>
<td>Node number (station address) of the receiver</td>
</tr>
<tr>
<td>MW</td>
<td>INT</td>
<td>First word variable of the user data words</td>
</tr>
<tr>
<td>STA</td>
<td>BOOL</td>
<td>Job has been stored in the sending buffer</td>
</tr>
</tbody>
</table>

Description

The `ASEND(1..16)` block is used to send off jobs (data packages) via the ARCNET network.

If several jobs have to be sent several `ASEND(1..16)` blocks must be planned.

The ARCNET controller must have been initialized before the `ASEND(1..16)` block can store data packages in the sending buffer.

The planned jobs are stored in a sending buffer. From here, they are transferred to the ARCNET controller for sending off.

A maximum of 63 data packages can be stored in the sending buffer.

The output STA indicates that the planned job was stored in the sending buffer.
**ASEND(1..16)**

**SEND ARCNET DATA PACKAGES**

**EN**  
**BOOL**  
Dependent on the input EN, the planned job is stored as a data package in the sending buffer and sent off after this.

The following applies:

- **EN = FALSE:**  
The planned job is not stored in the sending buffer and thus not sent off.
- **EN = FALSE/TRUE edge:**  
The planned job is stored as a data package in the sending buffer and sent off then.
- **EN = TRUE:**  
The planned job is stored in the sending buffer, only if the user data of the job have changed.

**D0**  
**INT**  
The DIN identification is specified for the planned job at input D0.

The following applies:  
D0 = 127 (7FH)  
not allowed: 111 (6FH)/95 (5FH)/79 (4FH)/ 63 (3FH)

**J0**  
**INT**  
The job number is specified for the planned job at input J0.

The following applies:  
-32767 < J0 < +32767

**L0**  
**INT**  
The number of user data words is specified for the planned job at input L0.

The following applies:  
0 < L0 < 125

**n**  
**INT**  
The number of node numbers (receiver) is specified at this input.

The following applies:  
0 ≤ n ≤ max. number of receivers (0..15)

**N0..Nn-1**  
**INT**  
If the same job has to be sent to several receivers, the node numbers (station addresses) of the receivers are defined at the inputs N0 ... Nn-1.

- **N0: Node number = 0:**  
  If the node number is »0« at input N0, the job is sent to all stations (Broadcast).
- **N1...Nn-1: Node number = 0:**  
  If the node number is »0« at the inputs N1 ... Nn-1, no broadcast is sent.

If one of the receivers is not able to receive data, the data package is lost for this receiver.

The following applies:  
0 < N0...Nn-1 < 255

**MW**  
**INT**  
The starting word flag of the word flag area to be sent off is specified at input MW. The data of this flag area are sent off as user data in this job.

**STA**  
**BOOL**  
The output STA (status) indicates for one cycle period that the planned job has been stored in the sending buffer.

The following applies:

- **STA = FALSE:**  
The planned job has not been stored in the sending buffer.
- **STA = TRUE:**  
The planned job has been stored in the sending buffer.

Note:
The inputs and outputs can neither be duplicated nor inverted.

---

**Function call in IL**

```
CAL ASEND11(EN := AS_EN, D0 := AS_D0, J0 := AS_J0, L0 := AS_L0, n := 1, N0 := AS_N0, MW := AS_MW)
LD ASEND11.STA
ST AS_STA
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
ASEND11(EN := AS_EN, D0 := AS_D0, J0 := AS_J0, L0 := AS_L0, n := 1, N0 := AS_N0, MW := AS_MW);
AS_STA:=ASEND11.STA;
```
**INITIALIZATION OF THE ARCNET COUPLER, VARIABLE AINIT_V**

The AINIT_V block initializes the ARCNET processing for the blocks ASEND_V, AREC_V and ASTO_V. The blocks ASEND1, ASEND4, ASEND16, AREC cannot be run with this blocks at the same time.

In order to make possible programming and MMI via the ARCNET, the DIP switch for setting the station address must be set unequal to 0 when power is switched on. The ARCNET coupler then is initialized by the operating system as follows:

- Interrupt after reception/sending/reconfiguration
- Long packages = 512 bytes
- Accept broadcasts (data packages to all stations)
- Node (station address) of the DIP switch

If the DIP switch is set to 0, the coupler will not be initialized by the operating system. In this case, it must be set to a node (station address) unequal to 0 using the AINIT_V block, input SNODE. In this operating mode, no programming and no MMI via the ARCNET are available. This is indicated by the DISDIN output (Disable DIN identification). In addition, the ENRA input is not in effect here, i.e. reception is always disabled in case of an imminent buffer overflow.

The AINIT_V block may be planned only once per ARCNET coupler.

**Block data**

<table>
<thead>
<tr>
<th>Available as of PLC runtime system:</th>
<th>V4.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Included in library:</td>
<td>ARCNET_S90_V41.LIB</td>
</tr>
</tbody>
</table>

**Block type**

Function block with historical values

**Parameters**

- **Instance**: AINIT_V
  - **EN**: BOOL, Initialization of the ARCNET coupler with a FALSE/TRUE edge
  - **CONO**: INT, Slot (card number) of the coupler
  - **TOUT**: INT, Timeout in ms for sending of data packages, standard timeout
  - **SNODE**: INT, Node setting (station address of its own), only in effect if the DIP switch = 0!
  - **LRB**: INT, Length of the receiving buffer in bytes
  - **LSHB**: INT, Length of the sending buffer in bytes for high priority
  - **LSLB**: INT, Length of the sending buffer in bytes for low priority
  - **LSTOB**: INT, Length of the buffer for timeout data packages in bytes
  - **LHEAD**: INT, Number of header data in bytes, which are copied with timeout packages
  - **ENBC**: BOOL, Enable/disable broadcast reception (data packages to all stations)
  - **ENLP**: BOOL, Enable/disable long data packages (512 bytes)
### AINIT_V

**INITIALIZATION OF THE ARCNET COUPLER, VARIABLE**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| ENRA | BOOL | TRUE = Always enable reception  
FALSE = Disable reception in case of reception buffer overflow |
| DONE | BOOL | Initialization completed |
| ERR | BOOL | Error has occurred |
| ERRN | INT | Error number |
| DISDIN | BOOL | Disable DIN identification, if signal = TRUE, no DIN identifier are evaluated during reception, e.g. 4F (programming) or 5F (MMI). |
| NODE | INT | Node number of its own (station address), DIP switch or input SNODE (if DIP switch = 0) |
| NENO | INT | Next node number (station address) |
| CRECO | INT | Counts the occurred reconfigurations |
| CMYRECO | INT | Counts the reconfigurations which are caused by this node (station address) |
| CREBC | INT | Counts the received broadcasts (data packages to all stations) |
| LEVHP_BY | INT | Filling level sending buffer high priority in bytes |
| LEVHP_DS | INT | Filling level sending buffer high priority in data sets |
| LEVLP_BY | INT | Filling level sending buffer low priority in bytes |
| LEVLP_DS | INT | Filling level sending buffer low priority in data sets |
| LEVTOS | INT | Filling level of the buffer for timeout data packages in data sets |

After an error-free initialization and as long as the EN input has a TRUE signal, all outputs are updated with every block call.

#### Description

The AINIT_V initializes the ARCNET processing for variable buffer lengths. There is a sending buffer for high priority and another one for low priority. Furthermore, there is a receiving buffer as well as a buffer for those data packages which could not be transmitted successfully (timeout data packages). All buffers can be set variably. The number of the header data of the timeout data packages to be copied can be defined at the LHEAD input.

**EN**

A FALSE/TRUE edge at the EN input triggers the initialization of the ARCNET processing.

**Caution:**

With each FALSE/TRUE edge at the EN input, the buffers are created and initialized. For this reason, the EN input should be kept at TRUE.

With EN=TRUE, the outputs are updated with each call.

If the DIP switch was set to 0 during power ON, the ARCNET coupler is set to the node address given at the SNODE input.

**CONO**

With the CONO input the coupler is selected which works with this function block.

The following applies:

<table>
<thead>
<tr>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>coupler number 1</td>
</tr>
<tr>
<td>1, 2</td>
<td>reserved</td>
</tr>
<tr>
<td>&gt;2</td>
<td>invalid coupler number</td>
</tr>
</tbody>
</table>

**TOUT**

At the TOUT input, the timeout value is specified in ms for sending data packages. This specification is not valid for data packages which get their specific timeout value (unequal to 0) with the ASEND_V block. If a data package cannot be sent within this time (or within the specific time), the sending procedure is aborted and lost. Some distinctive bytes of the data package (see LHEAD input) are stored into the timeout buffer. They can be read with ASTO_V function block.

Recommendation: TOUT = 100 ms
The INITIALIZATION of the ARCNET COUPLER, VARIABLE AINIT_V

**SNODE**

This input only becomes effective, if during power ON the DIP switch for setting the node number (station number) was set to 0. In this case, the ARCNET coupler is initialized with the node number given at the SNODE input.

The following applies: 0 < SNODE <= 255

**LRB**

At the LRB input, the intended length of the receiving buffer is specified in bytes.

**LSHB**

At the LSHB input, the intended length of the sending buffer of high priority is specified in bytes.

**LSLB**

At the LSLB input, the intended length of the sending buffer of low priority is specified in bytes.

**LSTOB**

At the LSTOB input, the intended length of the buffer for the timeout data packages is specified in bytes.

**LHEAD**

In case of timeout data packages, the LHEAD input specifies the number of header data which have to be copied into the readable timeout buffer (see also ASTO_V block). In addition to the header data, the timeout buffer contains the target node (station address) of this lost package as well as the information whether the target node is present in the network at all.

**ENBC**

ENBC = 1 enables the reception of broadcasts (data packages to all stations) for this node (station address). The ENBC input is continuously evaluated after the initialization was successful. The input becomes effective after the coupler has received a telegram and is enabled for the next reception.

ENBC = 0 disables the reception of broadcasts.

**ENLP**

ENLP = 1 enables (ENLP = 0 disable) the reception/sending of long data packages. In all cases, the coupler is able to receive long data packages. If ENLP = 1, the received data packages are made available. If ENLP = 0, they are rejected.

The ENLP input is continuously evaluated after the initialization was successful and the EN input has a signal 1. The sending block generates an error message if a long package is to be sent and the input is set to 0. In this case, the package will not be sent.

**ENRA**

With the ENRA input, the reception via the ARCNET is always possible (ENRA = TRUE) or impossible (ENRA = FALSE) even if the receiving buffer is full. Full means that there are less than 512 bytes free. The ENRA input is only evaluated if the DIP switch for setting the station address was unequal to 0 when the power was switched on.

The following applies:

No reception when receiving buffer is full, if
ENRA = FALSE and DIP switch = 0 or
ENRA = FALSE and DIP switch unequal to 0 or
ENRA = TRUE and DIP switch = 0.

With disabling, programming and MMI are no longer possible.

ENRA = TRUE and DIP switch unequal to 0 means:

Reception is enabled for programming and MMI even if the receiving buffer is full. Data may be lost with telegrams (except MMI and programming).

**DONE**

The DONE output indicates that the initialization was completed. The DONE output has always to be considered together with the ERR output.

The following applies:

DONE = TRUE and ERR = FALSE: The initialization was successful. No error occurred.

DONE = TRUE and ERR = TRUE: An error occurred during initialization.

For details see description of the ERN output.

**ERR**

The ERR output indicates that the initialization procedure of the ARCNET processing was not successful. The ERR output has always to be considered together with the DONE output. If an error occurred, the following applies:

DONE = TRUE and ERR = TRUE.
ERRN INT
The ERRN output describes an occurred error in detail. The ERRN output values mean:

ERRN = 0 No error
ERRN = 1 Error during initialization. The coupler did not response or the coupler number is invalid (>2).
ERRN = 3 Error during creation of the sending/receiving buffer or buffer for the timeout data sets.
ERRN = 4 Error while setting the node number of this station. The SNODE input has an invalid node number.
ERRN = 5 Error when receiving data. With ENLP = 0, the received package is a long data package. This error is automatically deleted as soon as a received package has an allowed length again.
ERRN = 6 Receiving buffer is full. Less than 512 bytes are free in this buffer. This error is automatically deleted as soon as the buffer was read by the AREC block and the free buffer memory becomes more than 512 bytes. ERRN = 6 is not generated if the input ENRA = 1 and the DIP switch for the station address was set to a value unequal to 0 during power ON.

NODE INT
The NODE output indicates the coupler's own node number (station address) after an error-free initialization.

NENO INT
After an error-free initialization, the NENO output indicates the station address of the communication partner which gets the token next.

CRECO INT
The CRECO output indicates the number of networking reconfigurations occurred since the power was switched on. Like power OFF/ON, an 0/1 edge at the EN input resets the CRECO output.

CMYRECO INT
The CMYRECO output indicates the number of networking reconfigurations caused by this node (station address) since the power was switched on. The output value can be cleared with a 0/1 edge at the EN input as well as with power OFF/ON.

CREBC INT
The CREBC output indicates the number of broadcasts (data packages to all stations) which were received by this node (station address) since the power was switched on.

The output value can be cleared with a FALSE/TRUE edge at the EN input as well as with power OFF/ON.

LEVHP_BY INT
The LEVHP_BY output indicates the filling level of the sending buffer (high priority) in bytes provided that the initialization was successful.

LEVHP_DS INT
The LEVHP_DS output indicates the filling level of the sending buffer (high priority) in data sets provided that the initialization was successful.

LEVLP_BY INT
The LEVLP_BY output indicates the filling level of the sending buffer (low priority) in bytes provided that the initialization was successful.

LEVLP_DS INT
The LEVLP_DS output indicates the filling level of the sending buffer (low priority) in data sets provided that the initialization was successful.

LEVTOS INT
The LEVTOS output indicates the filling level of the buffer for the timeout data packages in data sets provided that the initialization was successful.
FUNCTION CALL IN IL

CAL AINIT_V1(EN := AINITV_EN,
TOUT := AINIT_Tout, CONO :=
AINITV_CONO, SNODE := AINITV_SNODE,
LRB := AINITV_LRB, LSHB :=
AINITV_LSHB, LSLB := AINITV_LSHB,
LSTOB := AINITV_LSTOB, LHEAD :=
AINITV_LHEAD, ENBC := AINITV_ENBC,
ENLP := AINITV.ENLP)

LD AINIT_V1.DONE
ST AINITV_DONE
LD AINIT_V1.ERR
ST AINITV_ERR
LD AINIT_V1.ERRN
ST AINITV_ERRN
LD AINIT_V1.DISDIN
ST AINITV_DISDIN
LD AINIT_V1.NODE
ST AINITV_NODE
LD AINIT_V1.NENO
ST AINITV_NENO
LD AINIT_V1.CRECO
ST AINITV_CRECO
LD AINIT_V1.CMYRECO
ST AINITV_CMYRECO
LD AINIT_V1.CREBC
ST AINITV_CREBC
LD AINIT_V1.LEVHP_BY
ST AINITV_LEVHP_BY
LD AINIT_V1.LEVHP_DS
ST AINITV_LEVHP_DS
LD AINIT_V1.LEVLP_BY
ST AINITV_LEVLP_BY
LD AINIT_V1.LEVLP_DS
ST AINITV_LEVLP_DS
LD AINIT_V1.LEVTOS
ST AINITV_LEVTOS

FUNCTION CALL IN ST

AINIT_V1(EN := AINITV_EN,
TOUT := AINIT_Tout, CONO :=
AINITV_CONO, SNODE := AINITV_SNODE,
LRB := AINITV_LRB, LSHB :=
AINITV_LSHB, LSLB := AINITV_LSHB,
LSTOB := AINITV_LSTOB, LHEAD :=
AINITV_LHEAD, ENBC := AINITV_ENBC,
ENLP := AINITV.ENLP);

AINITV_DONE := AINIT_V1.DONE;
AINITV_ERR := AINIT_V1.ERR;
AINITV_ERRN := AINIT_V1.ERRN;
AINITV_DISDIN := AINIT_V1.DISDIN;
AINITV_NODE := AINIT_V1.NODE;
AINITV_NENO := AINIT_V1.NENO;
AINITV_CRECO := AINIT_V1.CRECO;
AINITV_CMYRECO := AINIT_V1.CMYRECO;
AINITV_CREBC := AINIT_V1.CREBC;
AINITV_LEVHP_BY := AINIT_V1.LEVHP_BY;
AINITV_LEVHP_DS := AINIT_V1.LEVHP_DS;
AINITV_LEVLP_BY := AINIT_V1.LEVLP_BY;
AINITV_LEVLP_DS := AINIT_V1.LEVLP_DS;
AINITV_LEVTOS := AINIT_V1.LEVTOS;

Note: The function call in IL has to be performed in one line.
RECEIVE ARCNET DATA PACKAGES

This block reads the next data set from the receiving buffer and makes the user data available in the planned memory area (DATA input). The sender's station address (NODE output) as well as the length of the data package in bytes (LEN output) are further information of the data package. Furthermore, the block informs about the filling level of the receiving buffer in bytes (LEVR_BY output) and in data sets (LEVR_DS output).

Block data

Available as of PLC runtime system: V4.05
Remarks: Included in library: ARCNET_S90_V41.LIB

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>AREC_V</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>INT</td>
<td>Slot (card number) of the coupler</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Variable, where the received user data is stored. The variable must be of ARRAY or STRUCT type.</td>
</tr>
<tr>
<td>NEW</td>
<td>BOOL</td>
<td>Data package is available.</td>
</tr>
<tr>
<td>NODE</td>
<td>INT</td>
<td>Node number (station address) of the sender</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>Length of the data package in bytes</td>
</tr>
<tr>
<td>LEVR_BY</td>
<td>INT</td>
<td>Filling level of the receiving buffer in bytes</td>
</tr>
<tr>
<td>LEVR_DS</td>
<td>INT</td>
<td>Filling level of the receiving buffer in data sets</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>An error occurred</td>
</tr>
<tr>
<td>ERRN</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description

The operating system reads the received data packages from the ARCNET coupler interrupt-controlled and stores them into the receiving buffer. The size of the receiving buffer is defined by the AINIT_V block. It is constructed as a ring buffer with WRITE/READ pointer (FIFO). If there are less than 256 bytes free in the receiving buffer, the reception is disabled and an error is displayed at the AINIT_V block. Only if the ENRA input of the AINIT_V block has a signal 1 and the DIP switch for setting the station address is unequal to 0 during the power ON procedure, the oldest entry is overwritten again. In this way, programming and MMI are possible although the receiving buffer is full. The data packages are stored with variable lengths. A data package with 16 bytes of user data needs exactly 19 bytes in the receiving buffer (user data + sender's node + package length).

The AREC_V block transfers exactly one data package, which is defined by the READ pointer, from the receiving buffer to the planned user memory (DATA input).

The data reception is indicated by the NEW output. The READ pointer is incremented then accordingly.

The ARCNET processing must have been initialized with the AINI_V block before data packages can be read from the receiving buffer with the AREC_V block.
EN \hspace{1cm} BOOL
The block is only processed, if the EN input has a signal 1. If EN = 0, it is not checked whether the receiving buffer contains data packages or not. Only the block outputs LEVR_BY and LEVR_DS are updated.

CONO \hspace{1cm} INT
With the CONO input the coupler is selected, which belongs to this function block.
The following applies:
0 \hspace{0.5cm} = Coupler number 1
1, 2 \hspace{0.5cm} = reserved
>2 \hspace{0.5cm} = invalid coupler number

DATA \hspace{1cm} DWORD
At the DATA input the variable is given, where the user data of a data package has to be copied to. DATA must contain the address of a variable of the ARRAY or STRUCT type.

Byte 0 of the user data is the DIN identifier, the bytes 1 and 2 are the job number of the ARCNET telegram.

CAUTION: Avoid memory area overlappings by adjusting the size of the variables to the maximum of expected data.

NEW \hspace{1cm} BOOL
The NEW output indicates that the user data of a data package has been continuously copied from the receiving buffer to the area of the DATA variable.
The following applies:
NEW = 1: Package received
NEW = 0: No package received

NODE \hspace{1cm} INT
The NODE output indicates the node number (station address) of the sender of the received data package.
The following applies: 0 ≤ NODE ≤ 255

LEN \hspace{1cm} INT
The LEN output indicates the length of the data package in bytes.

LEVR_BY \hspace{1cm} INT
The LEVR_BY output indicates the filling level of the receiving buffer in bytes. A data package uses LEN + 3 bytes in the receiving buffer (user data + 1 byte sender's node + 2 bytes length value).

LEVR_DS \hspace{1cm} INT
The LEVR_DS output indicates the filling level of the receiving buffer in data sets.

ERR \hspace{1cm} BOOL
The ERR output indicates that an error has occurred. This output must always be considered together with the ERRN output.

ERRN \hspace{1cm} INT
The ERRN output specifies an occurred error more exactly:
ERRN = 0 \hspace{0.5cm} No error has occurred
ERRN = 1 \hspace{0.5cm} Invalid coupler number
\hspace{2cm} The coupler number is specified with > 2.
ERRN = 2 \hspace{0.5cm} Initialization error
\hspace{2cm} The AINIT_V block has not initialized the ARCNET processing yet.
Function call in IL

CAL AREC_V1(EN := ARECV_EN, CONO := ARECV_CONO, DATA := ARECV_DATA)
LD ARECV.NEW
ST ARECV.NEW
LD ARECV.NODE
ST ARECV.NODE
LD ARECV.LEN
ST ARECV.LEN
LD ARECV.LEVR_BY
ST ARECV.LEVR_BY
LD ARECV.LEVR_DS
ST ARECV.LEVR_DS
LD ARECV.ERR
ST ARECV.ERR
LD ARECV.ERRN
ST ARECV.ERRN

Note: The function call in IL has to be performed in one line.

Function call in ST

AREC_V1(EN := ARECV_EN, CONO := ARECV_CONO, DATA := ARECV_DATA);
ARECV.NEW := ARECV.V1.NEW;
ARECV.NODE := ARECV.V1.NODE;
ARECV.LEN := ARECV.V1.LEN;
ARECV.LEVR_BY := ARECV.V1.LEVR_BY;
ARECV.LEVR_DS := ARECV.V1.LEVR_DS;
ARECV.ERR := ARECV.V1.ERR;
ARECV.ERRN := ARECV.V1.ERRN;
SEND ARCNET DATA PACKAGES TO ONE SUBSCRIBER

The ASEND_V block serves for sending data packages via the ARCNET.

The planned packages are stored in a sending buffer. From here, they are handed over by the operating system to the ARCNET coupler in order to be sent.

The ASEND_V block can send exactly one data package at a time to one station or as a broadcast (to all stations).

Block data
Available as of PLC runtime system: V4.05
Included in library: ARCNET_S90_V41.LIB

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>ASEND_V</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enable: send data package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE: do not send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TRUE: do send</td>
</tr>
<tr>
<td>CONO</td>
<td>INT</td>
<td>Slot (card number) of the coupler</td>
</tr>
<tr>
<td>NODE</td>
<td>INT</td>
<td>Node number (station address) of the receiver</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>Number of the user data bytes to be sent</td>
</tr>
<tr>
<td>PRIO</td>
<td>BOOL</td>
<td>Sending priority of the data package (FALSE = low priority, TRUE = high priority)</td>
</tr>
<tr>
<td>TOUT</td>
<td>INT</td>
<td>Timeout of this data package in ms</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Variable containing the start address for copying data to the sending buffer. The variable must be of ARRAY or STRUCT type.</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Data package has been stored in the sending buffer.</td>
</tr>
<tr>
<td>LEV_BY</td>
<td>INT</td>
<td>Filling level of the sending buffer of low/high priority (depending on the PRIO input) in bytes</td>
</tr>
<tr>
<td>LEV_DS</td>
<td>INT</td>
<td>Filling level of the sending buffer of low/high priority (depending on the PRIO input) in data sets</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>An error has occurred</td>
</tr>
<tr>
<td>ERRN</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description

The ASEND_V block serves for sending data packages via the ARCNET.

The planned packages are stored into the sending buffer, which is selected with the PRIO input. From here, they are handed over by the operating system to the ARCNET coupler in order to be sent.

The size of the sending buffer is defined with the AINIT_V block.

The DONE output indicates that the planned package has been stored in the sending buffer.

The outputs ERR and ERRN provide error messages. In case of an error, the data package must be sent again.

The ARCNET processing must have been initialized successfully by the AINIT_V block before the ASEND_V block can store data packages in the sending buffer.
The planned package is stored in the sending buffer and then sent depending on the status of the EN input.

The following applies:

- **EN = FALSE**: The planned package is not stored in the sending buffer and not sent.
- **EN = TRUE**: The planned package is stored in the sending buffer and sent.

**CONO**

With the CONO input the coupler is selected which works with this function block.

The following applies:

- 0 = coupler number 1
- 1, 2 = reserved
- >2 = invalid coupler number

**NODE**

The node number (station address) of the receiver is specified at the NODE input.

If the node number »0« is specified, the data package is sent to all stations (broadcast).

If in case of broadcasting one of the receivers is not ready for reception, this receiver will never receive the data package.

The following applies: 0 ≤ NODE ≤ 255

**LEN**

The number of user data bytes of the planned package is specified at the LEN input.

The following applies:

- 1 ≤ LEN ≤ 253 for short data packages
- 1 ≤ LEN ≤ 508 for long data packages

For both data packages, the values 0, 254, 255 and 256 bytes are always forbidden.

**PRIO**

The PRIO input specifies the sending priority of the data package.

The following applies:

- **PRIO = FALSE**: The planned data package has a low priority. It is entered into the sending buffer for low priority. All outputs refer to this buffer.
- **PRIO = TRUE**: The planned data package has a high priority. It is entered into the sending buffer for high priority. All outputs refer to this buffer.

**TOUT**

At the TOUT input, the timeout value is specified in ms for sending data packages. If a data package cannot be sent within this time, the sending procedure is aborted and lost. Some distinctive bytes of the data package (see LHEAD input at the AINIT_V block) are stored into the timeout buffer. They can be read with ASTO_V function block.

The following applies:

- **TOUT = 0**: The planned package does not have any specific timeout value. The sending monitoring uses the timeout specification which is given at the TOUT input of the AINIT_V block.
- **TOUT > 0**: The planned package is sent with the specified timeout value.

Recommendation: TOUT = 100 ms

**DATA**

At the DATA input, the variable is specified, whose data is sent in this package as user data. DATA must contain the address of a variable of ARRAY or STRUCT type.

Byte 0 of the user data is the DIN identifier, the bytes 1 and 2 are the job number of the ARCNET telegram.

**DONE**

The DONE output indicates whether or not the planned package has been stored in the sending buffer.

The following applies:

- **DONE = 0**: The planned package has not been stored in the sending buffer.
- **DONE = 1**: The planned package has been stored in the sending buffer.

**LEV_BY**

The LEV_BY output indicates the filling level in bytes of the sending buffer which was selected with the PRIO input. One data package uses LEN + 5 bytes in the sending buffer (user data + 1 byte for receiver's node + 2 bytes for the length value + 2 bytes for the specific timeout value).
The LEV_DS output indicates the filling level in data sets of the sending buffer which was selected with the PRIO input.

The ERR output indicates that an error has occurred. This output has always to be considered together with the ERRN output.

The ERRN output describes an occurred error in detail:

- **ERRN = 0** No error
- **ERRN = 1** Invalid coupler number; The coupler number is specified greater than 2.
- **ERRN = 2** Initialization error; The AINIT_V block did not initialize the ARCNET processing yet.
- **ERRN = 3** Error in data length; The data length specified at the LEN input is inadmissible. Either the value is within the prohibited range (see description of the LEN input) or it was attempted to send a long package although the ENLP input of the AINIT_V block is set to FALSE.
- **ERRN = 4** Error of a full sending buffer; The sending buffer selected with the PRIO input is full. The user has to take care for sending the package again.

### Function call in IL

```il
CAL STAT
(EN := STAT_EN, CONO := STAT_CONO)
LD STAT.ERR
ST STAT.ERR
LD STAT.STATE_BITS
ST STAT.STATE_BITS
LD STAT.DPM_STATE
ST STAT.DPM_STATE
LD STAT.COM_ERR
ST STAT.COM_ERR
LD STAT.BUS_ERR
ST STAT.BUS_ERR
LD STAT.TIME_OUT
ST STAT.TIME_OUT
```

Note: The function call in IL has to be performed in one line.

### Function call in ST

```st
STAT
(EN := STAT_EN, CONO := STAT_CONO);
STAT.ERR := STAT.ERR;
STAT.STATE_BITS := STAT.STATE_BITS;
STAT.DPM_STATE := STAT.DPM_STATE;
STAT.COM_ERR := STAT.COM_ERR;
STAT.BUS_ERR := STAT.BUS_ERR;
STAT.TIME_OUT := STAT.TIME_OUT;
```
READ ARCNET TIMEOUT DATA PACKAGES  

ASTO_V

If the sending monitoring time expires while sending a data package, the data package has not reached the receiver. The following information is stored in the timeout data buffer then: the receiver address, the first header data of the data package (see LHEAD input of the AINIT_V block for the number of them) and information whether the receiver was found in the networking or not. This block reads such lost data packages from the timeout data buffer and provides their information to the user.

Block data
Available as of PLC runtime system: V4.05
Included in library: ARCNET_S90_V41.LIB

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>ASTO_V</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>enable for the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>INT</td>
<td>Slot (card number) of the coupler</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address of a variable as of where the data of the timeout package are placed. The variable must be of ARRAY or STRUCT type.</td>
</tr>
<tr>
<td>NEW</td>
<td>BOOL</td>
<td>Data package is provided</td>
</tr>
<tr>
<td>LEVDS</td>
<td>INT</td>
<td>Filling level of the timeout data package in data sets</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>An error occurred</td>
</tr>
<tr>
<td>ERRN</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description
The sending procedure of a data package is monitored by a settable timeout value. If the timeout expires, distinctive information about the involved data package is placed into the timeout data buffer. There are in succession:

- Node ID (station address) of the receiver (1 byte)
- Receiver is in the networking or not (1 byte),
  0 = receiver is not in the networking or it was attempted to send a long package to a node which was initialized only for receiving short packages.
  1 = Receiver is in the networking
- Header data of the data set (the number of them is planned at the LHEAD input of the AINIT_V block).

The buffer length is settable at the LSTOB input of the AINIT_V block. It is constructed as a ring buffer (FIFO). If it is full, the oldest entry is overwritten again. The ASTO_V block checks with a signal 1 at the EN input, whether a data package was placed in the buffer and provides the above mentioned information to the user as of the variable given at the DATA input.

The ASTO_V block only can operate if the ARCNET processing was successfully initialized by the AINIT_V block.

EN BOOL
The block is processed as long as the EN input has a TRUE signal.
The CONO input selects the coupler which works together with this block.

The following applies:
0 = coupler number 1
1, 2 = reserved
>2 = invalid coupler number

At the DATA input, the variable is given where the information of the current data package is copied to continuously.
DATA must contain the address of a variable of ARRAY or STRUCT type.

CAUTION: Avoid memory area overlappings by adjusting the size of the variables to the maximum of expected data.

The NEW output indicates that the information of a data package has been continuously copied from the timeout data buffer into the area beginning at the variable given at the DATA input.

The following applies:
NEW = TRUE: Information copied
NEW = FALSE: No information available

Function call in IL

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL ASTO_V1(EN := ASTOV_EN, CONO := ASTOV_CONO, DATA := ASTOV_DATA)</td>
<td>Call block ASTO_V1</td>
</tr>
<tr>
<td>LD ASTO_V1.NEW</td>
<td>Load variable NEW</td>
</tr>
<tr>
<td>ST ASTOV_NEW</td>
<td>Store variable NEW</td>
</tr>
<tr>
<td>LD ASTO_V1.LEVDS</td>
<td>Load variable LEVDS</td>
</tr>
<tr>
<td>ST ASTOV_LEVDS</td>
<td>Store variable LEVDS</td>
</tr>
<tr>
<td>LD ASTO_V1.ERR</td>
<td>Load variable ERR</td>
</tr>
<tr>
<td>ST ASTOV_ERR</td>
<td>Store variable ERR</td>
</tr>
<tr>
<td>LD ASTO_V1.ERRN</td>
<td>Load variable ERRN</td>
</tr>
<tr>
<td>ST ASTOV_ERRN</td>
<td>Store variable ERRN</td>
</tr>
</tbody>
</table>

Note: The function call in IL has to be performed in one line.

Function call in ST

ASTO_V1(EN := ASTOV_EN, CONO := ASTOV_CONO, DATA := ASTOV_DATA);

ASTOV_NEW := ASTO_V1.NEW;
ASTOV_LEVDS := ASTO_V1.LEVDS;
ASTOV_ERR := ASTO_V1.ERR;
ASTOV_ERRN := ASTO_V1.ERRN;
The _5F_ARC97 block realizes a driver for the 5F protocol via ARCNET. The block works independently of other ARCNET blocks used at the same time. However, only one _5F_ARC97 block may exist. It is not necessary to use an AINIT block.

The _5F_ARC97 block can work for 1...255 masters. The exact number of masters with the 5F protocol is defined by the user by creating the corresponding communication buffers in the PLC program. The block manages the data in these communication buffers as ring memories.

All outputs are updated with every block call provided that the initialization was successful.

**Block data**

Available as of PLC runtime system: V4.05
- as of V4.15
- as of V4.16

Remarks:
- Telegrams with CP<>3 are recognized
- Masters with node >= 128 are recognized

Included in library: ARCNET_S90_V41.LIB

**Block type**

Function block with 22 bytes of historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>_5F_ARC97</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>INT</td>
<td>Number of masters with 5F protocol</td>
</tr>
<tr>
<td>_5F_MASTER</td>
<td>POINTER TO INT</td>
<td>Pointer to ARRAY OF INT</td>
</tr>
<tr>
<td>_5F_BUF</td>
<td>POINTER TO _5F_DATA_TYP</td>
<td>Pointer to the data memory for 5F telegrams</td>
</tr>
<tr>
<td>MASTER</td>
<td>WORD</td>
<td>Master address processed last</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Telegram received</td>
</tr>
<tr>
<td>ERR</td>
<td>INT</td>
<td>Error number</td>
</tr>
<tr>
<td>VOLL</td>
<td>BOOL</td>
<td>Indication &quot;data memory is full&quot;</td>
</tr>
</tbody>
</table>

**Description**

In order to get an easy-to-use and effective data transmission between the PLC and visualization systems, a protocol according to MODBUS was realized. The protocol employs the $5F_{HEX} = 95_{DEC}$ DIN identification. ARCNET communication with other DIN identifications, e.g. using the AREC and ASEND blocks as well as programming via ARCNET is also possible. The protocol uses the master/slave principle. The visualization system always works as the master, and the PLC always as the slave. In the visualization system it is configured, which data are to be sent or received. In the PLC always a standard program is used.

The _5F_ARC97 block realizes a driver for the 5F protocol via ARCNET. The block works independently of other ARCNET blocks used at the same time. However, only one _5F_ARC97 block may exist. It is not necessary to use an AINIT block. The _5F_ARC97 block can work for 1...255 masters. The exact number of masters with the 5F protocol is defined by the user by creating the corresponding communication buffers in the PLC program. The block manages the data in these communication buffers as ring memories.
The telegrams consist of an ARCNET header (CP, DIN, JOB_NR), a counter and the MODBUS telegram which may be max. 124 words or 248 bytes of size. All telegrams are sent with the 5F DIN identification. The ARCNET address of the corresponding participant is always the JOB number.

The ARCNET telegram is composed in the following way:

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>CP</th>
<th>DIN</th>
<th>MASTER Low</th>
<th>MASTER High</th>
<th>Counter Low</th>
<th>Counter High</th>
<th>SLAVE</th>
<th>FCT</th>
<th>User data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
<td>03</td>
<td>5F</td>
<td>xx</td>
<td>00</td>
<td>aa</td>
<td>aa</td>
<td>yy</td>
<td>zz</td>
<td>10-255</td>
</tr>
<tr>
<td>Byte</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- **xx** = ARCNET address of the master == JOB No at the AREC/ASEND block
- **yy** = ARCNET address of the slave, as of here the MODBUS telegram begins
- **aaaa** = Counter, is updated by the master and reflected by the slave (in order to avoid telegram duplication). This counter is not used for the checksum.
- **zz** = MODBUS function code

As of the FCT function code, the structure of the telegram follows the MODBUS standard. The telegrams are entered in the communication buffer (see _5F_BUF). The first half of the buffer is used for reception, the second half for sending. Here, one telegram needs 256 bytes.

**N**

The number of planned masters is specified at the N input. This number must be equal to the number of elements in the ARRAYS, which are pointing to _5F_MASTER and _5F_BUF. The 5F_BUF array is N+1 elements in size.

**_5F_MASTER**

In order to manage more than 1 master with the 5F protocol, an ARRAY OF INT must be set up containing the ARCNET addresses of the participants. If only one master exists, a single INT value is sufficient. A pointer to this value or to the first value of the array is set to the _5F_MASTER input (see ADR operator).

**_5F_BUF**

In the PLC program, a communication buffer has to be built an ARRAY of ARRAY OF _5F_DATA_TYP type. The ARRAY must have N+1 (N = number of the masters) elements. A pointer to the first element of the array is specified at the _5F_BUF input.

**MASTER**

The ARCNET address of the telegram last processed by the block is present at the MASTER output. If several masters are connected, the block runs through a loop in order to process all received telegrams.

**RDY**

The RDY output is set when the block has processed a telegram.

**ERR**

The ERR output indicates an error number. The error number is the error number of the MODBUS protocol.

**VOLL**

This output is set, if more telegrams were received than the communication buffer could load. This is the case if either masters sent more than one telegram per cycle or, participants, that are not entered in the 5F_MASTER ARRAY, have sent telegrams with 5F DIN identification. The communication buffer as well as the ARRAY can be expanded then and the N input may be given a greater value accordingly.
Function call in IL

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>a_5F_Master[1]</td>
</tr>
<tr>
<td>ADR</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>ARC97_1._5F_MASTER</td>
</tr>
<tr>
<td>LD</td>
<td>aARC_BUF[0]</td>
</tr>
<tr>
<td>ADR</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>ARC97_1._5F_BUF</td>
</tr>
<tr>
<td>CAL</td>
<td>ARC97_1(n := iAnzMaster)</td>
</tr>
<tr>
<td>LD</td>
<td>ARC97_1.MASTER</td>
</tr>
<tr>
<td>ST</td>
<td>ARC_Master</td>
</tr>
<tr>
<td>LD</td>
<td>ARC97_1.RDY</td>
</tr>
<tr>
<td>ST</td>
<td>ARC_RDY</td>
</tr>
<tr>
<td>LD</td>
<td>ARC97_1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>ARC_ERR</td>
</tr>
<tr>
<td>LD</td>
<td>ARC97_1.VOLL</td>
</tr>
<tr>
<td>ST</td>
<td>ARC_VOLL</td>
</tr>
</tbody>
</table>

Note: The function call in IL has to be performed in one line.

Function call in ST

```
ARC97_1(n := iAnzMaster,
_5F_MASTER := ADR(a_5F_Master[1]),
_5F_BUF := ADR(aARC_BUF[0]));
ARC_Master := ARC97_1.MASTER;
ARC_RDY := ARC97_1.RDY;
ARC_ERR := ARC97_1.ERR;
ARC_VOLL := ARC97_1.VOLL;
```
Programming example (in ST):

PROGRAM ARC_5F
VAR CONSTANT
  iAnzMaster : INT := 2; (* Number of masters *)
END_VAR
VAR
  a_5F_Master : ARRAY[1..iAnzMaster] OF INT := 1, 10; (* Node numbers of the masters *)
  ARC97_1 : _5F_ARC97; (* Instance *)
  aARC_BUF : ARRAY[0..iAnzMaster] OF _5F_DATA_TYP; (* Telegram buffer size: iAnzMaster+1 ! *)
  ARC_Master : INT; (* optional: Node of the master processed last *)
  ARC_RDY : BOOL; (* optional: Telegram processed *)
  ARC_ERR : INT; (* optional: Error number see MODMAST *)
  ARC_VOLL : BOOL; (* optional: Buffer overflow *)
  iTeleCounter : INT; (* optional: Telegram counter local *)
  MW00_00 AT %MW1000.0 : INT; (* optional: Telegram counter for transmission *)
END_VAR

(* Block call of _5F_ARC97 *)
ARC97_1(n := iAnzMaster,
       _5F_MASTER := ADR(a_5F_Master[1]),
       _5F_BUF := ADR(aARC_BUF[0]));
ARC_Master := ARC97_1.MASTER; (* just for display, can be deleted *)
ARC_RDY := ARC97_1.RDY; (* just for display, can be deleted *)
ARC_ERR := ARC97_1.ERR; (* just for display, can be deleted *)
ARC_VOLL := ARC97_1.VOLL; (* just for display, can be deleted *)

(* Counter for the telegrams / just for indication that the communication is running *)
iTeleCounter := SEL(ARC_RDY, iTeleCounter, iTeleCounter+1);
MW00_00 := SEL(F_TRIG1.Q, MW00_00, MW00_00+1);
(* If MW00_00 is used in the project, comment out here *)
Glossary

BOOL
Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT
DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

DWORD
DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

INT
INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

WORD
WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

Function blocks

Function blocks are subroutines which can have as much inputs, outputs and internal variables as required. They are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. This allows for example to realize counters which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters. A function has maximally one output parameter. Note that the output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

   For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

   For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

   The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

_5F_ARC97 S90 24

A
AINIT S90 4
AINIT_V S90 11
ARCNET S90
_5F_ARC97 24
AINIT 4
AINIT_V 11
AREC 7, 16
ASEND 9
ASEND_V 19
ASTO_V 22
AREC S90 7, 16
ASEND S90 9
ASEND_V S90 19
ASTO_V S90 22

C
Components of the ARCNET library 2
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

Function Blocks
for the CS31-Bus
90 Series

CS31 Bus
90 Series
Contents

The CS31 Library ............................................................. 2
PRECONDITIONS FOR THE USE OF THE CS31 LIBRARY ............... 2
COMPONENTS OF THE CS31 LIBRARY ................................. 2

Overview of blocks arranged according to their call names ........... 2
CONFIGURE AC31 MODULES CS31CO ................................. 3
ACKNOWLEDGE AC31 ERRORS CS31QU ................................ 10
TESTING OF AC31 MODULES CS31TE ................................. 11

Glossary .......................................................................... 13

Index ................................................................................ 15
The CS31 Library

PRECONDITIONS FOR THE USE OF THE CS31 LIBRARY

Note:
The blocks of the CS31 library only run in the RUN mode of the PLC, but not in the simulation mode.

COMPONENTS OF THE CS31 LIBRARY

The following function blocks are contained in the CS31 library:

CS31CO - Configure AC31 modules
CS31QU - Acknowledge AC31 errors
CS31TE - Testing of AC31 modules

Detailed descriptions of the blocks are in the following sections.

Overview of blocks arranged according to their call names

Character descriptions:
FBmV … Function block with historical values
FBoV … Function block without historical values
F … Function

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS31CO</td>
<td>FBmV</td>
<td>Configure AC31 modules</td>
<td>3</td>
</tr>
<tr>
<td>CS31QU</td>
<td>FBoV</td>
<td>Acknowledge AC31 errors</td>
<td>10</td>
</tr>
<tr>
<td>CS31TE</td>
<td>FBoV</td>
<td>Testing of AC31 modules</td>
<td>11</td>
</tr>
</tbody>
</table>
The function block is used to configure the AC31 remote modules. The block can both send configuration parameters to the remote modules and also scan their currently set configuration.

Apart from configuration of the AC31 remote modules, the function block can also process further jobs (see list of jobs).

Enable for processing a job once is triggered by a FALSE/TRUE edge at input FREI.
**FREI**

Processing of the block is controlled via input FREI.

**FREI = FALSE:**
All block outputs are set to the value »FALSE«. However, this is not valid, if a job is currently being processed, i.e. processing of a job which is currently being processed, is not affected by FREI = FALSE.

**FREI = FALSE/TRUE edge:**
Processing of the job is enabled. Input FREI is no longer evaluated during processing of the job.

**FREI = TRUE:**
The block is not processed, i.e. it no longer changes its outputs. However, this is not valid, if a job is currently being processed.

**GRN**

Group number with which the remote module is addressed by the PLC program.

Range: 0 … 63

Example:
On binary input E 12,08, »12« is the group number and »08« is the channel number.

**CODE**

The identification of the job to be executed is specified at input CODE (see list of jobs on the next page).

**D1…D8**

The parameters required for the job are preset at the inputs D1 ... D8. The number of parameters depends on the job to be executed. There are also jobs requiring no parameters (see list of jobs on the next page).

**RDY**

The output RDY indicates that processing of the job currently being processed is completed. This output does not indicate whether processing of the job was successful or not. The output RDY has therefore always to be considered together with the output OK.

**RDY = TRUE and OK = TRUE:**
Processing of the job is completed without errors. A new job can be started with a FALSE/TRUE edge at input FREI.

**RDY = TRUE and OK = FALSE:**
During processing of the job an error has been detected. The corresponding error identification is present at output ERR. A new job can be started with a FALSE/TRUE edge at input FREI.

**RDY = FALSE:**
Processing of an enabled job has not yet been completed (job is still running) or output RDY has been reset with FREI = FALSE.

**OK**

Output OK indicates whether the job has been handled successfully or whether an error has been detected during processing. In case of an error, an error number is indicated at output ERR. The output OK is not valid until the job has been completed, i.e. if RDY = TRUE.

The following applies:

If RDY = TRUE and OK = TRUE: The job has been processed successfully.

OK = FALSE: During processing of the job an error has been detected.

**ERR**

At the output ERR status and error identifications are output. The status identifications are output during processing of a job in order to signalize in what stage of processing the job currently is. After enabling a job, status identifications are signalized only for as long as RDY = FALSE.

The error identifications are output after completion of the job processing if an error has occurred. Error identifications are thus not signalized until

RDY = TRUE and OK = FALSE
Error identifications

ERR = 1: An illegal job identification has been specified at input CODE.

ERR = 2: Incorrect parameters have been specified at the inputs D1 ... D8 (e.g. a group number for which there is no remote module on the CS31 system bus).

ERR = 3: The addressed AC31 remote module does not accept the job.

Status identifications

ERR = 8: The function block is waiting since a job of another user is currently being processed.

ERR = 10: The job has been sent to the receiver and the block is waiting for its response.

A1...A7INT

After completion of job processing, the response is available at the outputs A1 ... A7. The number of response parameters depends on the job performed (see list of jobs).

List of jobs

Processing a job consists of:
– transferring the job and
– supplying the OK response or not-OK response

The OK response is described in connection with the corresponding job.
The not-OK response of the individual jobs always looks as follows:

* Not-OK response
The following basically applies for the not-OK response:

RDY: TRUE
OK: FALSE
ERR: 1. inadmissible job identification
2. wrong parameter; e.g. group number to which there exists no remote module
3. remote module does not accept the job
A1 ... A7: 0

Updating of the maximum number of remote modules detected

The input word EW 07,15 contains, amongst other things, the maximum number of remote modules detected in the past. The actual number of remote modules which exist at the moment may be less. This command is used to update this value. The modules which exist are counted and the value is stored. The user can inquire this value in the PLC program (EW 07,15, bit 8 ... 15).

Job

GRN: 255 (Master PLC with bus)
CODE: 132
D1 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1 ... A7: 0

Inquiring the open-circuit monitoring of an input to determine whether it is activated or deactivated

Job

GRN: group number 0 ... 63
CODE: 32
D1: channel number
D2 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1: 47. open-circuit monitoring ON
32. open-circuit monitoring OFF
A2 ... A7: 0

Inquiring the open-circuit monitoring of an output to determine whether it is activated or deactivated

Job

GRN: group number 0 ... 63
CODE: 33
D1: channel number
D2 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1: 47. open-circuit monitoring ON
32. open-circuit monitoring OFF
A2 ... A7: 0

Deactivating or activating the open-circuit monitoring of an input

Job

GRN: group number 0 ... 63
CODE: 224. open-circuit monitoring ON
160. open-circuit monitoring OFF
D1: channel number
D2 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1 ... A7: 0
Deactivating or activating the open-circuit monitoring of an output

Job
GRN: group number 0 ... 63
CODE: 225. open-circuit monitoring ON
       161. open-circuit monitoring OFF
D1: channel number
D2 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1 ... A7: 0

Setting the input delay of a channel

Job
GRN: group number 0...63
CODE: 166
D1: channel number
D2: input delay
   2. 2 ms
   4. 4 ms
   ...
   ... ...
   30. 30 ms
   32. 32 ms
A2 ... A7: 0

Inquiring a channel to determine whether it is configured as input or input/output

Job
GRN: group number 0 ... 63
CODE: 34
D1: channel number
D2 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1: 34. input
    35. input/output
A2 ... A7: 0

Configuration of a channel as input or input/output

Job
GRN: group number 0 ... 63
CODE: 162. input
       163. input/output
D1: channel number
D2 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1 ... A7: 0

Inquiring the input delay of a channel

Job
GRN: group number 0 ... 63
CODE: 38
D1: channel number
D2 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1: input delay:
    2. 2 ms
    4. 4 ms
    ...
A2 ... A7: 0

Acknowledging errors on remote module

This command can be used to reset the error messages registered on the selected remote module. A reset is possible only if the cause of the error is no longer operative.

Job
GRN: group number 0 ... 63
CODE: 232
D1: lowest channel number on the module:
    0. lowest channel number on the module is 0 (<7)
    8. lowest channel number on the module is 8 (>7)
D2: module type:
    0. binary input
    1. analog input
    2. binary output
    3. analog output
    4. binary input/output
    5. analog input/output
    30. 30 ms
    32. 32 ms
D3 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1 ... A7: 0
Acknowledging errors on remote module and resetting configuration values to default setting

In addition to the job »Acknowledging errors on remote module«, all configurable settings are reset to the default setting.

Job

| GRN: | group number 0 ... 63 |
| CODE: | 233 |
| D1: | first channel number on the module: 0. first channel number on the module is 0 (<7) 8. first channel number on the module is 8 (>7) |
| D2: | module type: 0. binary input 1. analog input 2. binary output 3. analog output 4. binary input/output 5. analog input/output |
| Note: | Bit: even number (0, 2, 4) Word: odd number (1, 3, 5) |
| D3 ... D8: | not used |

OK response

| RDY: | TRUE |
| OK: | TRUE |
| A1 ... A7: | 0 |

Inquiring the configuration of an analog input

Job

| GRN: | group number 0 ... 63 |
| CODE: | 42 |
| D1: | channel number |
| D2 ... D8: | not used |

OK response

| RDY: | TRUE |
| OK: | TRUE |
| A1: | 50. input 0 ... 20 mA 49. input 4 ... 20 mA |
| A2 ... A7: | 0 |

Configuration of an analog input

Job

| GRN: | group number 0 ... 63 |
| CODE: | 170 |
| D1: | channel number |
| D2: | 50. input 0 ... 20 mA 49. input 4 ... 20 mA |
| D3 ... D8: | not used |

OK response

| RDY: | TRUE |
| OK: | TRUE |
| A1 ... A7: | 0 |

Configuration of an analog output

Job

| GRN: | group number 0 ... 63 |
| CODE: | 171 |
| D1: | channel number |
| D2: | 50. output 0 ... 20 mA 49. output 4 ... 20 mA 51. output +10 V |
| D3 ... D8: | not used |

OK response

| RDY: | TRUE |
| OK: | TRUE |
| A1 ... A7: | 0 |

Inquiring the bus configuration

The bus interface of the Master PLC has a list which stores specific data of the remote modules. The remote modules are numbered in this list in the order in which they can be found on the CS31 system bus. The internal number of the modules must be specified with this command. The response to this command is the group number stored under this number and status information on the corresponding module.

Job

| GRN: | not evaluated |
| CODE: | 80 |
| D1: | number from the module list (1 ... 31) |
| D2 ... D8: | not used |
OK response
RDY: TRUE
OK: TRUE
A1: status of the remote module:
  Bit 0 ... 3: number of process data bytes (binary module) or words (word module), which the module sends to the master.
  Bit 4 ... 7: number of process data bytes (binary module) or words (word module), which the master sends to the module
A2: group number (0 ... 63)
A3: Bit 0: 0. lowest channel number <7
  1. lowest channel number >7
  Bit 1: 0. binary module
  1. word module
A4 ... A7: 0

Read 1 ... 6 bytes

Job
GRN: group number 0 ... 63
CODE: 49. read 1 byte
  50. read 2 byte
  51. read 3 byte
  52. read 4 byte
  53. read 5 byte
  54. read 6 byte
D1: first channel number on the module:
  0. first channel number on the module is 0 (<7)
  8. first channel number on the module is 8 (>7)
D2: module type:
  0. binary input
  1. analog input
  2. binary output
  3. analog output
  4. binary input/output
  5. analog input/output
Note:
Bit: even number (0, 2, 4)
Word: odd number (1, 3, 5)
D3: byte start address (Low Byte)
D4: byte start address (High Byte)
D5 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1: value of 1st byte
A2 ... A7: 0

Write 1 ... 4 bytes

Job
GRN: group number 0 ... 63
CODE: 65. write 1 byte
  66. write 2 byte
  67. write 3 byte
  68. write 4 byte
D1: first channel number on the module:
  0. first channel number on the module is 0 (<7)
  8. first channel number on the module is 8 (>7)
D2: module type:
  0. binary input
  1. analog input
  2. binary output
  3. analog output
  4. binary input/output
  5. analog input/output
Note:
Bit: even number (0, 2, 4)
Word: odd number (1, 3, 5)
D3: byte start address (Low Byte)
D4: byte start address (High Byte)
D5 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1: value of 1st byte
A2 ... A7: 0
OK response
RDY: TRUE
OK: TRUE
A1 ... A7: 0

Write 1 bit of 1 byte

Job
GRN: group number 0 ... 63
CODE: 79
D1: first channel number on the module:
0. first channel number on the module is 0 (<7)
8. first channel number on the module is 8 (>7)
D2: module type:
0. binary input
1. analog input

2. binary output
3. analog output
4. binary input/output
5. analog input/output

Note:
Bit: even number (0, 2, 4)
Word: odd number (1, 3, 5)
D3: byte start address (Low Byte)
D4: byte start address (High Byte)
D5: bit position within bytes 0 ... 7
D6: bit value (0 or 1)
D7 ... D8: not used

OK response
RDY: TRUE
OK: TRUE
A1 ... A7: 0

Function call in IL
CAL CS31CO1(FREI := CSCO_FREI,
GRN := CSCO_GRN,
CODE := CSCO_CODE, D1 := CSCO_D1,
D2 := CSCO_D2, D3 := CSCO_D3,
D4 := CSCO_D4, D5 := CSCO_D5,
D6 := CSCO_D6, D7 := CSCO_D7,
D8 := CSCO_D8)
LD CS31CO1.OK
ST CSCO_OK
LD CS31CO1.ERR
ST CSCO_ERR
LD CS31CO1.A1
ST CSCO_A1
LD CS31CO1.A2
ST CSCO_A2
LD CS31CO1.A3
ST CSCO_A3
LD CS31CO1.A4
ST CSCO_A4
LD CS31CO1.A5
ST CSCO_A5
LD CS31CO1.A6
ST CSCO_A6
LD CS31CO1.A7
ST CSCO_A7
LD CS31CO1.RDY
ST CSCO_RDY

Note: In IL, the function call has to be performed in one line.

Function call in ST
CS31CO1(FREI := CSCO_FREI,
GRN := CSCO_GRN, CODE := CSCO_CODE,
D1 := CSCO_D1, D2 := CSCO_D2, D3 := CSCO_D3,
D4 := CSCO_D4, D5 := CSCO_D5, D6 := CSCO_D6,
D7 := CSCO_D7, D8 := CSCO_D8);
CSCO_OK:=CS31CO1.OK;
CSCO_ERR:=CS31CO1.ERR;
CSCO_A1:=CS31CO1.A1;
CSCO_A2:=CS31CO1.A2;
CSCO_A3:=CS31CO1.A3;
CSCO_A4:=CS31CO1.A4;
CSCO_A5:=CS31CO1.A5;
CSCO_A6:=CS31CO1.A6;
CSCO_A7:=CS31CO1.A7;
CSCO_RDY:=CS31CO1.RDY;
ACKNOWLEDGE AC31 ERRORS

This block allows to acknowledge automatically error messages of AC31 remote modules.

Block type
- Function block with historical values

Parameters
- Instance: CS31QU
- Instance name
- FREI: BOOL
  - Enabling of the block processing

Description
- This function block allows to acknowledge automatically error messages of AC31 remote modules. Error messages are stored on the AC31 remote modules until they are acknowledged. Even if the error has been removed, the error message is still pending on the module until acknowledgement and is also signaled to the PLC until the message is acknowledged.

  Processing of the block is enabled with a TRUE signal at input FREI, and the block then acknowledges AC31 errors continuously.

  It may take several PLC cycles to acknowledge an error on an AC31 module.

  If the function block is enabled, it constantly checks whether an AC31 error of class 3 or 4 has occurred and acknowledges this error.

- An AC31 error of class 3 has occurred:
  - The block acknowledges the error on the AC31 remote module which signals the error and also clears the error message on the PLC, i.e. the error flag M 255,13 is reset and LED FK3 is deactivated.

  Example of a FK3 error:
  - a remote module is disconnected from the CS31 system bus.

- An AC31 error of class 4 has occurred:
  - The block acknowledges the error on the AC31 remote module which signals the error and also clears the error message on the PLC, i.e. the error flag M 255,14 is reset.

  Example of a FK4 error:
  - a remote module signals an open circuit

Function call in IL
- CAL CS31QU1(FREI := CSQU_FREI)

Function call in ST
- CS31QU1(FREI := CSQU_FREI);
TESTING OF AC31 MODULES

This block checks whether the module defined at the block inputs is connected to the CS31 system bus.

### Block type

Function block without historical values

### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>CS31TE</td>
</tr>
<tr>
<td>AD</td>
<td>INT</td>
</tr>
<tr>
<td>KA</td>
<td>BOOL</td>
</tr>
<tr>
<td>SE</td>
<td>INT</td>
</tr>
<tr>
<td>RE</td>
<td>INT</td>
</tr>
<tr>
<td>OK</td>
<td>BOOL</td>
</tr>
<tr>
<td>AER</td>
<td>BOOL</td>
</tr>
<tr>
<td>DER</td>
<td>BOOL</td>
</tr>
</tbody>
</table>

- **Instance**: Instance name
- **AD**: Module address
- **KA**: Channel number
- **SE**: Number of bytes which the module sends to the master
- **RE**: Number of bytes which the module receives from the master
- **OK**: Module was detected on the bus
- **AER**: Error module address, address not available on the bus
- **DER**: Error data transfer range

### Description

This block checks whether the module defined at the block inputs is connected to the CS31 system bus.

- **AD** (INT): Module address
- **KA** (BOOL): Channel number
  - KA = FALSE -> Channel number <= 7
  - KA = TRUE -> Channel number > 7
- **SE** (INT): Number of bytes which the module sends to the master (value range 0...15)
  - **CAUTION**: If KT9x is used as slave the value 4 must be adjusted for the default setting.
  - Or:
  - Number of words which the module sends to the master (value range 100 ... 108).
- **RE** (INT): Number of bytes which the module receives from the master (value range 0...15)
  - **CAUTION**: If KT9x is used as slave the value 4 must be adjusted for the default setting.
  - Or:
  - Number of words which the module receives from the master (value range 100 ... 108).

### Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>BOOL</td>
</tr>
<tr>
<td>AER</td>
<td>BOOL</td>
</tr>
<tr>
<td>DER</td>
<td>BOOL</td>
</tr>
</tbody>
</table>

- **OK**: Module detected, connected to the bus
- **AER**: Error module address, address not available on the bus
- **DER**: Error data transfer range

The following is valid:

<table>
<thead>
<tr>
<th>AER</th>
<th>DER</th>
<th>OK</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>Module connected to bus</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>Module address not detected</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>Module address correct, incorrect data range</td>
</tr>
<tr>
<td>Other combinations are not possible</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


---

907 AC 1131/Issued: 06.02

CS31_S90 Library
Function call in IL

CAL CS31TE1(AD := CSTE_AD, KA := CSTE_KA, SE := CSTE_SE, RE := CSTE_RE)
LD CS31TE1.AER
ST CSTE_AER
LD CS31TE1.DER
ST CSTE_DER
LD CS31TE1.OK
ST CSTE_OK

Function call in ST

CS31TE1(AD := CSTE_AD, KA := CSTE_KA, SE := CSTE_SE, RE := CSTE_RE);
CSTE_AER:=CS31TE1.AER;
CSTE_DER:=CS31TE1.DER;
CSTE_OK:=CS31TE1.OK;

Note: In IL, the function call has to be performed in one line.
Glossary

BOOL

Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT

DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

DWORD

DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

INT

INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

WORD

WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

Function blocks

Function blocks are subroutines which can have as much inputs, outputs and internal variables as required. They are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. This allows for example to realize counters which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters. A function has maximally one output parameter. Note that the output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
Components of the CS31 library 2
CS31 S90
   CS31CO 3
   CS31QU 10
   CS31TE 11
CS31CO S90 3
CS31QU S90 10
CS31TE S90 11
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

Function Blocks for the Serial Communication 90 Series

Commun. 90 Series
Contents

The library for serial communication, 90 series 2
  PRECONDITIONS FOR THE USE OF THE COM LIBRARY ......................... 2
  COMPONENTS OF THE COM_S90 LIBRARY .................................. 2

Overview of blocks listed alphabetically according to their call names 3
  AUTOMATIC 907 AC 1131 LOGIN IDENTIFICATION IN THE PASSIVE MODE
  COMAUTOLOGIN .................. 4
  INITIALIZATION OF A SERIAL INTERFACE
  COMINIT....................... 6
  SENDING OF DATA VIA A SERIAL INTERFACE
  COMSND....................... 12
  RECEPTION OF DATA VIA A SERIAL INTERFACE
  COMREC....................... 18
  COMPARISON OF DATA AREAS
  WITH OPTIONAL WILDCARDS
  DATACMP ................. 24
  COPYING/MOVING DATA AREAS, INDEXED
  DATAMOV............... 27
  DETERMINING A WILDCARD COMPARISON VALUE
  IN A DATA TELEGRAM
  DATapos.................. 30
  INITIALIZATION AND CONFIGURATION
  OF THE MODBUS INTERFACES
  MODINIT................. 35
  OPERATION MODE MODBUS MASTER
  MODMAST .............. 40
  RECEIVING A DATA SET FROM THE
  COMMUNICATION MODULE 07 KP 92
  SIREC............... 53
  SENDING A DATA SET TO THE
  COMMUNICATION MODULE 07 KP 92
  SISEND............... 56

Glossary 59

Index 61
The library for serial communication, 90 series

PRECONDITIONS FOR THE USE OF THE COM LIBRARY

Note:
The blocks of the COM library only run in the RUN mode of the PLC, but not in the simulation mode.

COMPONENTS OF THE COM_S90 LIBRARY

The following function blocks are contained in the COM_S90 library:

Group: ASCII communication
Subgroup: COM
COMAUTOLOGIN - Automatic 907 AC 1131 LOGIN identification in the passive mode
COMINIT - Initialization of a serial interface in the "free mode"
COMREC - Reception of data via a serial interface in the "free mode"
COMSND - Sending of data via a serial interface in the "free mode"
Subgroup: DATA
DATACMP - Comparison of data areas with optional wildcards
DATAMOV - Copying/moving data areas, indexed
DATapos - Determining a wildcard comparison value in a data telegram

Group: MODBUS communication
MODINIT - Initialization and configuration of the MODBUS interfaces (serial interfaces and 07 KP 93)
MODMAST - Operation mode MODBUS master (serial interfaces and 07 KP 93)

Group: Serial coupler DPR
SIREC - Receiving a data set from the communication module 07 KP 92
SISEND - Sending a data set to the communication module 07 KP 92

Detailed descriptions of the blocks are in the following sections.
Overview of blocks listed alphabetically according to their call names

Used abbreviations:

- FBmV … Function block with historical values
- FBoV … Function block without historical values
- F … Function

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMAUTOLOGIN</td>
<td>F</td>
<td>Automatic 907 AC 1131 LOGIN identification in the passive mode</td>
<td>4</td>
</tr>
<tr>
<td>COMINIT</td>
<td>FBmV</td>
<td>Initialization of a serial interface in the &quot;free mode&quot;</td>
<td>6</td>
</tr>
<tr>
<td>COMREC</td>
<td>FBoV</td>
<td>Reception of data via a serial interface in the &quot;free mode&quot;</td>
<td>18</td>
</tr>
<tr>
<td>COMSND</td>
<td>FBmV</td>
<td>Sending of data via a serial interface in the &quot;free mode&quot;</td>
<td>12</td>
</tr>
<tr>
<td>DATACMP</td>
<td>F</td>
<td>Comparison of data areas with optional wildcards</td>
<td>24</td>
</tr>
<tr>
<td>DATAMOV</td>
<td>F</td>
<td>Copying/moving data areas, indexed</td>
<td>27</td>
</tr>
<tr>
<td>DATAPOS</td>
<td>F</td>
<td>Determining a wildcard comparison value in a data telegram</td>
<td>30</td>
</tr>
<tr>
<td>MODINIT</td>
<td>FBmV</td>
<td>Initialization and configuration of the MODBUS interfaces</td>
<td>35</td>
</tr>
<tr>
<td>MODMAST</td>
<td>FBmV</td>
<td>Operation mode MODBUS master</td>
<td>40</td>
</tr>
<tr>
<td>SIREC</td>
<td>FBoV</td>
<td>Receiving a data set from the communication module 07 KP 92</td>
<td>53</td>
</tr>
<tr>
<td>SISEND</td>
<td>FBmV</td>
<td>Sending a data set to the communication module 07 KP 92</td>
<td>56</td>
</tr>
</tbody>
</table>
By means of the COMAUTOLOGIN function, the automatic 907 AC 1131 login identification can be activated for a serial interface (EN = TRUE) or deactivated (EN = FALSE).

### Block type
- **Function**

### Parameters

<table>
<thead>
<tr>
<th>Input</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Auto Login identification ON / OFF (TRUE / FALSE)</td>
</tr>
<tr>
<td>COM</td>
<td>INT</td>
<td>Interface number, 1 or 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMAUTOLOGIN</td>
<td>INT</td>
<td>Return value, error</td>
</tr>
</tbody>
</table>

**Description** see next page
By means of the COMAUTOLOGIN function, the automatic 907 AC 1131 login identification can be activated for a serial interface (EN = TRUE) or deactivated (EN = FALSE). It is recommended only to activate the login identification, if it is really required. Otherwise, the communication procedures are slowed down unnecessarily. As a default, the login identification is deactivated.

It is also recommended, to keep the EN parameter constant, i.e. to keep the login identification permanently activated or deactivated. In addition, it is useful to insert the block in the user program directly after the initialization block belonging to it (MODINIT or COMINIT).

The return code of the function is 0 (COM_NO_ERROR), if COM = 1 or COM = 2, otherwise 20 (COM_INVALID_COM).

EN  

BOOL

EN   FALSE

If EN = FALSE, the COMAUTOLOGIN normally has no function. If, however, EN was TRUE before and 907 AC 1131 is/was logged in via the serial interface, the function reconfigures the interface after terminating the online session even if EN = FALSE. A renewed automatic login via the interface is no longer possible, because the automatic login identification was deactivated.

Function call in IL

LD   EN_COMAUTOLOGIN
COMAUTOLOGIN      COM_COMAUTOLOGIN
ST   RET_COMAUTOLOGIN

Note: In IL, the function call has to be performed in one line.

EN   TRUE

If EN = TRUE, the function compares with every call the characters received in the meantime with the pattern of the 907 AC 1131 login telegram. This comparison only takes place if the specified interface is running in the passive mode and 907 AC 1131 is not logged in over an other path. If a login telegram is detected, the interface will be configured as the 907 AC 1131 programming access and the login identification is skipped. From now on, it is checked continuously whether 907 AC 1131 is still logged in via this interface. If this is no longer the case, the original interface parameters are restored and the interface is switched over again to the passive mode.

COM  

INT

At the COM input, the number of the serial interface is specified.

COM = 1:    COM1
COM = 2:    COM2

COMAUTOLOGIN  

INT

The return code of the COMAUTOLOGIN is 0, if the COM input has a valid interface number (1 or 2). If an invalid number is specified, the function will no longer be carried out. It only returns the code of 20 (COM_INVALID_COM).

Function call in ST

RET_COMAUTOLOGIN =COMAUTOLOGIN
(EN := EN_COMAUTOLOGIN,
COM := COM_COMAUTOLOGIN);
INITIALIZATION OF A SERIAL INTERFACE IN THE "FREE MODE"  COMINIT

The COMINIT function block is used for initialization of a serial interface employing the sending and receiving function blocks COMSND/COMREC. With one project, only one instance of the function block may be used per interface.

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Input / Output</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>COMINIT</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enable for processing the block</td>
</tr>
<tr>
<td>COM</td>
<td>INT</td>
<td>Interface number (COM1 or COM2)</td>
</tr>
<tr>
<td>BAUD</td>
<td>INT</td>
<td>Baud rate (300...19200 bits/s)</td>
</tr>
<tr>
<td>CH_LEN</td>
<td>INT</td>
<td>Character length (5, 6, 7 or 8 bits)</td>
</tr>
<tr>
<td>STOP</td>
<td>INT</td>
<td>Number of stop bits (1 or 2)</td>
</tr>
<tr>
<td>PTY</td>
<td>INT</td>
<td>Parity bit (none, even, odd, always 1 or always 0)</td>
</tr>
<tr>
<td>RTSCTRL</td>
<td>BOOL</td>
<td>Direction control (off/on)</td>
</tr>
<tr>
<td>TLS</td>
<td>INT</td>
<td>Carrier leading time in milliseconds (0 – 500ms)</td>
</tr>
<tr>
<td>CDLY</td>
<td>INT</td>
<td>Carrier lagging time in milliseconds (0 – 100ms)</td>
</tr>
<tr>
<td>MODE</td>
<td>INT</td>
<td>Mode of telegram end (terminating character(s), receiving pause, total time or number of characters)</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Readiness for operation of the selected interface</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description see next page
The COMINIT function block is used for initialization of a serial interface employing the sending and receiving function blocks COMSND/COMREC. With one project, only one instance of the function block may be used per interface.

The values of the input parameters come into effect with a FALSE->TRUE edge at the EN input. As long as the EN input remains TRUE, the interface is run in the "free mode" then. The outputs indicate errors which occur during initialization and current operation. If EN = FALSE, the interface is free for other modes.

The inputs can neither be duplicated nor negated/inverted.

Telegram end identifier
The telegram end identifier is used for identifying received telegrams or signaling the telegram termination when sending data to the counter-station. Since the kind of telegram termination differs from device to device (e.g. using a defined ASCII character or an agreed number of characters), it can be set at the MODE input which event shall be interpreted as the telegram termination. This agreement is project-globally valid for all sending and receiving operations via an interface. The values belonging to it can be defined separately for every sending or receiving operation. In the following explanations, it is assumed that the input buffer does not overflow, i.e. the telegram end event occurs before 256 bytes were received and read. Otherwise, an error message is generated (see also COMREC block).

**MODE = 0**
**NO TELEGRAM END IDENTIFICATION**

<table>
<thead>
<tr>
<th>Sending</th>
<th>Receiving</th>
</tr>
</thead>
</table>
| no effect | All received characters are output immediately.

**MODE = 1**
**TELEGRAM END = SPECIAL CHARACTER**

<table>
<thead>
<tr>
<th>Sending</th>
<th>Receiving</th>
</tr>
</thead>
<tbody>
<tr>
<td>The end-of-telegram character, specified at the END.CH input of a COMSND sending block, is automatically added on the telegram last.</td>
<td>The special character is defined at the END input of a COMREC receiving block. If the block receives this character, the telegram is recognized to be complete. The telegram is output together with the end-of-telegram character.</td>
</tr>
</tbody>
</table>

**MODE = 2**
**TELEGRAM END = TELEGRAM LENGTH**

<table>
<thead>
<tr>
<th>Sending</th>
<th>Receiving</th>
</tr>
</thead>
<tbody>
<tr>
<td>no effect</td>
<td>The number of characters is specified at the END input of a COMREC receiving block. The telegram is recognized to be complete when the given number of characters has been received. The telegram is output then.</td>
</tr>
</tbody>
</table>

**MODE = 3**
**TELEGRAM END = TIME INTERVAL**

<table>
<thead>
<tr>
<th>Sending</th>
<th>Receiving</th>
</tr>
</thead>
<tbody>
<tr>
<td>no effect</td>
<td>The time interval is specified in ms at the END input of a COMREC receiving block. When the time interval has elapsed, all characters received up to this moment are output.</td>
</tr>
</tbody>
</table>
MODE = 4

Sending
no effect

TELEGRAM END = PAUSE

Receiving
The pause length is specified at the END input of the receiving block. The pause monitoring starts with the reception of the first character and ends, when a pause of the specified length was recognized. All received characters are output now.

Use of modems

The COMINIT block takes the special features of modems, interface converters and repeaters into account. If these devices are used at a serial interface operating in the "free mode", their possibly supported compressing mechanisms must be deactivated. For more information see also the manuals of the corresponding devices.

In order to set the transmission direction, some repeaters, modems or interface converters require a control signal. This direction control can be switched on/off with the RTSCTRL input.

In addition, some devices need a leading time (settling time) in order to stabilize their carrier signal. Only when this time has elapsed, the devices are ready for data transmission in sending direction. This settling time can be set at the TLS input.

Some devices require that the carrier signal in sending direction is kept stable for a while after data transmission. Only when this lagging time has elapsed, a complete data transmission is guaranteed and the devices are ready for data transmission in the opposite direction. The lagging time can be specified at the CDLY input.

Note: Leading and lagging times must be matched between all communication partners connected to the same data transmission line.

EN  BOOL
EN   FALSE → TRUE:
With a FALSE->TRUE edge, the values of the inputs come into effect. It is recommended, to set the inputs to constant values. The block tries to configure the COM interface (specified at the COM input) with the values of the input parameters. If this configuration procedure fails, the ERNO output indicates an error. The current setting remains unchanged then.

EN   TRUE:
As long as no error occurred during initialization (ERNO output), the interface is activated in the "free mode". If 907 AC 1131 logs-in temporarily, the block is locked during the online session. After logging out 907 AC 1131 and after the following automatic re-initialization, the block returns to its normal function. The enable input EN of the block must still be TRUE.

EN  TRUE → FALSE:
The interface is deactivated. It can be used for other applications. The RDY output is set to FALSE. The ERNO output is set to COM_NO_ERROR.

EN   FALSE
The block is not processed. The interface is not operated in the "free mode".

COM  INT
At the COM input, the number of the serial interface is specified.
COM = 1:  COM1
COM = 2:  COM2

BAUD  INT
At the BAUD input, the data transmission speed is set in bits per second. The following settings are possible:
300, 1200, 2400, 4800, 9600, 14400 or 19200 bits/s
CH_LEN INT
At the CH_LEN input, the character transmission length is specified in bits. The following settings are possible:
5, 6, 7 or 8 bits per character

Note: If the character length is set to 5 bits (CH_LEN = 5) and 2 stop bits are set at the same time, the UART configures itself to 1.5 stop bits.

STOP INT
At the STOP input, the number of stop bits per character is given. The following settings are possible:
1 or 2 stop bits per character

PTY INT
At the PTY input, the parity bit type is defined. The settable parity (PTY) is encoded in the following way:
PTY = 0: No parity bit
PTY = 1: Odd parity. The number of character bits owing 1 is complemented so that the total number is odd.
PTY = 2: Even parity. The number of character bits owing 1 is complemented so that the total number is even.
PTY = 3 Parity bit is always 1
PTY = 4 Parity bit is always 0

RTS_CTRL BOOL
At the RTSCTRL input, it is specified whether the synchronization between the interface and the connected device(s) is to be carried out with the RTS direction control signal.
RTSCTRL = TRUE RTS is used
RTSCTRL = FALSE RTS is unused

With the activated RTS mechanism (RTSCTRL = TRUE), the required control signals are automatically generated by the interface.

TLS INT
At the TLS input, the carrier leading time (settling time) is specified in ms (0...500 ms).

The TLS input is only processed, if RTS_CTRL = TRUE. The realization of the leading time depends on the cycle time of the task in which the COMINIT block is processed. Under worst-case conditions, the real leading time can have a deviation of (one task cycle - 1 ms) referred to the specification.

CDLY INT
At the CDLY input, the carrier lagging time is specified in ms (0...100 ms).

The CDLY input is only processed, if RTS_CTRL = TRUE. The realization of the leading time depends on the cycle time of the task in which the COMINIT block is processed. Under worst-case conditions, the real lagging time can have a deviation of (one task cycle - 1 ms) referred to the specification.

MODE INT
At the MODE input, the type of the telegram end identification is set. The values belonging to it can be defined at the corresponding sending and receiving blocks. MODE is encoded as follows:
MODE = 0 no telegram end identification
MODE = 1 Telegram end by special character
MODE = 2 Telegram end by telegram length
MODE = 3 Telegram end by time interval
MODE = 4 Telegram end by receiving pause (character timeout)

RDY BOOL
At the READY output, it is indicated that the interface is ready for operation. RDY has always to be considered together with the ERNO output.
The ERNO output indicates errors occurred. The following table explains the meaning of the ERNO codes:

<table>
<thead>
<tr>
<th>ERNO</th>
<th>INT</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>COM_NO_ERROR</td>
<td>No error occurred. The interface is operated as configured.</td>
</tr>
<tr>
<td>20</td>
<td>COM_INVALID_COM</td>
<td>Invalid value at the COM input</td>
</tr>
<tr>
<td>51</td>
<td>COM_INVALID_BAUD_RATE</td>
<td>Invalid baud rate at the BAUD input</td>
</tr>
<tr>
<td>52</td>
<td>COM_INVALID_PARITY</td>
<td>Invalid parity at the PTY input</td>
</tr>
<tr>
<td>53</td>
<td>COM_INVALID_NUM_STOP_BITS</td>
<td>Invalid number of stop bits at the STOP input</td>
</tr>
<tr>
<td>54</td>
<td>COM_TLS_OVERRIDE</td>
<td>Invalid carrier leading time at the TLS input</td>
</tr>
<tr>
<td>55</td>
<td>COM_CDLY_OVERRIDE</td>
<td>Invalid carrier lagging time at the CDLY input</td>
</tr>
<tr>
<td>56</td>
<td>COM_CDLY_EXCEEDS_TLS</td>
<td>Carrier lagging time is greater than carrier leading time</td>
</tr>
<tr>
<td>59</td>
<td>COM_6_8_DISCONNECT</td>
<td>No connection between the pins 6 and 8 of the COM interface. If currently no serial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>programming interface is defined for 907 AC 1131, the COM interface is temporarily set up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>as a programming interface.</td>
</tr>
<tr>
<td>62</td>
<td>COM_RECONFIGURED</td>
<td>Interface was reconfigured with other Init blocks (e.g. MODINIT re-configured). It is no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>longer operated in the way set with COMINIT.</td>
</tr>
<tr>
<td>65</td>
<td>COM_AC1131_LOGIN</td>
<td>907 AC 1131 is logged in via this interface.</td>
</tr>
<tr>
<td>70</td>
<td>COM_INVALID_CHAR_LENGTH</td>
<td>Invalid number of bits (or characters) specified at the CH_LEN input</td>
</tr>
<tr>
<td>71</td>
<td>COM_INVALID_TELEGRAM_END</td>
<td>Invalid end identification at the MODE input</td>
</tr>
<tr>
<td>65535</td>
<td>COM_AC1131_REMOTE_ACCESS</td>
<td>907 AC 1131 is temporarily logged in</td>
</tr>
</tbody>
</table>
### Function call in IL

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>INIT_COM</td>
</tr>
<tr>
<td>(EN)</td>
<td>:= EN_COMINIT,</td>
</tr>
<tr>
<td>COM</td>
<td>:= COM_COMINIT,</td>
</tr>
<tr>
<td>BAUD</td>
<td>:= BAUD_COMINIT,</td>
</tr>
<tr>
<td>CH_LEN</td>
<td>:= CH_LEN_COMINIT,</td>
</tr>
<tr>
<td>STOP</td>
<td>:= STOP_COMINIT,</td>
</tr>
<tr>
<td>PTY</td>
<td>:= PTY_COMINIT,</td>
</tr>
<tr>
<td>RTSCTRL</td>
<td>:= RTSCTRL_COMINIT,</td>
</tr>
<tr>
<td>TLS</td>
<td>:= TLS_COMINIT,</td>
</tr>
<tr>
<td>CDLY</td>
<td>:= CDLY_COMINIT,</td>
</tr>
<tr>
<td>MODE</td>
<td>:= MODE_COMINIT)</td>
</tr>
</tbody>
</table>

### Function call in ST

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT_COM</td>
<td></td>
</tr>
<tr>
<td>(EN)</td>
<td>:= EN_COMINIT,</td>
</tr>
<tr>
<td>COM</td>
<td>:= COM_COMINIT,</td>
</tr>
<tr>
<td>BAUD</td>
<td>:= BAUD_COMINIT,</td>
</tr>
<tr>
<td>CH_LEN</td>
<td>:= CH_LEN_COMINIT,</td>
</tr>
<tr>
<td>STOP</td>
<td>:= STOP_COMINIT,</td>
</tr>
<tr>
<td>PTY</td>
<td>:= PTY_COMINIT,</td>
</tr>
<tr>
<td>RTSCTRL</td>
<td>:= RTSCTRL_COMINIT,</td>
</tr>
<tr>
<td>TLS</td>
<td>:= TLS_COMINIT,</td>
</tr>
<tr>
<td>CDLY</td>
<td>:= CDLY_COMINIT,</td>
</tr>
<tr>
<td>MODE</td>
<td>:= MODE_COMINIT)</td>
</tr>
</tbody>
</table>

LD INIT_COM.RDY
ST RDY_COMINIT
LD INIT_COM.ERNO
ST ERNO_COMINIT

Note: In IL, the function call has to be performed in one line.
SENDING OF DATA VIA A SERIAL INTERFACE IN THE "FREE MODE"  

The COMSND block is used for sending data via a serial interface. The number of COMSND blocks used in the same project is not limited. It is also possible to use the blocks in different user tasks. A sending operation is started with a FALSE->TRUE edge at the EN input. Specifying the memory address of the data to be sent instead of the values itself makes a telegram structure possible without any format restrictions. The length of a data block to be sent is not limited. Since, however, the data only can be sent as long as the sending buffer has enough free space, it is recommended to copy only data blocks to the sending buffer with a maximum of 256 bytes in size. By using COMSND blocks adjacent to one another and ignoring their RDY outputs it is also possible to generate longer telegrams if necessary.

Block type
Function block with historical value

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>COMSND</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enable sending with a FALSE -&gt; TRUE edge</td>
</tr>
<tr>
<td>COM</td>
<td>INT</td>
<td>Interface number (COM1 or COM2)</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Memory address of data to be sent via ADR operator</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>Length of data to be sent as of the DATA address in bytes</td>
</tr>
<tr>
<td>END_CH</td>
<td>INT</td>
<td>Value for telegram end, depending on the set telegram mode at the COMINIT input (MODE):</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Sending buffer is empty</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description

The COMSND block is used for sending data via a serial interface. The number of COMSND blocks used in the same project is not limited. It is also possible to use the blocks in different user tasks. A sending operation is started with a FALSE->TRUE edge at the EN input. Specifying the memory address of the data to be sent instead of the values itself makes a telegram structure possible without any format restrictions. The length of a data block to be sent is not limited. Since, however, the data only can be sent as long as the sending buffer has enough free space, it is recommended to copy only data blocks to the sending buffer with a maximum of 256 bytes in size. By using COMSND blocks adjacent to one another and ignoring their RDY outputs it is also possible to generate longer telegrams if necessary.

The inputs can neither be duplicated nor negated/inverted.
Telegram length

The maximum length of a sending telegram can be controlled by evaluating the RDY output. RDY = TRUE indicates an empty sending buffer. If a sending process always is started with RDY = TRUE, a single telegram is sent per COMSND each. If RDY is ignored, data streams of any length are possible (theoretically). See also the figure below.

If it was planned with COMINIT direction control (and carrier leading and lagging signals), the carrier lagging time is automatically adjusted, i.e. the carrier lagging time is triggered when the sending buffer has no more data to be sent.

Data to be sent in buffer

Sending buffer

Real telegram end block 1 + block 2

Theor. telegram end block 1

Theor. telegram start block 2

Real telegram end block 1 + block 2

Data to be sent in buffer

Sending buffer

COMSND ignores RDY
COMSND is waiting for RDY = TRUE

COMSND ignores RDY

**EN**

**FALSE->TRUE**

With a FALSE->TRUE edge at the EN input, it is tried to send the telegram specified by the block inputs. If an error is detected while checking the input parameters, this is indicated at the ERNO output. In case of an error, no data transmission takes is carried out.

Independent of the status of the EN input, the block is disabled as long as 907 AC 1131 is logged in temporarily via the COM interface. After logging out 907 AC 1131 and the following automatic re-initialization of the interface, the block returns to the normal function. The interface must be still active (COMINT EN = TRUE).
DATA DWORD

At the DATA input, the start address of the data to be sent is specified by an ADR operator. Sending data can be both operand values and variable values. If binary values have to be transmitted, some special things must be taken into account.

If IEC bit operators are used as telegram start addresses, only such operands are permitted which end with an "0" (e.g. %QX62.0 is allowed, %QX62.1 is forbidden). If this condition is ignored, the block generates an error message. The restriction mentioned can be got round if necessary when the IEC bit operands are copied into variables before the transmission and as the telegram start address is given the first variable then.

IEC bit operands %IX62.0 - %IX62.7 direct
(DATA = ADR(%IX62.0), LEN = 1)

Symbolic representation of the operands %IX62.0 - %IX62.7

<table>
<thead>
<tr>
<th>TRUE₀</th>
<th>FALSE₁</th>
<th>TRUE₂</th>
<th>TRUE₃</th>
<th>TRUE₄</th>
<th>FALSE₅</th>
<th>FALSE₆</th>
<th>TRUE₇</th>
</tr>
</thead>
</table>

Position of the operands %IX62.0 - %IX62.7 in the memory

\[ \text{7100111010} \]

Transmission

\[ \text{7100111010} \]

IEC bit operands %IX62.0 - %IX62.7 unpacked into BIN_VAL: ARRAY [0..7] OF BOOL
(DATA = ADR(BIN_VAL[0]), LEN = 8)

Symbolic representation of the operands %IX62.0 - %IX62.7

<table>
<thead>
<tr>
<th>TRUE₀</th>
<th>FALSE₁</th>
<th>TRUE₂</th>
<th>TRUE₃</th>
<th>TRUE₄</th>
<th>FALSE₅</th>
<th>FALSE₆</th>
<th>TRUE₇</th>
</tr>
</thead>
</table>

Position of the operands %IX62.0 - %IX62.7 in the memory

\[ \text{7100111010} \]

Position of the BOOLean variables BIN_VAL[0] - BIN_VAL[7] in the memory

\[ \begin{align*}
00000001₀ & 00000000₁ & 00000001₂ & 00000001₃ & 00000001₄ & 00000000₅ & 00000000₆ & 00000001₇ \\
\end{align*} \]

Transmission

\[ \begin{align*}
\text{00000001₀} & \text{00000000₁} & \text{00000001₂} & \text{00000001₃} & \text{00000001₄} & \text{00000000₅} & \text{00000000₆} & \text{00000001₇} \\
\end{align*} \]
BOOLean variables in BIN_VAL: ARRAY [0..7] OF BOOL direct
(DATA = ADR(BIN_VAL[0]), LEN = 8)

Symbolic representation of the variables BIN_VAL[0] - BIN_VAL[7]

<table>
<thead>
<tr>
<th>TRUE_0</th>
<th>FALSE_1</th>
<th>TRUE_2</th>
<th>TRUE_3</th>
<th>TRUE_4</th>
<th>FALSE_5</th>
<th>FALSE_6</th>
<th>TRUE_7</th>
</tr>
</thead>
</table>

Position of the BOOLean variables BIN_VAL[0] - BIN_VAL[7] in the memory

- 00000001_0 00000001_1 00000001_2 00000001_3 00000001_4 00000000_5 00000000_6 00000000_7

BOOLean variables in BIN_VAL: ARRAY [0..7] OF BOOL packed in BIN_PACK: BYTE
(DATA = ADR(BIN_PACK), LEN = 1)

Symbolic representation of the variables BIN_VAL[0] - BIN_VAL[7]

<table>
<thead>
<tr>
<th>TRUE_0</th>
<th>FALSE_1</th>
<th>TRUE_2</th>
<th>TRUE_3</th>
<th>TRUE_4</th>
<th>FALSE_5</th>
<th>FALSE_6</th>
<th>TRUE_7</th>
</tr>
</thead>
</table>

Position of the BOOLean variables BIN_VAL[0] - BIN_VAL[7] in the memory

- 00000001_0 00000001_1 00000001_2 00000001_3 00000001_4 00000000_5 00000000_6 00000000_7

Position of the BYTE variables BIN_PACK in the memory

- \_10011101_0 = 157

LEN INT
At the LEN input, the length of data to be sent is specified in bytes, without an possibly existing end character. The LEN value comes into effect with a FALSE->TRUE edge at the EN input.

END_CH INT
The value of the telegram end character is specified at the END_CH input. This character is added to the sending data. END_CH is only evaluated, if at the COMINIT initialization block the telegram end identifier mode was planned with "Telegram End Character" for the corresponding interface (MODE = 1). For END_CH, only values between 0 and 255 are valid. END_CH comes into effect with a FALSE->TRUE edge at the EN input. If necessary, the end of telegram character can be varied from telegram to telegram in this way.

RDY BOOL
Independent of the status of the EN input, the RDY output indicates whether the sending buffer is absolutely empty (RDY = TRUE) or not (RDY = FALSE). A precondition is that the interface was successfully configured as a "free interface" before. It also must be active. RDY has always to be considered together with the ERNO output.
SENDING OF DATA VIA A SERIAL INTERFACE IN THE "FREE MODE" COMSND

ERNO INT
The ERNO output indicates occurred errors. The output values are encoded as follows:

<table>
<thead>
<tr>
<th>ERNO</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>COM_NO_ERROR No error. Interface is operated as configured.</td>
</tr>
<tr>
<td>15</td>
<td>COM_INIT_ERROR Initialization error. The interface has not been initialized correctly.</td>
</tr>
<tr>
<td>20</td>
<td>COM_INVALID_COM Invalid value at the COM input.</td>
</tr>
<tr>
<td>59</td>
<td>COM_6_8_DISCONNECT No connection between the pins 6 and 8 of the COM interface.</td>
</tr>
<tr>
<td>63</td>
<td>COM_DIFFERENT_CONFIG Interface is not configured in the &quot;free mode&quot;.</td>
</tr>
<tr>
<td>69</td>
<td>COM_INVALID_DATA_ADDRESS The address at DATA is invalid</td>
</tr>
<tr>
<td>71</td>
<td>COM_INVALID_TELEGRAM_END Invalid end identifier. End character planned. END_CH is less than 0 or greater than 255</td>
</tr>
<tr>
<td>72</td>
<td>COM_TELEGRAM_SIZE.Override LEN is less than 0 or greater than the free buffer area</td>
</tr>
<tr>
<td>65535</td>
<td>COM_AC1131_REMOTE_ACCESS 907 AC 1131 temporarily logged-in.</td>
</tr>
</tbody>
</table>

Function call in IL
LD DATA_COMSND
ADR
ST SEND_COM.DATA
CAL SEND_COM (EN := EN_COMSND, COM := COM_COMSND, DATA := ADR(DATA_COMSND), LEN := LEN_COMSND, END_CH := END_CH_COMSND)
LD SEND_COM.RDY
ST RDY_COMSND
LD SEND_COM.ERNO
ST ERNO_COMSND

Function call in ST
SEND_COM (EN := EN_COMSND, COM := COM_COMSND, DATA := ADR(DATA_COMSND), LEN := LEN_COMSND, END_CH := END_CH_COMSND);
RDY_COMSND := SEND_COM.RDY;
ERNO_COMSND := SEND_COM.ERNO;

Note: In IL, the function call has to be performed in one line.
RECEPTION OF DATA VIA A SERIAL INTERFACE IN THE "FREE MODE"    COMREC

The COMREC function block is used for receiving data from a serial interface in the "free mode". The number of COMREC blocks used in the same project is not limited. It is also possible to use the blocks in different user tasks. It is important that the blocks must be interlocked to one another, i.e. only one block may be active at the same time. In order to avoid that telegram parts are lost or evaluated incorrectly or not at all, an activity change between two COMREC blocks may only be carried out when the block to be deactivated has signalized that it has completely evaluated the received telegram (RDY = TRUE).

Here it must be made sure that at first all instances of the block are deactivated before the activity change can be performed. In order to avoid competence conflicts, it is strongly recommended to use only one COMREC block within one project.

Block type

- Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Input / Output</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>COMREC</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enable sending by a FALSE -&gt; TRUE edge</td>
</tr>
<tr>
<td>COM</td>
<td>INT</td>
<td>Interface number (COM1 or COM2)</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Memory address for the receiving data via ADR operator</td>
</tr>
<tr>
<td>END</td>
<td>INT</td>
<td>Value for telegram end identification, depending on the set telegram mode at the COMINIT input (MODE): Termination character value of the end character Telegram pause Minimum length of the receiving pause in ms Time Time value in ms (as of last receiving) Number of characters Telegram length in No. of characters RDY</td>
</tr>
</tbody>
</table>

Description

The COMREC function block is used for receiving data from a serial interface in the "free mode". The number of COMREC blocks used in the same project is not limited. It is also possible to use the blocks in different user tasks. It is important that the blocks must be interlocked to one another, i.e. only one block may be active at the same time. In order to avoid that telegram parts are lost or evaluated incorrectly or not at all, an activity change between two COMREC blocks may only be carried out when the block to be deactivated has signalized that it has completely evaluated the received telegram (RDY = TRUE).

Here it must be made sure that at first all instances of the block are deactivated before the activity change can be performed. In order to avoid competence conflicts, it is strongly recommended to use only one COMREC block within one project.
Since the block enable input EN works level-dependent and not edge-triggered, the block checks the values at the inputs with every call for validity, as long as EN = TRUE.

Specifying a memory address for the receiving data makes a telegram structure possible without any format restrictions. The length of a receiving data block is limited to a maximum of 256 bytes. It is indicated at the LEN output.

Note: During the planning phase, it must be kept in mind that there is enough memory space free for storing the receiving data, (e.g. ARRAY [1..256] OF BYTE). The memory space begins at the DATA address.

If a valid receiving telegram exists in the memory area as of DATA, this is always indicated by RDY = TRUE. The behavior of RDY as a function of time is here dependent on the ratio of the task cycle-time related to the rate of incoming telegrams. If receiving telegrams come in in shorter distances than the cycle time, RDY is TRUE in every cycle. If the distance between the telegrams is longer than the task cycle time, at first RDY indicates the reception of a telegram by a FALSE->TRUE edge, then (beginning with the next cycle) RDY goes back to FALSE for a certain number of cycles. After that, the reception of another telegram is indicated by a new FALSE->TRUE edge. Important: It is strongly recommended to evaluate RDY level-dependent and not edge-triggered.

Independent of the status at the EN input, the block is disabled as long as 907 AC 1131 is logged in temporarily. After logging out 907 AC 1131 and the following re-initialization of the interface, the block returns to its normal function. The interface must be still active (COMINT.EN = TRUE).

The inputs can neither be duplicated nor negated/inverted.

Receiving errors
If receiving errors occur, they will be recognized and indicated by the block. The block detects overflow, parity and framing errors. If at least one of these errors has occurred, the block deletes the receiving buffer.

In this case, the interface parameters of the communication partners must be checked (for e.g. baud rate, character length, number of stop bits and parity).

Times
If the telegram end identifier was selected as "time interval" (MODE = 3) or as "receiving pause" (MODE = 4), the times are monitored by the block. In MODE = 3, the monitoring time defined at END is triggered cyclically. The runtime system counts up the time in steps of 5 ms. Because of this, it is possible that the real waiting time is longer than the defined time (by up to 5 ms plus the cycle time of the task which called the block). The receiving pause identification, however, is carried out with a resolution of 1 millisecond.
RECEPTION OF DATA VIA A SERIAL INTERFACE IN THE "FREE MODE" COMREC

**EN** **BOOL**

EN FALSE

If EN = FALSE, the block is not operated.

EN TRUE

If EN = TRUE, the block checks the receiving buffer for a telegram (end identification) specified at the block inputs with every call.

**COM** **INT**

At the COM input, the number of the serial interface is specified.

COM = 1: COM1

COM = 2: COM2

**DATA** **DWORD**

At the DATA input, the start address for storing the receiving data is specified by an ADR operator. Received data can be stored in the operand area as well as in variables. When binary values are received, some special things must be considered.

If IEC bit operators are used as memory address, only such operands are permitted which end with an "0" (e.g. %QX62.0 is allowed, %QX62.1 is forbidden). If this condition is ignored, the block generates an error message.

When a receiving telegram is stored in the IEC bit operand area, it must be taken into account that a received byte contains 8 bit operands. If, however, a received byte is stored in a BOOLean variable, this variable is FALSE (byte value = 0) or TRUE (byte value unequal 0).

Received byte

| TRUE0 | FALSE1 | TRUE2 | TRUE3 | TRUE4 | FALSE5 | FALSE6 | TRUE7 |

Byte stored in 8 IEC bit operands %MX62.0 - %MX62.7 direct

(DATA = ADR(%MX62.0))

**TRUE**

Byte stored in a BOOLean variable BIN: BOOL

(DATA = ADR(BIN))

**END** **INT**

The value for the telegram end identification is specified at the END input. The evaluation of END is performed dependent on the MODE set at COMINIT. The value at END is evaluated with every block call. So it is possible to change the value at any time, even during a reception process.

If MODE = 0, END is ignored. Received characters are output immediately.

If MODE = 1 (telegram end = special character), END is to be set to the character which marks a telegram end. In this case, END must have a value from 1 to 255.

If MODE = 2 (telegram end by telegram length), END is to be set to the number of expected bytes. END must have a value from 1 to 256.

If MODE = 3 (telegram end by time interval), END must be specified in milliseconds. After this time has elapsed, all received characters are output. Allowed values: 10 to 10,000 ms. The time monitoring is triggered again with every new cycle.

If MODE = 4 (telegram end by receiving pause / character timeout), END must be specified with the maximum allowed receiving pause between two characters in ms. Valid values: 1 to 500 ms. The time monitoring is triggered repeatedly with every received character.
RECEPTION OF DATA VIA A SERIAL INTERFACE IN THE "FREE MODE" COMREC

RDY BOOL

The RDY output of the block indicates, whether a telegram was received (RDY = TRUE) or not (RDY = FALSE) since the last operation of the COMREC block. A telegram is interpreted as received, when the condition given at END is satisfied. Since an error also could have been occurred during reception (e.g. telegram length is greater than the buffer), RDY must always be considered together with the ERNO output.

ERNO INT

The ERNO output indicates errors occurred. There are the following error codes:

<table>
<thead>
<tr>
<th>ERNO</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>COM_NO_ERROR No error. Interface is operated as configured.</td>
</tr>
<tr>
<td>15</td>
<td>COM_INIT_ERROR Initialization error. Interface was not initialized correctly.</td>
</tr>
<tr>
<td>20</td>
<td>COM_INVALID_COM Invalid value at COM input.</td>
</tr>
<tr>
<td>57</td>
<td>COM_CHAR_TO_CHAR_TIMEOUT_OVERRIDE Telegram end identification = receiving pause (COMINIT MODE = 4), value for the time at END is less than 1 or greater than 500</td>
</tr>
<tr>
<td>59</td>
<td>COM_6_8_DISCONNECT No connection between pins 6 and 8 at the COM interface</td>
</tr>
<tr>
<td>63</td>
<td>COM_DIFFERENT_CONFIG Interface is not configured for the &quot;free mode&quot;</td>
</tr>
<tr>
<td>67</td>
<td>COM_DRV_RECEPTION_PARITY_OVERRUN_FRAMING_ERROR Reception error. Received data do not match the settings given at COMINIT concerning baud rate, parity, number of stop bits and/or bits per character.</td>
</tr>
<tr>
<td>69</td>
<td>COM_INVALID_DATA_ADDRESS Address at DATA is invalid</td>
</tr>
<tr>
<td>71</td>
<td>COM_INVALID_TELEGRAM_END Invalid end identification: COMINIT MODE = 1 (end character) configured, value of the character at END is less than 0 or greater than 255</td>
</tr>
<tr>
<td>72</td>
<td>COM_TELEGRAM_SIZE_OVERRIDE Received more characters than planned or more than 256. The meaning depends on COMINIT MODE, for explanations see the table below</td>
</tr>
<tr>
<td>73</td>
<td>COM_TIMEOUT_OVERRIDE Telegram end identification: time interval (COMINIT MODE = 3) configured, value for the time at END is less than 10 or greater than 10,000</td>
</tr>
<tr>
<td>75</td>
<td>COM_INVALID_DATA_ADDRESS Invalid address for data block</td>
</tr>
<tr>
<td>65535</td>
<td>COM_AC1131_REMOTE_ACCESS 907 AC 1131 temporarily logged in</td>
</tr>
</tbody>
</table>
LEN

The LEN output indicates the length of the received data in bytes inclusive the possibly existing end character. LEN is only valid if RDY = TRUE. LEN has always to be considered together with the ERNO output. ERNO indicates whether the telegram is genuine and complete according to the settings (MODE, END), see the table below.

<table>
<thead>
<tr>
<th>MODE</th>
<th>Parameters / Status</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 0, 1, 2, 3, 4 | RDY = FALSE  
ERNO = 0  
LEN = 0 | No telegram detected (yet) which meets the conditions concerning the telegram end identification |
| 0 | RDY = TRUE  
ERNO = 0  
LEN = Number of characters | There are between 1 and 256 characters in the receiving buffer. The telegram is stored as of the DATA address. |
| 0 | RDY = TRUE  
ERNO = 72  
LEN = 256 | More than 256 characters in the receiving buffer. The first 256 bytes of the telegram are stored as of the DATA address. The left characters stay in buffer. They are picked up in the next cycle. |
| 1 | RDY = TRUE  
ERNO = 0  
LEN = Number of characters  
(incl. end character) | End character END received and 1 to 256 characters (incl. end character) present in receiving buffer. The telegram inclusive the end character is stored as of the DTA address. |
| 1 | RDY = TRUE  
ERNO = 72  
LEN = 256 | End character END was not received, but there are 256 or more characters in the buffer. The first 256 bytes of the telegram are stored as of the DATA address. The left characters stay in buffer at first. They can be picked up in the following cycles. |
| 2 | RDY = TRUE  
ERNO = 0  
LEN = Number of characters (= END) | Exactly the number of characters are received as specified at END. The telegram is stored as of the DTA address. |
| 2 | RDY = TRUE  
ERNO = 72  
LEN = Number of characters (= END) | More characters than specified at END were received. The first Number of Characters Byte of the telegram are stored as of the DATA address. The left characters stay in buffer at first. They can be picked up in the following cycles. |
<table>
<thead>
<tr>
<th>MODE</th>
<th>Parameters / Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>RDY = TRUE</td>
<td>The time specified at the END input has expired. Within this period, up to 256 characters were received.</td>
</tr>
<tr>
<td></td>
<td>ERNO = 0</td>
<td>The telegram is stored as of the DTA address.</td>
</tr>
<tr>
<td></td>
<td>LEN = Number of characters (0 is also possible)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RDY = TRUE</td>
<td>The time specified at the END input has expired. Within this period of time, more than 256 characters were received.</td>
</tr>
<tr>
<td></td>
<td>ERNO = 72</td>
<td>The first 256 bytes of the telegram are stored as of the DATA address. The left characters stay in buffer at first. They can be picked up in the following cycles.</td>
</tr>
<tr>
<td></td>
<td>LEN = 256</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>RDY = TRUE</td>
<td>The pause between the reception of two characters specified at the END input has expired. 1 to 256 characters were received within this period of time.</td>
</tr>
<tr>
<td></td>
<td>ERNO = 0</td>
<td>The telegram is stored as of the DTA address.</td>
</tr>
<tr>
<td></td>
<td>LEN = Number of characters</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>RDY = TRUE</td>
<td>The pause between the reception of two characters specified at the END input has expired. More than 256 characters were received within this period of time.</td>
</tr>
<tr>
<td></td>
<td>ERNO = 72</td>
<td>The first 256 bytes of the telegram are stored as of the DATA address. The left characters stay in buffer at first. They can be picked up in the following cycles.</td>
</tr>
<tr>
<td></td>
<td>LEN = 256</td>
<td></td>
</tr>
</tbody>
</table>

**Function call in IL**

```
LD DATA_COMREC  
ADR             
ST REC_COM.DATA 
CAL REC_COM     
(EN := EN_COMREC,  
COM := COM_COMREC,  
END := END_COMREC) 
```

**Function call in ST**

```
REC_COM         
(EN := EN_COMREC,  
COM := COM_COMREC,  
DATA := ADR(DATA_COMREC),  
END := END_COMREC); 
```

Note: In IL, the function call has to be performed in one line.
The DATACMP function compares two data areas. As an option, wildcards can be used. DATACMP works independent of the data format at the inputs DATA and COMP.

### Parameters

<table>
<thead>
<tr>
<th>Input</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enable for block processing</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address of the received telegram to be compared (by ADR operator)</td>
</tr>
<tr>
<td>COMP</td>
<td>DWORD</td>
<td>Address of the comparison telegram (by ADR operator)</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>Number of bytes to be compared</td>
</tr>
<tr>
<td>DEF</td>
<td>INT</td>
<td>Wildcard (-1 no wildcard, otherwise 0 &lt;= DEF &lt;= 255)</td>
</tr>
</tbody>
</table>

### Description

The DATACMP function compares two data areas. As an option, wildcards can be used. DATACMP works independent of the data format at the inputs DATA and COMP.

#### EN

**BOOL**

**EN FALSE**

With EN = FALSE, the function is not performed.

**EN TRUE**

If EN = TRUE, the function compares with every call the data areas specified at DATA and COMP with a length given at LEN taking into consideration a wildcard if exists at DEF.

#### DATA

**DWORD**

The DATA input specifies the start address (given by an ADR operator) of the data to be compared. The data can be stored in the operand area or as variables. If binary values are used, some special things must be taken into account.

### COMP

**DWORD**

The COMP input specifies the start address (given by an address operator) of the comparison data. The data can be stored in the operand area or as variables. If binary values are used, some special things must be taken into account.

If IEC bit operators are used as data addresses, only such operands are permitted which end with an "0" (e.g., %QX62.0 is allowed, %QX62.1 is forbidden). If this condition is ignored, the block generates an error message.

While IEC bit operands are stored as packed bytes (8 operands per byte), self-defined BOOlean variables use one byte each. This byte can only be FALSE = 0 or TRUE = 1.
LEN INT
The LEN input specifies the number of bytes to be compared.

DEF INT
AT the DEF input, an optional wildcard character can be specified. Valid values: −1 and 0 <= DEF <= 255.

If DEF = −1, the data areas are compared directly (without wildcards). If 0 <= DEF <= 255, every character (byte) within the COMP area, that is equal to the DEF-defined character, is interpreted as a wildcard. During comparison between the data areas DATA and COMP, all characters of the DATA area that have a wildcard in the COMP area are ignored.

DATACMP INT
The return code of the DATACMP function indicates the result of comparison and can have the following values:

<table>
<thead>
<tr>
<th>Parameter / Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN = FALSE</td>
<td>DATA_NO_ACTION</td>
</tr>
<tr>
<td>LEN &lt; 0</td>
<td>DATA_INVALID_LENGTH</td>
</tr>
<tr>
<td>Address DATA or COMP invalid</td>
<td>DATA_INVALID_ADDRESS</td>
</tr>
<tr>
<td>DEF invalid</td>
<td>DATA_INVALID_WILDCARD</td>
</tr>
<tr>
<td>Data areas are equal</td>
<td>DATA_AREA_EQUAL</td>
</tr>
<tr>
<td>Data areas are unequal</td>
<td>DATA_AREA_UNEQUAL</td>
</tr>
</tbody>
</table>

Example
It is to be checked whether a telegram received equals the following pattern:

'Bispiel xx/x'.

The areas marked with 'x' are to contain digits in ASCII format (00/0 to 99/9). They are not to be evaluated. The received telegram is stored in the DATA_DATACMP string. The character § (ASCII 15 hex / 21 dec.) is defined as wildcard character.

For better understanding, the receiving telegram in the following example is fixed. At first, the variables are defined:

```
EN_DATACMP: BOOL;
DATA_DATACMP: STRING(13) := 'Beispiel 12/3';
COMP_DATACMP: STRING(13) := 'Beispiel $15$15/$15';
RET_DATACMP: INT;
```

This results in the following function call:
The following figure shows the working method of the DATACOMP function.

**Function call in IL**

1. LD DATA_DATACMP
2. ADR
3. ST _DWORD_0
4. LD COMP_DATACMP
5. ADR
6. ST _DWORD_1
7. LD EN_DATACMP
8. DATACMP _DWORD_0, _DWORD_1, LEN_DATACMP, DEF_DATACMP
9. ST RET_DATACMP

**Function call in ST**

1. RET_DATACMP = DATACMP
2. (EN := EN_DATACMP, DATA := ADR(DATA_DATACMP), COMP := ADR(COMP_DATACMP), LEN := LEN_DATACMP, DEF := DEF_DATACMP);

Note: In IL, the function call has to be performed in one line.
The DATAMOV function copies indexed data areas. Data areas which overlap each other are automatically considered. DATAMOV works independently of data formats and can therefore be used for putting together telegrams with different data formats.

### Parameters

<table>
<thead>
<tr>
<th>Input</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enable of block processing</td>
</tr>
<tr>
<td>SOURCE</td>
<td>DWORD</td>
<td>Start address of the data area to be copied (by ADR operator)</td>
</tr>
<tr>
<td>S_POS</td>
<td>INT</td>
<td>Offset of the data to be copied, within the area, beginning as of SOURCE in bytes</td>
</tr>
<tr>
<td>DEST</td>
<td>DWORD</td>
<td>Start address of the target data area (by ADR operator)</td>
</tr>
<tr>
<td>D_POS</td>
<td>INT</td>
<td>Offset of the target data, within the area, beginning as of DEST in bytes</td>
</tr>
</tbody>
</table>

**Return**  

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATACMP</td>
<td>Return code, result of the copy process</td>
</tr>
</tbody>
</table>

### Description

The DATAMOV function copies indexed data areas. Data areas which overlap each other are automatically considered. DATAMOV works independently of data formats and can therefore be used for putting together telegrams with different data formats.

**EN**  

EN **FALSE**  
If EN = FALSE, the function is not processed.

**EN**  

EN **TRUE**  
If EN = TRUE and the input values are correct, the function copies the data from the SOURCE area (byte offset = S_POS) to the DEST area (byte offset = D_POS) with a length of LEN bytes.

**SOURCE**  

At the SOURCE input, the start address of the data to be copied by an ADR operator is specified.

The data can be stored in the operand area or as variables. If binary values are used, some special things must be taken into account.

If IEC bit operators are used as data addresses, only such operands are permitted which end with an "0" (e.g. %QX62.0 is allowed, %QX62.1 is forbidden). If this condition is ignored, the block generates an error message.

**S_POS**  

At the S_POS input, the byte offset of the SOURCE data to be copied is specified.

**DEST**  

At the DEST input, the start address of the target area is specified by an address operator.

The data can be stored in the operand area or as variables. If binary values are used, some special things must be taken into account.

If IEC bit operators are used as data addresses, only such operands are permitted which end with an "0" (e.g. %QX62.0 is allowed, %QX62.1 is forbidden). If this condition is ignored, the block generates an error message.

While IEC bit operands are stored as packed bytes (8 operands per byte), self-defined BOOLEAN variables use one byte each. This byte can only be FALSE = 0 or TRUE = 1.
D_POS INT LEN INT
At the D_POS input, the byte offset is specified within the DEST target area, where the data is to be copied to.

DATAMOV INT
The return code of the DATAMOV function indicates errors occurred.

Parameter / Status Result
EN = FALSE DATAMOV = 0 DATA_NO_ACTION
LEN <= 0 DATAMOV = 1 DATA_INVALID_LENGTH
S_POS < 0 or D_POS < 0 DATAMOV = 2 DATA_INVALID_POSITION
Address SOURCE or DEST invalid DATAPOS = 3 DATA_INVALID_ADDRESS
No other error DATAMOV = 0 DATA_OK

Example
In the following example, telegrams are to be sent which have the following pattern:
‘Beispiel xx/x’.
In this example, the areas marked with ‘x’ are to contain the digits 12/3 in ASCII format. The telegram to be sent is stored in the DEST_DATAMOV string.
At first, the variables are defined here:

\[
\begin{align*}
\text{EN_DATAMOV} & : \text{BOOL}; \\
\text{DEST_DATAMOV} & : \text{STRING}(13) := \text{’Beispiel xx/x’}; \\
\text{Zahl1} & : \text{INT} := 12; \\
\text{Zahl2} & : \text{INT} := 3; \\
\text{Zahl1Str} & : \text{STRING}(2); \\
\text{Zahl2Str} & : \text{STRING}(1); \\
\text{RET_DATAMOV} & : \text{INT};
\end{align*}
\]

In the first step, the area ‘xx’ of the pre-initialized DEST_DATAMOV string has to be substituted with an ASCII character string containing the value 12.
The DEST_DATAMOV string now contains 'Beispiel 12/x'. In the next step, the 'x' has to be replaced by an ASCII character containing the digit 3.

As the result, the DEST_DATAMOV string now contains 'Beispiel 12/3' and can be sent as the telegram.

### Function call in IL

```plaintext
LD   SOURCE_DATAMOV
ADR                      
ST   _DWORD_0
LD   DEST_DATAMOV
ADR                      
ST   _DWORD_1
LD   EN_DATAMOV

DATAMOV   _DWORD_0, S_POS_DATAMOV,
           _DWORD_1, D_POS_DATAMOV, LEN_DATAMOV

ST   RET_DATAMOV
```

Note: In IL, the function call has to be performed in one line.

### Function call in ST

```plaintext
RET_DATAMOV =
DATAMOV (EN := EN_DATAMOV,
SOURCE := ADR(SOURCE_DATAMOV),
S_POS := S_POS_DATAMOV,
DEST := ADR(DEST_DATAMOV),
D_POS := D_POS_DATAMOV,
LEN := LEN_DATAMOV);
```
DETERMINING A WILDCARD COMPARISON VALUE IN A DATA TELEGRAM

The DATAPOS function determines and outputs wildcard areas within a data area. It works independent of data formats.

Block type

Function

Parameter

<table>
<thead>
<tr>
<th>Input</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Block enable</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address of the telegram to be compared (by ADR operator)</td>
</tr>
<tr>
<td>COMP</td>
<td>DWORD</td>
<td>Address of the comparison telegram (by ADR operator)</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>Valid length of the comparison in bytes</td>
</tr>
<tr>
<td>DEF</td>
<td>INT</td>
<td>Wildcard (0 &lt;= DEF &lt;= 255)</td>
</tr>
<tr>
<td>POS</td>
<td>INT</td>
<td>Consecutive number of the wildcard area searched for</td>
</tr>
<tr>
<td>VAL</td>
<td>DWORD</td>
<td>Address for character string matching the wildcard area (by ADR operator)</td>
</tr>
</tbody>
</table>

Return

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATAPOS</td>
<td>INT Return code, result of the searching process</td>
</tr>
</tbody>
</table>

Description

The DATAPOS function determines and outputs wildcard areas within a data area. It works independent of data formats.

When the function has found the wildcard area with the number POS, the corresponding data of the DATA area is output in the VAL area. The return code of the function is then the length of the area as of VAL.

DATA DWORD

The DATA input specifies the start address (given by an ADR operator) of the data to be compared. The data can be stored in the operand area or as variables. If binary values are used, some special things must be taken into account.

If IEC bit operators are used as data addresses, only such operands are permitted which end with an "0" (e.g. %QX62.0 is allowed, %QX62.1 is forbidden). If this condition is ignored, the block generates an error message.

While IEC bit operands are stored as packed bytes (8 operands per byte), self-defined BOOLEAN variables use one byte each. This byte can only be FALSE = 0 or TRUE = 1.
DETERMINING A WILDCARD COMPARISON VALUE IN A DATA TELEGRAM

COMP DWORD
The COMP input specifies the start address (given by an ADR operator) of the comparison data. The data can be stored in the operand area or as variables. If binary values are used, some special things must be taken into account.

If IEC bit operators are used as data addresses, only such operands are permitted which end with an "0" (e.g. %QX62.0 is allowed, %QX62.1 is forbidden). If this condition is ignored, the block generates an error message.

While IEC bit operands are stored as packed bytes (8 operands per byte), self-defined BOOLEAN variables use one byte each. This byte can only be FALSE = 0 or TRUE = 1.

LEN INT
At the LEN input, the maximum number of bytes has to be specified which are to be considered when searching for wildcards DEF within the COMP area.

DEF INT
At the DEF input, the value of the wildcard character is specified, that is searched for in COMP. For DEF, the values 0 <= DEF <= 255 are valid.

VAL DWORD
The VAL input specifies the start address of the data area where the data have to be stored in DATA corresponding to the wildcard area in COMP.

While IEC bit operands are stored as packed bytes (8 operands per byte), self-defined BOOLEAN variables use one byte each. This byte can only be FALSE = 0 or TRUE = 1.

Note: The VAL data area must be dimensioned big enough. It must have at least the size of the largest possible wildcard area in COMP.

DATapos INT
The return code of the DATapos function indicates the valid length of the VAL area (return code > 0) or that an error occurred (return code < 0).

<table>
<thead>
<tr>
<th>Parameter / Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN = FALSE</td>
<td>DATapos = 0 DATA_NO_ACTION</td>
</tr>
<tr>
<td>LEN &lt;= 0</td>
<td>DATapos = -1 DATA_INVALID_LENGTH</td>
</tr>
<tr>
<td>POS &lt;= 0</td>
<td>DATapos = -2 DATA_INVALID_POSITION</td>
</tr>
<tr>
<td>Address DATA, COMP or VAL invalid</td>
<td>DATapos = -3 DATA_INVALID_ADDRESS</td>
</tr>
<tr>
<td>Wildcard area number (POS) not found</td>
<td>DATapos = -4 DATA_AREA_NOT_FOUND</td>
</tr>
<tr>
<td>DEF invalid</td>
<td>DATapos = -5 DATA_INVALID_WILDCARD</td>
</tr>
<tr>
<td>No other errors</td>
<td>DATapos &gt; 0, length of the valid area as of VAL in bytes</td>
</tr>
</tbody>
</table>

Parameter / Status Result

EN = FALSE DATapos = 0 DATA_NO_ACTION
LEN <= 0 DATapos = -1 DATA_INVALID_LENGTH
POS <= 0 DATapos = -2 DATA_INVALID_POSITION
Address DATA, COMP or VAL invalid DATapos = -3 DATA_INVALID_ADDRESS
Wildcard area number (POS) not found DATapos = -4 DATA_AREA_NOT_FOUND
DEF invalid DATapos = -5 DATA_INVALID_WILDCARD
No other errors DATapos > 0, length of the valid area as of VAL in bytes
DETERMINING A WILDCARD COMPARISON VALUE
IN A DATA TELEGRAM

Example

From a received telegram which has the pattern

'Beispiel xx/x'

the digit values have to be determined which are contained in the areas marked with 'x'. The received telegram is
stored in the DATA_DATAPOS string. As the wildcard, the § character (ASCII 15 hex. / 21 dec.) is defined.

For better understanding, the receiving telegram is fixed in the following example program. Therefore, the variables
are defined at first.

EN_DATAPOS: BOOL;

DATA_DATAPOS: STRING(13):='Beispiel 12/3';

COMP_DATAPOS: STRING(13):='Beispiel $15$15/$15';

RET_DATAPOS: INT;

Zahl1Str: STRING(2);

Zahl2Str: STRING(1);

Zahl1: INT;

Zahl2: INT;

The resulting program:
The following figure shows the way of working of the DATAPOS function.

```
<table>
<thead>
<tr>
<th>Bi</th>
<th>e</th>
<th>i</th>
<th>s</th>
<th>p</th>
<th>i</th>
<th>e</th>
<th>l</th>
<th>1</th>
<th>2</th>
<th>/</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>101</td>
<td>105</td>
<td>115</td>
<td>112</td>
<td>105</td>
<td>101</td>
<td>108</td>
<td>49</td>
<td>50</td>
<td>47</td>
<td>51</td>
</tr>
</tbody>
</table>
```

**DATA_DATAPOS**

```
<table>
<thead>
<tr>
<th>Bi</th>
<th>e</th>
<th>i</th>
<th>s</th>
<th>p</th>
<th>i</th>
<th>e</th>
<th>l</th>
<th>§</th>
<th>§</th>
<th>/</th>
<th>§</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>101</td>
<td>105</td>
<td>115</td>
<td>112</td>
<td>105</td>
<td>101</td>
<td>108</td>
<td>21</td>
<td>21</td>
<td>47</td>
<td>21</td>
</tr>
</tbody>
</table>
```

**COMP_DATAPOS**

```
<table>
<thead>
<tr>
<th>§</th>
<th>STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>ASCII dec</td>
</tr>
<tr>
<td>15</td>
<td>ASCII hex</td>
</tr>
</tbody>
</table>
```

**DEF**

\[
\text{LEN} = \text{LEN(DATA_DATAPOS)} = 13
\]

POS = 1

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>50</td>
<td>ASCII dec</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>ASCII hex</td>
</tr>
</tbody>
</table>
```

\[
\text{Zahl1Str} = \text{STRING_TO_INT(Zahl1Str)} = \text{STRING_TO_INT('12')} = 12
\]

POS = 2

```
<table>
<thead>
<tr>
<th>3</th>
<th>STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>ASCII dec</td>
</tr>
<tr>
<td>33</td>
<td>ASCII hex</td>
</tr>
</tbody>
</table>
```

\[
\text{Zahl2Str} = \text{STRING_TO_INT(Zahl2Str)} = \text{STRING_TO_INT('3')} = 3
\]
DETERMINING A WILDCARD COMPARISON VALUE
IN A DATA TELEGRAM

Function call in IL

LD DATA_DATAPOS
ADR
ST _DWORD_0

LD COMP_DATAPOS
ADR
ST _DWORD_1

LD VAL_DATAPOS
ADR
ST _DWORD_2

LD EN_DATAPOS
LD DATAPOS _DWORD_0, _DWORD_1, LEN_DATAPOS, DEF_DATAPOS, POS_DATAPOS, _DWORD_2
ST RET_DATAPOS

Note: In IL, the function call has to be performed in one line.

Function call in ST

RET_DATAPOS = DATAPOS
(EN := EN_DATAPOS,
DATA := ADR(DATA_DATAPOS),
COMP := ADR(COMP_DATAPOS),
LEN := LEN_DATAPOS,
DEF := DEF_DATAPOS),
VAL := ADR(VAL_DATAPOS));
INITIALIZATION AND CONFIGURATION OF THE MODBUS INTERFACES

With every FALSE → TRUE edge at input EN, the MODINIT block reads the values at the inputs and initializes the interface (COM1, COM2, COM3, COM4) specified at input COM.

The MODBUS interface is active, until EN is set to TRUE. With a TRUE → FALSE edge at EN, the interface is enabled for other functions.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>MODINIT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>COM</td>
<td>INT</td>
<td>Interface identifier (COM1, COM2, COM3 or COM4)</td>
</tr>
<tr>
<td>BAUD</td>
<td>INT</td>
<td>Data transfer rate (300...19200 baud)</td>
</tr>
<tr>
<td>PTTY</td>
<td>INT</td>
<td>Parity (no parity bit, odd or even parity)</td>
</tr>
<tr>
<td>STOP</td>
<td>INT</td>
<td>Number of stop bits (1, 1½ or 2 stop bits)</td>
</tr>
<tr>
<td>RTSCTRL</td>
<td>BOOL</td>
<td>RTS controller (on/off)</td>
</tr>
<tr>
<td>TLS</td>
<td>INT</td>
<td>Carrier lead time (0...500 ms)</td>
</tr>
<tr>
<td>CDLY</td>
<td>INT</td>
<td>Carrier delay time (0...100 ms)</td>
</tr>
<tr>
<td>CHTO</td>
<td>INT</td>
<td>Character timeout (0...500 ms)</td>
</tr>
<tr>
<td>MAST_SLV</td>
<td>INT</td>
<td>MODBUS identifier (100 = master, 101...355 = slave (bus address + 100))</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error message</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error identifier</td>
</tr>
<tr>
<td>REC</td>
<td>BOOL</td>
<td>Telegram receiving message</td>
</tr>
<tr>
<td>SND</td>
<td>BOOL</td>
<td>Telegram sending message</td>
</tr>
<tr>
<td>FCT</td>
<td>INT</td>
<td>Function code of the sent / received telegram</td>
</tr>
</tbody>
</table>

Description

With every FALSE → TRUE edge at input EN, the MODINIT block reads the values at the inputs and initializes the MODBUS interface of the controller (COM1, COM2) or of the 07 KP 93 (COM3, COM4) specified at input COM.

The MODBUS interface is active, until EN is set to TRUE. With a TRUE → FALSE edge at EN, the interface is enabled for other functions.

The inputs can neither be duplicated nor negated/inverted.

Serial interfaces

The value processing at the inputs depends on the interface selected (PLC or 07 KP 93).

Usage of modems

The block MODINIT supports the special features of modems, interface converters and repeaters. If they are used at the MODBUS interfaces, the compression mechanism possibly supported by these devices must be deactivated. For detailed information, please refer to the manual of the respective device.

Some repeaters, modems or interface converters require a control signal to set the transfer direction.

For the interfaces COM1 and COM 2, the direction control can be activated or deactivated via input RTSCTRL.

Various devices additionally require a lead time to stabilize their carrier signal. These devices are not ready to transfer data in sending direction until this time has elapsed.
For the interfaces COM1 and COM 2, the carrier lead time can be specified via input TLS.

The 07 KP 93 interfaces COM3 and COM4 use no carrier lead time.

Additionally it is necessary for some devices to maintain the carrier signal in sending direction for some time after the data transfer. Only if this time has elapsed, the complete transfer of a telegram is ensured and the devices are ready for data transfer in opposite direction.

For the interfaces COM1 and COM 2, the carrier lead time can be specified via input TLS.

The 07 KP 93 interfaces COM3 and COM4 use no carrier delay time.

For using modems, a telegram can be possibly split up into several data blocks during the transfer. The character timeout defines the time which may pass between the receiving of two telegram data blocks without occurring an error message.

For the interfaces COM1 and COM 2, the character timeout can be specified via input CHTO.

The 07 KP 93 interfaces COM3 and COM4 use no character timeout.

Note: The carrier lead time and the carrier delay time must be coordinated at all devices at the bus.

**EN**

**BOOL**

**EN** $\text{FALSE} \rightarrow \text{TRUE}$:

If a $\text{FALSE} \rightarrow \text{TRUE}$ edge is applied to input EN, all further inputs are read in.

If the values at the inputs are valid, the interface which number is specified at input COM is initialized and ready for operation afterwards.

If there is at least one input invalid, the error is displayed at the outputs ERR and ERNO and the current interface setting is not changed.

**EN** $\text{TRUE}$:

The MODBUS interface is active, provided no errors occurred during the initialization (outputs ERR and ERNO). The telegram communication is displayed at the outputs REC, SND and FCT.

**EN** $\text{TRUE} \rightarrow \text{FALSE}$:

The MODBUS interface is deactivated and enabled for other applications (only COM1 and COM2). All outputs are set to zero (FALSE).

**EN** $\text{FALSE}$

The block is not processed. The interface is not used as MODBUS interface.

**COM**

**INT**

At input COM the MODBUS interface number is specified.

COM = 1: COM1 (SPS)

COM = 2: COM2 (SPS)

COM = 3: COM3 (07 KP 93)

COM = 4: COM4 (07 KP 93)

**BAUD**

**INT**

At input BAUD the value for the data transfer rate is specified.

The following settings are possible:

COM1 / COM2:

300, 1200, 2400, 4800, 9600, 14400, 19200

COM3 / COM4:

1200, 2400, 4800, 9600, 19200
<table>
<thead>
<tr>
<th>PTY</th>
<th>INT</th>
<th>TLS</th>
<th>INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>At input PTY is specified, with which parity a character is transferred.</td>
<td>The carrier lead time in milliseconds (0...500ms) is specified at input TLS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input PTY is only processed, if COM1 or COM2 is defined as MODBUS interface.</td>
<td>Input TLS is only processed, if COM1 or COM2 is defined as MODBUS interface and RTSCTRL = TRUE.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The 07 KP 93 interfaces COM3 and COM4 are permanently set to no parity bit.</td>
<td>The 07 KP 93 interfaces COM3 and COM4 use no carrier lead time.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PTY = 0:</strong> no parity bit</td>
<td><strong>Note:</strong> Keeping the carrier lead time depends on the cycle time of the task in which the MODINIT block is processed. The real time may deviate from the specification in worst case task cycle – 1ms.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PTY = 1:</strong> odd parity</td>
<td><strong>PTY = 2:</strong> even parity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The number of “1” values within a character is increased until an odd number is reached.</td>
<td>The number of “1” values within a character is increased until an even number is reached.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td>INT</td>
<td>CDLY</td>
<td>INT</td>
</tr>
<tr>
<td>The number of stop bits for each character is specified at input STOP.</td>
<td>The carrier delay time in milliseconds (0...100ms) is specified at input CDLY.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input STOP is only processed, if COM1 or COM2 is defined as MODBUS interface.</td>
<td>Input CDLY is only processed, if COM1 or COM2 is defined as MODBUS interface and RTSCTRL = TRUE.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The 07 KP 93 interfaces COM3 and COM4 are permanently set to one stop bit.</td>
<td>The 07 KP 93 interfaces COM3 and COM4 use no carrier delay time.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>STOP = 1:</strong> 1 stop bit</td>
<td><strong>Note:</strong> Keeping the carrier delay time depends on the cycle time of the task in which the MODINIT block is processed. The real time may deviate from the specification in worst case task cycle – 1ms.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>STOP = 2:</strong> 2 stop bits</td>
<td>CHTO</td>
<td>INT</td>
<td></td>
</tr>
<tr>
<td>RTSCTRL</td>
<td>BOOL</td>
<td>The character timeout in milliseconds (1...500ms) is specified at input CHTO.</td>
<td></td>
</tr>
<tr>
<td>Input RTSCTRL specifies whether the synchronization between MODBUS interface and connected device should be performed via the control signal RTS (no RTS/CTS handshake).</td>
<td>Input CHTO is only processed, if COM1 or COM2 is defined as MODBUS interface.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input RTSCTRL is only processed, if COM1 or COM2 is defined as MODBUS interface.</td>
<td>The 07 KP 93 interfaces COM3 and COM4 use no character timeout.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The 07 KP 93 interfaces COM3 and COM4 are permanently set to RTS/CTS handshake.</td>
<td><strong>Note:</strong> Keeping the character timeout depends on the cycle time of the task in which the MODINIT block is processed. The real time may deviate from the specification in worst case, task cycle – 1ms.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For activated RTS control (RTSCTRL = TRUE), the control signals are generated automatically by the interface.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**MAST_SLV**

The MODBUS subscriber identifier is specified at input MAST_SLV.

MAST_SLV = 100: MODBUS master

MAST_SLV = 101...355: MODBUS slave

In operating mode MODBUS slave, the bus address MAST_SLV – 100 is set.

**RDY**

Output RDY indicates whether the MODBUS interface is ready. RDY has always to be considered together with output ERR. The following applies:

RDY = FALSE and ERR = FALSE:

The interface initialization has not been completed successfully. It is not used as MODBUS interface.

An access contention to the 07 KP 93 interfaces COM3 / COM4 can possibly cause the initialization to be performed only in the next cycle.

RDY = FALSE and ERR = TRUE:

An error occurred at the interface. It is not used as MODBUS interface. Output ERNO displays the error identifier.

RDY = TRUE and ERR = FALSE:

The MODBUS interface has been initialized successfully and works error-free.

RDY = TRUE and ERR = TRUE:

A MODBUS error occurred. Output ERNO displays the error identifier. Interface continues to operate normally.

**ERR**

Output ERR displays, whether an error occurred at the interface. ERR has always to be considered together with output RDY.

If an error occurred (ERR = TRUE), the error type can be evaluated via the error number at output ERNO.

**ERNO**

At output ERNO an error identifier is output. ERNO has always to be considered together with RDY and ERR.

**REC**

Output REC displays whether a telegram has been received since the last block processing. If REC = TRUE, FCT displays the function code of the received telegram.

If an error occurred during the reception, it is displayed at the outputs ERR and ERNO.

**SND**

Output SND displays whether a telegram has been sent since the last block processing. If SND = TRUE, FCT displays the function code of the sent telegram.

If an error occurred during the reception, it is displayed at the outputs ERR and ERNO.

**FCT**

FCT displays the function code of the last telegram. FCT has always to be considered together with the outputs REC and SND. Output FCT is only valid, if REC = TRUE or SND = TRUE.
Function call in IL

```plaintext
CAL MODINIT_COM1
(EN := MOD1_EN,
COM := MOD1_COM,
BAUD := MOD1_BAUD,
PTY := MOD1PTY,
STOP := MOD1_STOP,
RTSCTRL := MOD1_RTSCTRL,
TLS := MOD1_TLS,
CDLY := MOD1_CDLY,
CHTO := MOD1_CHTO,
MAST_SLV := MOD1_MAST_SLV)
```

LD MODINIT_COM1.RDY
ST MOD1_RDY
LD MODINIT_COM1.ERR
ST MOD1_ERR
LD MODINIT_COM1.ERNO
ST MOD1_ERNO
LD MODINIT_COM1.REC
ST MOD1_REC
LD MODINIT_COM1.SND
ST MOD1_SND
LD MODINIT_COM1.FCT
ST MOD1_FCT

Note: In IL, the function call has to be performed in one line.

Function call in ST

```plaintext
MODINIT_COM1
(EN := MOD1_EN,
COM := MOD1_COM,
BAUD := MOD1_BAUD,
PTY := MOD1PTY,
STOP := MOD1_STOP,
RTSCTRL := MOD1_RTSCTRL,
TLS := MOD1_TLS,
CDLY := MOD1_CDLY,
CHTO := MOD1_CHTO,
MAST_SLV := MOD1_MAST_SLV);
```

MOD1_RDY := MODINIT_COM1.RDY;
MOD1_ERR := MODINIT_COM1.ERR;
MOD1_ERNO := MODINIT_COM1.ERNO;
MOD1_REC := MODINIT_COM1.REC;
MOD1_SND := MODINIT_COM1.SND;
MOD1_FCT := MODINIT_COM1.FCT;
OPERATION MODE MODBUS MASTER

The block MODMAST realizes the MODBUS master function for the MODBUS interface (COM1, COM2, COM3, COM4) specified at input COM.

For each interface, a separate MODMAST block must be used.

Before MODMAST can be used for an interface, this must be configured as a MODBUS master interface via the block MODINIT.

With each FALSE $\rightarrow$ TRUE edge at input EN, the function block MODMAST reads the values at the inputs, generates a telegram according to the inputs and sends this telegram to the slave.

Block type
Function block with historical values

Parameter

<table>
<thead>
<tr>
<th>Instance</th>
<th>MODMAST</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>COM</td>
<td>INT</td>
<td>Interface identifier (COM1, COM2, COM3 or COM4)</td>
</tr>
<tr>
<td>SLAVE</td>
<td>INT</td>
<td>Slave address (1...255)</td>
</tr>
<tr>
<td>FCT</td>
<td>INT</td>
<td>Function code</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>INT</td>
<td>Telegram timeout in milliseconds</td>
</tr>
<tr>
<td>ADDR</td>
<td>INT</td>
<td>Operand/register address in the slave</td>
</tr>
<tr>
<td>NB</td>
<td>INT</td>
<td>Number of data</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>First operand address of an operand area in the master, from which data are sent to the slave or data read by the slave should be stored.</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Ready message</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error message</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error identifier</td>
</tr>
</tbody>
</table>

Description

The block MODMAST realizes the MODBUS master function for the MODBUS interface of the controller (COM1, COM2) or the 07 KP 93 (COM3, COM4) specified at input COM.

For each interface, a separate MODMAST block must be used.

Before MODMAST can be used for an interface, this must be configured as a MODBUS master interface via the block MODINIT.

With each FALSE $\rightarrow$ TRUE edge at input EN, the function block MODMAST reads the values at the inputs, generates a telegram according to the inputs and sends this telegram to the slave.

Using several MODMAST for one interface, it has to be ensured that always only one MODMAST is enabled.

The inputs can neither be duplicated nor negated/inverted.
EN BOOL
If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.
If the input values are valid, a request telegram is sent to a slave.
If at least one input is invalid, no telegram is generated and the error is displayed at the outputs ERR and ERNO instead.

COM INT
At input COM the MODBUS interface number is specified.
COM = 1: COM1 (SPS)
COM = 2: COM2 (SPS)
COM = 3: COM3 (07 KP 93)
COM = 4: COM4 (07 KP 93)

SLAVE INT
At input SLAVE the address of the slave to which a telegram should be sent is specified. Valid values are 1...255.

FCT INT
The function code of the request telegram is specified at input FCT.
01 or 02 read n bits
03 or 04 read n words
05 write one bit
06 write one word
07 read M01,00...M01,07
15 write n bits
16 write n words

Note: Function 07 is only available for COM1 / COM2.

TIMEOUT INT
The telegram timeout in milliseconds (ms) is specified at input TIMEOUT.
If no response is received within the time interval specified in TIMEOUT, the procedure is aborted and an error identifier is output.
Note: Keeping the timeout depends on the cycle time of the task in which the MODMAST block is processed. The real time may deviate from the specification in worst case task cycle – 1ms.

ADDR INT
The operand/register address in the slave from which data should be read or written is specified at input ADDR.
The access to operands of AC31 devices in MODBUS slave mode is defined via the MODBUS cross-reference list. Only operands can be used which are listed in the cross-reference list.

NB INT
The number of data to be written or read is specified at input NB.
The unit of NB depends on the selected function. For bit accesses the number of bits, for word and double word accesses the number of words is specified at NB.
The following restrictions apply to the length:

<table>
<thead>
<tr>
<th>FCT</th>
<th>NBmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 or 02</td>
<td>96 bit</td>
</tr>
<tr>
<td>03 or 04</td>
<td>96 words / 48 double words</td>
</tr>
<tr>
<td>05</td>
<td>1 bit</td>
</tr>
<tr>
<td>06</td>
<td>1 word</td>
</tr>
<tr>
<td>07</td>
<td>8 bit</td>
</tr>
<tr>
<td>15</td>
<td>192 bit</td>
</tr>
<tr>
<td>16</td>
<td>96 words / 48 double words</td>
</tr>
</tbody>
</table>

DATA DWORD
The first operand address of an operand area in the master, from which data are written to the slave or in which the data read by the slave should be stored is specified at input DATA. Only operands can be used which are listed in the cross-reference list.
The ADR operator provides the address of an operand.
**RDY**  
Output **RDY** indicates whether the MODBUS is ready. Only with **RDY** = **TRUE**, the master is ready to send a new telegram.

**RDY** has always to be considered together with output **ERR**. The following applies:

**RDY** = **FALSE** and **ERR** = **FALSE**:
The processing of the last request has not been completed.

**RDY** = **FALSE** and **ERR** = **TRUE**:
An error occurred at the interface. Output **ERNO** displays the error identifier.

**RDY** = **TRUE** and **ERR** = **FALSE**:
The processing of the last request has been completed successfully.

**RDY** = **TRUE** and **ERR** = **TRUE**:
At least one input is invalid or a MODBUS error occurred. Output **ERNO** displays the error identifier. Interface continues to operate normally.

**ERR**  
Output **ERR** displays whether an error occurred. **ERR** has always to be considered together with output **RDY**.

If an error occurred (**ERR** = **TRUE**), the error type can be evaluated via the error number at output **ERNO**.

**ERNO**  
At output **ERNO** an error identifier is output. **ERNO** has always to be considered together with **RDY** and **ERR**.

<table>
<thead>
<tr>
<th>Function call in IL</th>
<th>Function call in ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>MODMAST_COM1</td>
</tr>
<tr>
<td>Operand</td>
<td>(EN := MODM1_EN,</td>
</tr>
<tr>
<td></td>
<td>COM := MODM1_COM,</td>
</tr>
<tr>
<td></td>
<td>SLAVE := MODM1_SLAVE,</td>
</tr>
<tr>
<td></td>
<td>FCT := MODM1_FCT,</td>
</tr>
<tr>
<td></td>
<td>TIMEOUT := MODM1_TIMEOUT,</td>
</tr>
<tr>
<td></td>
<td>ADDR := MODM1_ADDR,</td>
</tr>
<tr>
<td></td>
<td>NB := MODM1_NB,</td>
</tr>
<tr>
<td></td>
<td>DATA := MODM1_DATA)</td>
</tr>
<tr>
<td>ADR</td>
<td>MODMAST_COM1.RDY</td>
</tr>
<tr>
<td>ST</td>
<td>MODM1_RDY</td>
</tr>
<tr>
<td>ST</td>
<td>MODM1_ERR</td>
</tr>
<tr>
<td>LD</td>
<td>MODMAST_COM1.ERNO</td>
</tr>
<tr>
<td>ST</td>
<td>MODM1_ERNO</td>
</tr>
</tbody>
</table>

**Note:** In IL, the function call has to be performed in one line.
**Activation of the input EN depending on the output RDY**

The input EN (enabling) and the outputs RDY (ready), ERR (error) and ERNO (error number) have always to be considered together.

After a successful MODBUS master initialization via the block MODINIT, output RDY of the block MODMAST first has the value TRUE. The telegram communication is started via a $\text{FALSE} \rightarrow \text{TRUE}$ edge at MODMAST input EN.

After processing the request or after an error has occurred the master signalizes his readiness to perform a new job with RDY = TRUE. Jobs are only accepted, if the master is ready.

- **EN**: $\text{FALSE} \rightarrow \text{TRUE}$ edge starts job if master is ready ($\text{RDY} = \text{TRUE}$)

- **RDY**: Master ready for job, if RDY = TRUE

- **ERR**: Error, if ERR = TRUE; error identifier in ERNO

- **ERNO**: new job, master not ready, no error (init. state) | processing job, master ready, no error, new jobs are ignored | job done, master ready, error while processing job | processing new job, master not ready, no error, new jobs are ignored | job done, master ready, no error, job is not processed
Cross-reference list

**Caution:** An access to reserved (e.g. E065_00 = %IX065.00) or undefined operands (e.g. E64_08 = %IX064.08) is not permissible and leads to unforeseeable results. This applies to the operand area in the master (MODMAST input DATA) as well as to the operand area in the slave (MODMAST input ADDR).

Please note that forbidden operands can also be generated by a valid start address and a length NB (e.g. DATA = ADR(E064_00) and NB = 8 or ADDR = 400_{HEX} and NB = 8). Such combinations of addresses and lengths must be absolutely avoided.

**Caution:** Writing accesses to inputs must be avoided.

When writing via MODBUS to inputs of a 07 KT 97 which acts as a MODBUS slave, the written values are immediately overwritten by the "true" input values. When the 07 KT 97 acting as MODBUS master reads via MODBUS and the data are stored to inputs, the values which were read are also immediately overwritten again by the "true" input values.

**Caution:** When writing via MODBUS to constants of the 07 KT 97 which acts as a MODBUS slave, the written values are first only applied to the running operation. When the 07 KT 97 acting as MODBUS master reads via MODBUS and the data are stored to constants, the values which were read are also first only applied to the running operation.

In order to overwrite constants permanently, they must be saved additionally in the Flash memory. Otherwise the original values of the project are loaded after the supply voltage is switched on once again.

<table>
<thead>
<tr>
<th>Operands (symbolic)</th>
<th>Operands (IEC)</th>
<th>MODBUS address (HEX)</th>
<th>Operand description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E000_00</td>
<td>%IX000.00</td>
<td>0000_{HEX}</td>
<td>Binary inputs, CS31 bus</td>
</tr>
<tr>
<td>E061_15</td>
<td>%IX061.15</td>
<td>03DF_{HEX}</td>
<td></td>
</tr>
<tr>
<td>E062_00</td>
<td>%IX062.00</td>
<td>03E0_{HEX}</td>
<td>Binary inputs, local</td>
</tr>
<tr>
<td>E063_15</td>
<td>%IX063.15</td>
<td>03FF_{HEX}</td>
<td></td>
</tr>
<tr>
<td>E064_00</td>
<td>%IX064.00</td>
<td>0400_{HEX}</td>
<td>Binary inputs, local</td>
</tr>
<tr>
<td>E064_07</td>
<td>%IX064.07</td>
<td>0407_{HEX}</td>
<td>(bin. access to EW006_00..EW006_07)</td>
</tr>
<tr>
<td>E065_00</td>
<td>%IX065.00</td>
<td>0410_{HEX}</td>
<td>Binary inputs, central expansion (reserved)</td>
</tr>
<tr>
<td>E099_15</td>
<td>%IX099.15</td>
<td>063F_{HEX}</td>
<td></td>
</tr>
<tr>
<td>E100_00</td>
<td>%IX100.00</td>
<td>0640_{HEX}</td>
<td>Binary inputs, 2nd non-central expansion (reserved)</td>
</tr>
<tr>
<td>E163_15</td>
<td>%IX163.15</td>
<td>0A3F_{HEX}</td>
<td></td>
</tr>
<tr>
<td>E200_00</td>
<td>%IX200.00</td>
<td>0C80_{HEX}</td>
<td>Binary inputs, 3rd non-central expansion (reserved)</td>
</tr>
<tr>
<td>E255_15</td>
<td>%IX255.15</td>
<td>0FFF_{HEX}</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E256_00</td>
<td>%IX256.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E263_15</td>
<td>%IX263.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No direct access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binary inputs, 3rd non-central expansion (reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E263_15</td>
<td>%IX1.0000.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No direct access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binary inputs, PROFIBUS line 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E263_15</td>
<td>%IX1.1792.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A000_00</td>
<td>%QX000.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A061_15</td>
<td>%QX061.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A062_00</td>
<td>%QX062.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A063_15</td>
<td>%QX063.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A064_00</td>
<td>%QX064.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A099_15</td>
<td>%QX099.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A100_00</td>
<td>%QX100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A163_15</td>
<td>%QX163.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A200_00</td>
<td>%QX200.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A255_15</td>
<td>%QX255.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A256_00</td>
<td>%QX256.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A263_15</td>
<td>%QX263.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No direct access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binary outputs, 3rd non-central expansion (reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A263_15</td>
<td>%QX1.1792.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No direct access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binary outputs, PROFIBUS line 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A256_00</td>
<td>%QX1.1792.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No direct access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binary outputs, PROFIBUS line 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M000_00</td>
<td>%MX0000.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M254_15</td>
<td>%MX0254.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2000\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binary flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Start Value</td>
<td>End Value</td>
<td>Hex Value</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>M255_00</td>
<td>%MX0255.00</td>
<td></td>
<td>2FF0Hex</td>
</tr>
<tr>
<td>M255_15</td>
<td>%MX0255.15</td>
<td></td>
<td>2FFFHex</td>
</tr>
<tr>
<td>M256_00</td>
<td>%MX0256.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M256_15</td>
<td>%MX0256.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M279_00</td>
<td>%MX0279.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M279_15</td>
<td>%MX0279.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S000_00</td>
<td>%MX5000.00</td>
<td></td>
<td>3000Hex</td>
</tr>
<tr>
<td>S255_15</td>
<td>%MX5255.15</td>
<td></td>
<td>3FFFHex</td>
</tr>
<tr>
<td>EW000_00</td>
<td>%IW1000.00</td>
<td></td>
<td>0000Hex</td>
</tr>
<tr>
<td>EW005_15</td>
<td>%IW1005.15</td>
<td></td>
<td>005FHex</td>
</tr>
<tr>
<td>EW006_00</td>
<td>%IW1006.00</td>
<td></td>
<td>0060Hex</td>
</tr>
<tr>
<td>EW006_07</td>
<td>%IW1006.07</td>
<td></td>
<td>0067Hex</td>
</tr>
<tr>
<td>EW007_00</td>
<td>%IW1007.00</td>
<td></td>
<td>0070Hex</td>
</tr>
<tr>
<td>EW007_14</td>
<td>%IW1007.14</td>
<td></td>
<td>007EHex</td>
</tr>
<tr>
<td>EW007_15</td>
<td>%IW1007.15</td>
<td></td>
<td>007FHex</td>
</tr>
<tr>
<td>EW008_00</td>
<td>%IW1008.00</td>
<td></td>
<td>0080Hex</td>
</tr>
<tr>
<td>EW015_15</td>
<td>%IW1015.15</td>
<td></td>
<td>00FFHex</td>
</tr>
<tr>
<td>EW016_00</td>
<td>%IW1016.00</td>
<td></td>
<td>0100Hex</td>
</tr>
<tr>
<td>EW034_15</td>
<td>%IW1034.15</td>
<td></td>
<td>022FHex</td>
</tr>
<tr>
<td>EW100_00</td>
<td>%IW1100.00</td>
<td></td>
<td>0640Hex</td>
</tr>
<tr>
<td>EW107_15</td>
<td>%IW1107.15</td>
<td></td>
<td>06BFHex</td>
</tr>
<tr>
<td>EW200_00</td>
<td>%IW1200.00</td>
<td></td>
<td>0C80Hex</td>
</tr>
<tr>
<td>EW207_15</td>
<td>%IW1207.15</td>
<td></td>
<td>0CFFHex</td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%IW1.0000</td>
<td>No direct access Analog inputs, PROFIBUS line 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%IW1.1792</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%IW2.0000</td>
<td>No direct access Analog inputs, PROFIBUS line 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%IW2.1792</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW000_00</td>
<td>%QW1000.00 1000\text{HEX} Analog outputs, CS31 bus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW005_15</td>
<td>%QW1005.15 105F\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW006_00</td>
<td>%QW1006.00 1060\text{HEX} Analog outputs, local</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW006_03</td>
<td>%QW1006.03 1063\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW007_00</td>
<td>%QW1007.00 1070\text{HEX} Analog outputs (reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW007_15</td>
<td>%QW1007.15 107F\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW008_00</td>
<td>%QW1008.00 1080\text{HEX} Analog outputs, CS31 bus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW015_15</td>
<td>%QW1015.15 10FF\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW016_00</td>
<td>%QW1016.00 1100\text{HEX} Analog outputs, central expansion (reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW034_15</td>
<td>%QW1034.15 122F\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW100_00</td>
<td>%QW1100.00 1640\text{HEX} Analog outputs, 1st non-central expansion (reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW107_15</td>
<td>%QW1107.15 16BF\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW200_00</td>
<td>%QW1200.00 1C80\text{HEX} Analog outputs, 2nd non-central expansion (reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AW207_15</td>
<td>%QW1207.15 1CFF\text{HEX}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%QW1.0000</td>
<td>No direct access Analog outputs, PROFIBUS line 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%QW1.1792</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%QW2.0000</td>
<td>No direct access Analog outputs, PROFIBUS line 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%QW2.1792</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MW000_00</td>
<td>%MW1000.00 2000\text{HEX} Word flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MW253_15</td>
<td>%MW1253.15 2FDF\text{HEX}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MW254_00 %MW1254.00 2FE0\text{HEX}  \text{Word flags (error message)}
MW255_15 %MW1255.15 2FF\text{FHEX}

MW256_00 %MX1256.00 \textit{No direct access} \text{Word flags (system)}
MW259_15 %MX1259.15 \textit{(reserved)}

MW260_00 %MW1260.00 \textit{No direct access} \text{Word flags (user area)}
MW511_15 %MW1511.15

KW000_00 %MW3000.00 300\text{OHEX} \text{Word constants (system)}
KW000_15 %MW3000.15 300\text{FHEX}

KW001_00 %MW3001.00 3010\text{HEX} \text{Word constants}
KW079_15 %MW3079.15 34\text{FFHEX}

KW080_00 %MW3080.00 3500\text{HEX} \text{Word constants (system)}
KW089_15 %MW3089.15 359\text{FHEX}

MD000_00 %MD2000.00 4000\text{HEX} \text{Double word flags}
MD063_15 %MD2063.15 43\text{FFHEX}

KD000_00 %MD4000.00 5000\text{HEX} \text{Double word constant (system)}
KD000_01 %MD4000.01 5001\text{HEX} \text{Double word constants}
KD023_15 %MD4023.15 517\text{FHEX}
Error numbers at output ERNO of the blocks MODINIT and MODMAST

<table>
<thead>
<tr>
<th>ERNO</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error</td>
</tr>
</tbody>
</table>
| 1    | Function is not supported  
Master:  
Invalid value at input FCT of block MODMAST for corresponding COM  
or  
Slave responds with error telegram because it does not support the requested function  
Slave: Master requests function which the controller does not support |
| 2    | Invalid operand address  
Master:  
Slave responds with error telegram, invalid value for corresponding COM at input ADDR of block MODMAST  
Slave:  
Master attempts to access to an invalid operand address |
| 2    | Operand range exceeded  
Master:  
Slave responds with error telegram, combination of input ADDR and input NB of block MODMAST for corresponding COM exceed operand range limit in slave  
or  
Combination of input DATA and input NB of block MODMAST for corresponding COM exceed operand range limit in master  
Slave:  
Master attempts to access to data which exceed an operand range limit |
| 3    | Date is outside the permitted value range  
Master:  
Writing access to slave with invalid value, slave cannot process value |
| 10   | Length specifications in the telegram do not match  
Master/Slave:  
Transfer error or error while generating the telegram |
| 11   | Type of operand area and function code do not match  
Master:  
Invalid combination of input ADDR and input FCT of block MODMAST for corresponding COM  
or  
Invalid combination of input DATA and input FCT of block MODMAST for corresponding COM  
Slave:  
Master sends telegram which access type (bit / word) does not match with the operand range |
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Data length is greater than the permitted value</td>
</tr>
<tr>
<td></td>
<td>Master: Invalid value at input NB of block MODMAST for corresponding COM</td>
</tr>
<tr>
<td></td>
<td>Slave: Master request exceeds the max. data length which can be processed</td>
</tr>
<tr>
<td>13</td>
<td>Odd number of words during double word access</td>
</tr>
<tr>
<td></td>
<td>Master: Input ADDR of block MODMAST for corresponding COM has encoding for double word operand range and input NB has odd number of words or Input DATA of block MODMAST for corresponding COM has encoding for double word operand range and input NB has odd number of words</td>
</tr>
<tr>
<td></td>
<td>Slave: Master attempts to access to double word operand but length specification in telegram contains odd number of words</td>
</tr>
<tr>
<td>15</td>
<td>Initialization error</td>
</tr>
<tr>
<td></td>
<td>Master: Block MODMAST for corresponding COM attempts to access to interface although this has not been initialized by MODINIT</td>
</tr>
<tr>
<td>18</td>
<td>Checksum error</td>
</tr>
<tr>
<td></td>
<td>Master/Slave: Telegram contains wrong check sum, transfer error</td>
</tr>
<tr>
<td>20</td>
<td>Invalid interface</td>
</tr>
<tr>
<td></td>
<td>Master/Slave Invalid interface at input COM</td>
</tr>
<tr>
<td>50</td>
<td>Interface assigned by 907 AC 1131</td>
</tr>
<tr>
<td></td>
<td>Master/Slave Interface at input COM is used as programming interface and cannot be used as MODBUS interface</td>
</tr>
<tr>
<td>51</td>
<td>Baud rate not supported</td>
</tr>
<tr>
<td></td>
<td>Master/Slave Input BAUD has invalid value</td>
</tr>
<tr>
<td>52</td>
<td>Parity not supported</td>
</tr>
<tr>
<td></td>
<td>Master/Slave Input PTY has invalid value</td>
</tr>
<tr>
<td>53</td>
<td>Number of stop bits not supported</td>
</tr>
<tr>
<td></td>
<td>Master/Slave Input STOP has invalid value</td>
</tr>
<tr>
<td>54</td>
<td>Carrier lead not supported</td>
</tr>
<tr>
<td></td>
<td>Master/Slave Input TLS has invalid value</td>
</tr>
<tr>
<td>55</td>
<td>Carrier delay not supported</td>
</tr>
<tr>
<td></td>
<td>Master/Slave Input CDLY has invalid value</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 56 | Carrier delay longer than carrier lead  
Master/Slave:  
Input CDLY > input TLS                                                                                                                                 |
| 57 | Character timeout not supported  
Master/Slave:  
Input CHTO has invalid value                                                                                                                                 |
| 58 | Invalid MODBUS subscriber identifier  
Master/Slave:  
Input MAST_SLV has invalid value                                                                                                                                 |
| 59 | No bridge at COM  
Master/Slave:  
Interface has no bridge between pin 6 and pin 8, cable not plugged or pulled off                                                                 |
| 60 | Timeout during reception  
Master:  
Master did not receive a complete telegram within the specified time at input TIMEOUT of block MODMAST for corresponding COM  
Slave:  
Too long pause during the reception of a master telegram, telegram is interrupted                                                                        |
| 61 | Timeout during sending  
Master/Slave:  
Interface did not send the telegram within the specified time                                                                                                                                 |
| 62 | No MODBUS interface  
Master/Slave:  
Interface has been reconfigured by another block; operation as MODBUS interface is only possible after new initialization |
| 63 | Interface not configured as MODBUS master  
Master:  
MODMAST reads interface which has not been configured before as a MODBUS master using MODINIT                                                                 |
| 65 | 907 AC 1131 logged on  
Master/Slave:  
907 AC 1131 is logged on at input COM via the interface, interface cannot be used as MODBUS interface until 907 AC 1131 is logged off |
| 66 | MODBUS driver not ready  
Master:  
MODMAST reads interface although the processing of the existing request has not yet been completed                                                                 |
| 67 | Master/Slave:  
Parity, overflow or framing error on interface                                                                                                                                 |
| 68 | Master/Slave:  
Character timeout exceeded                                                                                                                                 |
| 69 | Master:  
Slave address at input SLAVE of block MODMAST is not within the range 1...255                                                                 |
| 101 | COM3 / COM4:  
No 07 KP 93                                                                                                                                 |
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>COM3 / COM4: Invalid request received</td>
</tr>
<tr>
<td>104</td>
<td>COM3 / COM4: Invalid operation mode</td>
</tr>
<tr>
<td>105</td>
<td>COM3 / COM 4: Life identifier failed</td>
</tr>
</tbody>
</table>
RECEIVING A DATA SET FROM THE COMMUNICATION MODULE 07 KP 92

This block reads data from the DP RAM of the communication module. In the communication module, these data can be supplied using the function *plc_send*.

**Block type**

Function block without historical values

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance</strong></td>
<td>SIREC</td>
</tr>
<tr>
<td><strong>EN</strong></td>
<td>BOOL</td>
</tr>
<tr>
<td><strong>COM</strong></td>
<td>INT</td>
</tr>
<tr>
<td><strong>MAX</strong></td>
<td>INT</td>
</tr>
<tr>
<td><strong>SMW</strong></td>
<td>INT</td>
</tr>
<tr>
<td><strong>LEN</strong></td>
<td>INT</td>
</tr>
<tr>
<td><strong>NEW</strong></td>
<td>BOOL</td>
</tr>
<tr>
<td><strong>ERR</strong></td>
<td>INT</td>
</tr>
<tr>
<td><strong>ERNO</strong></td>
<td>INT</td>
</tr>
</tbody>
</table>

**Description**

The block receives a data set from the communication module 07 KP 92 and stores it in the planned word flag area. The communication module has 2 serial interfaces (COM3 and COM4). A transmission range is available for each serial interface. The block reads the data set from the assigned area.

Input SMW specifies, where the data set read is stored. Input COM specifies to which communication module interface the data set is assigned.

**Note:**

The start variable must be either an addressable variable (%MW...,%IW...,%QW...) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

The enabled input (input EN = TRUE) checks, if a new data set is available in its transmission range. In this case, the data set is transferred to the planned word flag area and output NEW is set to »TRUE« for a cycle.

If no error occurred when reading the data, the outputs ERR and ERNO are additionally set to zero. The data set is stored beginning at the defined start word flag SMW. The number of received words is available at output LEN.

If an error occurs during the reading procedure, LEN is set to 0, NEW to TRUE and ERR to 1. The error type is indicated at output ERNO.

The outputs LEN, NEW, ERR and ERNO are set to zero by a zero signal at input EN. If the block is enabled (input EN = TRUE) and no new data set is available in the assigned transmission range, the outputs NEW, ERR and ERNO are set to zero.

**EN**

Block enabling

EN = FALSE → the block is not processed; the outputs LEN, NEW, ERR and ERNO are set to zero.

EN = TRUE → the block is processed.

**COM**

The number of the serial interface of the communication module which is assigned to the block is specified at input COM.

The following applies:

- 3 corresponds to COM 3
- 4 corresponds to COM 4
RECEIVING A DATA SET FROM THE COMMUNICATION MODULE 07 KP 92

**MAX**  
The maximum permissible number of the data set words is specified at input MAX. If the number of received words is higher than the planned number MAX, NEW is set to TRUE, ERR to 1 and ERNO to 4.

**SMW**  
At input SMW, the first word flag of the flag range in which the block stores the data set read, is specified.

**Note:**  
The start variable must be either an addressable variable (%MW..., %IW..., %QW...) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

**LEN**  
The number of words read is displayed at output LEN.

**NEW**  
The output NEW displays that the block outputs are valid. This output has always to be considered together with output ERR. The following applies:

\[
\text{NEW} = \text{TRUE and ERR} = 0:
\]

New data set received.

The data are available in the planned flag range and the number of data is displayed at output LEN.

\[
\text{NEW} = \text{TRUE and ERR} = 1:
\]

An error has occurred.

The error type is indicated at output ERNO.

The outputs LEN, NEW, ERR and ERNO are set to zero by a zero signal at input EN.

If the block is enabled (EN = TRUE) and no new data set is available in the assigned transmission range, the outputs NEW, ERR and ERNO are set to zero.

**ERR**  
Output ERR indicates whether an error occurred during the reading operation. This output has always to be considered together with output NEW. If an error occurred during the reading operation, NEW is set to TRUE and ERR is set to 1. The error type is indicated at output ERNO.

**ERNO**  
The output ERNO indicates an error number. This output has always to be considered together with the outputs NEW and ERR. The following applies:

\[
\text{ERNO} = 0: \quad \text{No error occurred}
\]

\[
\text{ERNO} = 1: \quad \text{Communication partner is no 07 KP 92 communication module. (Bit 0 of ERNO = 1)}
\]

\[
\text{ERNO} = 2: \quad \text{Initialization of 07 KP 92 is running. (Bit 1 of ERNO = 1)}
\]

\[
\text{ERNO} = 4: \quad \text{Number of received words is higher than the transmission range of the communication module. (Bit 2 of ERNO = 1)}
\]

\[
\text{ERNO} = 8: \quad \text{Number of received words is higher than the planned number MAX. (Bit 3 of ERNO = 1)}
\]

\[
\text{ERNO} = 16: \quad \text{Checksum error occurred (the data set is saved using a checksum). (Bit 4 of ERNO = 1)}
\]

The inputs and outputs can neither be duplicated nor inverted.
Function call in IL

CAL SIREC1(FREI := SIR_FREI,  
COM := SIR_COM,  
ANZ_MAX := SIR_ANZM,  
SMW := SIR_SMW)
LD SIREC1.NEW  
ST SIR_NEW  
LD SIREC1.ERR  
ST SIR_ERR  
LD SIREC1.ERNO  
ST SIR_ERNO  
LD SIREC1.LEN  
ST SIR_LEN

Function call in ST

SIREC1  
(FREI := SIR_FREI,  
COM := SIR_COM,  
ANZ_MAX := SIR_ANZM,  
SMW := SIR_SMW);  
SIR_NEW :=SIREC1.NEW;  
SIR_ERR :=SIREC1.ERR;  
SIR_ERNO :=SIREC1.ERNO;  
SIR_LEN :=SIREC1.LEN;

Note: In IL, the function call has to be performed in one line.
SENDING A DATA SET TO THE COMMUNICATION MODULE 07 KP 92

This block writes data to the DP RAM of the communication module. From the communication module, these data can be read using the function plc_rec.

Block type

Function block with historical values

Parameter

<table>
<thead>
<tr>
<th>Instance</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Transmission of a data set by a FALSE/TRUE edge</td>
</tr>
<tr>
<td>COM</td>
<td>INT</td>
<td>Number of the serial interface; 3 corresponds to COM 3, 4 corresponds to COM 4</td>
</tr>
<tr>
<td>LEN</td>
<td>INT</td>
<td>Number of data set words to be sent</td>
</tr>
<tr>
<td>SMW</td>
<td>INT</td>
<td>Starting word flag of the word range to be sent</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Transmission completed</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error has occurred</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description

The block sends a data set to the communication module 07 KP 92. The communication module has 2 serial interfaces (COM 3 and COM 4). A transmission range is available for each interface. The block writes the data set to the assigned area. The data can be further processed in the 07 KP 92.

The inputs LEN and SMW define which data are written to the transmission range. Input COM defines for which interface of the communication module these data are specified. When the transmission of a data set is started (FALSE/TRUE edge at input EN), the data at the inputs LEN and SMW must not be changed until the end of the transmission (RDY = TRUE).

Note:
The start variable must be either an addressable variable (%MW..., %IW..., %QW...) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

With a FALSE/TRUE edge at input EN the data set transmission is performed once. Until the transmission has not been finished (RDY = TRUE), input EN will not be evaluated again.

After the transmission is completed, all block outputs are updated. The transmission was successful, if RDY = TRUE and ERR = FALSE. For the outputs RDY = TRUE and ERR = TRUE, an error occurred. The error type is indicated at output ERNO. If the transmission range of the communication module is busy, RDY remains FALSE and with the next block call it is attempted again to transmit the data set. This procedure is repeated until the data set has been transmitted.

After transmitting the data set, a FALSE signal at input FREI sets all block outputs to zero (FALSE). A new FALSE/TRUE edge at input FREI starts a new transmission.

EN BOOL

Processing of the block is controlled via input FREI.

EN FALSE:

All outputs are set to the value FALSE (0). This is not valid during a transmission procedure.

EN FALSE/TRUE-Flanke:

Transmission of the data set is started once. Until the transmission has not been finished (RDY = TRUE), input EN will not be evaluated again.

EN TRUE:

The block is not processed, i.e. it no longer changes its outputs. This is not valid during a transmission procedure.
SENDING A DATA SET TO THE COMMUNICATION MODULE 07 KP 92

**COM**
The number of the serial interface of the communication module which is assigned to the block is specified at input COM.

The following applies:
- 3 corresponds to COM 3
- 4 corresponds to COM 4

**LEN**
The number of the data set words is specified at input LEN.

**SMW**
The first word flag of the data set is specified at input SMW.

Note:
The start variable must be either an addressable variable (%MW.., %IW.., %QW..) or an ARRAY. All other variables are freely defined by the system and therefore have no definite order.

**RDY**
Output RDY indicates that the data set transmission has been completed. This output has always to be considered together with output ERR.

The following applies:
RDY = TRUE and ERR = FALSE:
The transmission has been completed. The data set has been transmitted to the 07 KP 92.

RDY = TRUE and ERR = TRUE:
An error occurred during the transmission. Output ERNO signalizes the error number.

**ERR**
Output ERR indicates whether an error occurred during the transmission. This output has always to be considered together with output RDY. If an error has occurred, the following applies:
RDY = TRUE and ERR = TRUE:
Output ERNO signalizes the error number.

**ERNO**
The output ERNO indicates an error number. This output has always to be considered together with the outputs RDY and ERR.

The following applies:
ERNO = 0: No error occurred
ERNO = 1: Communication partner is no 07 KP 92 communication module.
(Bit 0 of ERNO = 1)
ERNO = 2: Initialization of 07 KP 92 is running.
(Bit 1 of ERNO = 1)
ERNO = 4: Number of words to be sent is higher than the transmission range of the communication module.
(Bit 2 of ERNO = 1)

The inputs and outputs can neither be duplicated nor inverted.

---

**Function call in IL**

CAL SISEND1(FREI := SIS_FREI, COM := SIS_COM, ANZ := SIS_ANZ, SMW := SIS_SMW)
LD SISEND1.ERR
ST SIS_ERR
LD SISEND1.ERNO
ST SIS_ERNO
LD SISEND1.RDY
ST SIS_RDY

Note: In IL, the function call has to be performed in one line.

---

**Function call in ST**

SISEND1(FREI := SIS_FREI, COM := SIS_COM, ANZ := SIS_ANZ, SMW := SIS_SMW);
SIS_ERR:=SISEND1.ERR;
SIS_ERNO:=SISEND1.ERNO;
SIS_RDY:=SISEND1.RDY;
**Glossary**

**BOOL**

Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

**DINT**

DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**DWORD**

DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**INT**

INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**WORD**

WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types..
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

Function blocks

Function blocks are subroutines which can have as much inputs, outputs and internal variables as required. They are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. This allows for example to realize counters which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters. A function has maximally one output parameter. Note that the output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

   For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

   For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

   The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
COM S90
  COMAUTOLOGIN 4
  COMINIT 6
  COMREC 18
  COMSND 12
  DATACMP 24
  DATAMOV 27
  DATAPOS 30
  MODINIT 35
  MODMAST 40
  SIREC 53
  SISEND 56
COMAUTOLOGIN S90 4
COMINIT S90 6
Components of the COM_S90 library 2
COMREC S90 18
COMSND S90 12

D
DATACMP S90 24
DATAMOV S90 27
DATAPOS S90 30

M
MODINIT S90 35
MODMAST S90 40

S
SIREC S90 53
SISEND S90 56
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

Function Blocks for Data Storage in SMC and FLASH 90 Series

SMC/FLASH 90 Series
Contents

The data storage library .......................... 2
PRECONDITIONS FOR THE USE OF THE DATA STORAGE LIBRARY ........................................... 2
COMPONENTS OF THE DATA STORAGE LIBRARY ................................................................. 2

Block overview arranged according to their call names .................................................. 2
DELETE DATA SEGMENT IN FLASH EPROM ................................................................. 3
READ DATA SET FROM FLASH EPROM ........................................................................ 5
WRITE DATA SET TO FLASH EPROM ........................................................................... 8
DELETE DATA SEGMENT IN SMARTMEDIA CARD ...................................................... 11
INITIALIZE SMARTMEDIA CARD ................................................................................ 14
READ DATA SET FROM SMARTMEDIA CARD ........................................................... 17
WRITE DATA SET TO SMARTMEDIA CARD .................................................................... 20

Glossary .......................................................................................................................... 23

Index ............................................................................................................................... 25
The data storage library

PRECONDITIONS FOR THE USE OF THE DATA STORAGE LIBRARY

Note: The blocks of the data storage library only run in the RUN mode of the PLC, but not in the simulation mode.

COMPONENTS OF THE DATA STORAGE LIBRARY

The following function blocks are contained in the data storage library:

Group: FCP
- FDEL - Delete data segment in Flash EPROM
- FRD - Read data set from Flash EPROM
- FWR - Write data set to Flash EPROM

Group: SMC
- FCDEL - Delete data segment in SmartMedia Card
- FCINIT - Initialize SmartMedia Card
- FCRD - Read data set from SmartMedia Card
- FCWR - Write data set to SmartMedia Card

Block overview arranged according to their call names

Explanation of abbreviations:

FBmV  … Function block with historical values
FBoV  … Function block without historical values
F    … Function

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCDEL</td>
<td>FBmV</td>
<td>Delete data segment in SmartMedia Card</td>
<td>11</td>
</tr>
<tr>
<td>FCINIT</td>
<td>FBmV</td>
<td>Initialize SmartMedia Card</td>
<td>14</td>
</tr>
<tr>
<td>FCRD</td>
<td>FBmV</td>
<td>Read data set from SmartMedia Card</td>
<td>17</td>
</tr>
<tr>
<td>FCWR</td>
<td>FBmV</td>
<td>Write data set to SmartMedia Card</td>
<td>20</td>
</tr>
<tr>
<td>FDEL</td>
<td>FBmV</td>
<td>Delete data set in Flash EPROM</td>
<td>3</td>
</tr>
<tr>
<td>FRD</td>
<td>FBmV</td>
<td>Read data set from Flash EPROM</td>
<td>5</td>
</tr>
<tr>
<td>FWR</td>
<td>FBmV</td>
<td>Write data set to Flash EPROM</td>
<td>8</td>
</tr>
</tbody>
</table>
DELETE DATA SEGMENT IN FLASH EPROM

This function block deletes a user data segment in the Flash EPROM. After deletion, all data are lost in this data segment.

CAUTION: runs only on 07 KT 98

Block data
Available as of PLC run-time system: V4.0
Contained in library: Datenablage_S90_V41.LIB

Block type
Function block with historical values

Parameters
Instance FDEL Instance name
FREI BOOL Deletion of the data segment with a FALSE/TRUE edge
SEG INT Number of the data segment; 0 or 1
RDY BOOL Deletion procedure completed
ERR BOOL An error occurred, data segment cannot be deleted
ERNO INT Error number

Description
This function block deletes a user data segment in the Flash EPROM. After deletion, all data are lost in this data segment.

Important note:
If the block is called from 07 KT 95/96/97 or 07 SL 97, an error is indicated: ERNO = -1.

An access to the Flash EPROM is only possible with the function blocks FWR and FRD.

The SEG input defines the data segment in the Flash EPROM. In the 07 KT 98 basic unit, two segments of 64 K bytes each are reserved for the user. These segments' numbers are 0 and 1. The deletion procedure of a data segment in the Flash EPROM can take several PLC cycles.

With a FATSE/TRUE edge at the FREI input the data segment deletion is performed once. Until the deletion procedure has not been finished (RDY = TRUE), the FREI input will not be evaluated again.

After completion of the deletion procedure all function block outputs are updated. The deletion process was successful, if RDY = TRUE and ERR = FALSE. If the outputs indicate RDY = TRUE and ERR = TRUE, the data segment could not be deleted.

FREI BOOL
The processing of the block is controlled with the FREI input.

FREI = FALSE:
All outputs are set to the FALSE status. This is not valid during a deletion procedure.

FREI = FALSE/TRUE edge:
Deletion of the data segment is started once. Until the deletion procedure has not been finished (RDY = TRUE), the FREI input will not be evaluated again.

FREI = TRUE:
The block is not processed, i.e. it no longer changes its outputs. This is not valid during a deletion procedure.

SEG INT
The number of the data segment in the Flash EPROM is specified at the SEG input. In the 07 KT 98 basic unit, two data segments are available for the user.
Valid values: 0 or 1
DELETE DATA SEGMENT IN FLASH EPROM  FDEL

RDY  BOOL
The RDY output indicates that the data segment deletion has been completed. This output has always to be considered together with the ERR output.
The following applies:
RDY = TRUE and ERR = FALSE:
The deletion procedure has been completed. The data segment has been deleted successfully.
RDY = TRUE and ERR = TRUE:
An error has occurred during the deletion procedure. The data segment could not be deleted.

ERR  BOOL
The ERR output indicates whether an error occurred during the deletion operation. This output has always to be considered together with the RDY output. If the data segment could not be deleted, the following is valid: RDY = TRUE and ERR = TRUE. The ERNO output indicates the error number.

ERNO  INT
The ERNO output indicates an error number. This output has always to be considered together with the outputs RDY and ERR. The error number is encoded binary. The error numbers are valid for all of the Flash blocks. The following applies:
ERNO = 0: No error occurred.
ERNO = 1 (bit 0 = 1): Block number and number of blocks is greater than 1926.

The operating system executes the functions FDEL, FWR and FRD in the background. This procedures can take quite a long time because the PLC user program is performed with priority. At the ERNO output then is displayed in which processing phase the block is.
ERNO = 301hex: – The block has detected the edge and is waiting for enabling by the operating system.
ERNO = 302hex: – The operating system has accepted the task.
ERNO = 303hex: – The task is being performed.
During this phase the outputs ERR and RDY are set to FALSE:
The inputs and outputs can neither be duplicated nor inverted.

Function call in IL
CAL FDEL1( FREI := FDEL_FREI, SEG := FDEL_SEG)
LD  FDEL1.RDY
ST  FDEL_RDY
LD  FDEL1.ERR
ST  FDEL_ERR
LD  FDEL1.ERNO
ST  FDEL_ERNO

Note: In IL, the function call has to be performed in one line.

Function call in ST
FDEL1(FREI := FDEL_FREI, SEG := FDEL_SEG);
FDEL_RDY:=FDEL1.RDY;
FDEL_ERR:=FDEL1.ERR;
FDEL_ERNO:=FDEL1.ERNO;
READ DATA SET FROM FLASH EPROM

The function block reads a data set from a data segment of the Flash EPROM and stores the read data set beginning at the start flag defined by SM.

Attention: block only runs on 07 KT 98

Important note:

If the block is called from 07 KT 95/96/97 or 07 SL 97, an error is indicated (ERNO = -1).

An access to the Flash EPROM is only permitted by using the function blocks FWR and FRD.

 NB blocks are copied from block BNR in segment SEG to the memory start address SM.

32 binary data or 16 word data or 8 double word data are read per block.

One block contains the following 34 bytes:
32 bytes of data
  1 byte for CRC checksum
  1 byte for the identifier ‘written’

(see also the table at the end of this block description)
**READ DATA SET FROM FLASH EPROM**

**FREI**  
**BOOL**  
Processing of the block is controlled by the status at the FREI input.  
The following applies:  
FREI = FALSE:  
The outputs RDY, ERR and ERNO are set to "0" resp. "FALSE".  
FREI = FALSE/TRUE edge:  
The reading procedure of the data set is carried out once.  
FREI = TRUE:  
The block is not processed, i.e. the block no longer changes its outputs.  

**NB**  
**INT**  
The number of data set blocks is specified at the NB input. 32 binary data or 16 word data or 8 double word data are read per block.  
Valid values: 1 ... 1927  
Example:  
- SM = MW 02,00 / %MW1002.0 and NB = 1: storage of the data from MW 02,00 / %MW1002.0 to MW 02,15 / %MW102.15  
  (1 block = 16 word data)  
- SM = MW 02,00 / %MW1002.0 and NB = 2: storage of the data from MW 02,00 / %MW1002.0 to MW 03,15 / %MW1003.15  
  (2 blocks = 32 word data)  

**SEG**  
**INT**  
The number of the data segment in the Flash EPROM is specified at the SEG input. There are two data segments available in the 07 KT 98 basic unit.  
Valid values: 0 or 1  

**BNR**  
**INT**  
The block number in the data segment is specified at the BNR input.  
Valid values: 0 ... 1926  

**SM**  
**DWORD**  
By means of an ADR operator, the address of the first variable for the storage of the data set is specified at the SM input.  

**RDY**  
**BOOL**  
The RDY output indicates that reading the data set has been completed. This output has always to be considered together with the ERR output.  
The following applies:  
RDY = TRUE and ERR = FALSE: The reading operation has been completed. The data set is stored beginning at the defined input SM.  
RDY = TRUE and ERR = TRUE: An error occurred during the reading procedure. The ERNO output indicates the error number.  

**ERR**  
**BOOL**  
The ERR output indicates whether an error occurred during the reading operation. This output has always to be considered together with the RDY output. If an error has occurred, the following applies:  
RDY = TRUE and ERR = TRUE. The ERNO output indicates the error code.  

**ERNO**  
**INT**  
The ERNO output indicates an error code. This output has always to be considered together with the outputs RDY and ERR.  
The error code is encoded binary. The following applies:  
ERNO = 0:  
No error occurred.  
ERNO = 1 (bit 0 = 1):  
Block number and number of blocks is greater than 1926.  
ERNO = 2 (bit 1 = 1):  
Data segment is greater than 1.  
ERNO = 4 (bit 2 = 1):  
Block not programmable or checksum error.  
ERNO = 8 (bit 3 = 1):  
Block already contains data.  
ERNO = 16 (bit 4 = 1):  
Block cannot be deleted.  
ERNO = -1 (bits 0...15 = 1):  
Block cannot be run on this PLC.  
The inputs and outputs can neither be duplicated nor inverted.
The following table shows the structure of a Flash segment.

<table>
<thead>
<tr>
<th>Byte offset</th>
<th>Block No.</th>
<th>Word 1</th>
<th>Word 2</th>
<th>Word 3</th>
<th>...</th>
<th>Word 15</th>
<th>Word 16</th>
<th>CRC</th>
<th>'Written identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65450</td>
<td>1925</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65484</td>
<td>1926</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Function call in IL

CAL FRD1(FREI := FRD_FREI,
NB := FRD_NB, SEG := FRD_SEG,
BMR := FRD_BMR, SM := FRD_SM)

LD FRD1.RDY
ST FRD_RDY
LD FRD1.ERR
ST FRD_ERR
LD FRD1.ERNO
ST FRD_ERRNO

Function call in ST

FRD1(FREI := FRD_FREI,
NB := FRD_NB, SEG := FRD_SEG,
BMR := FRD_BMR, SM := FRD_SM);
FRD_RDY := FRD1.RDY;
FRD_ERR := FRD1.ERR;
FRD_ERNO := FRD1.ERNO;

Note: In IL, the function call has to be performed in one line.
WRITE DATA SET TO FLASH EPROM

The function block writes a data set into a data segment of the Flash EPROM.

Attention: runs only on 07 KT 98

Block data

Available as of PLC run-time system: V4.0
Contained in library: Datenablage_S90_V41.LIB
Remark: Only for 07 KT 98

Block type

Function block with historical values

Parameters

Instance FWR Instance name
FREI BOOL Saving a data set with a FALSE/TRUE edge
NB INT Number of blocks of the data set; 1 ... 1927
SEG INT Number of the data segment; 0 or 1
BNR INT Number of the block in the data segment; 0 ... 1926
SM DWORD Address of the start flag of the data set
RDY BOOL Writing procedure completed
ERR BOOL An error occurred
ERNO INT Error number (error code)

Description

The function block writes a data set into a data segment in the Flash EPROM. For this purpose, two data segments are available in the 07 KT 98 basic unit. A deletion procedure (function block FDEL) always deletes an entire data segment. A data segment consists of 1927 blocks (0...1926). Each block consists of 34 bytes.

After deletion, each of these 1927 blocks of a data segment can be written only once. If a block, which already contains data, has to be written with other data, the entire data segment has to be deleted before. All data is lost in this data segment then.

Important note:

If the block is called from 07 KT 95/96/97 or 07 SL 97, an error is indicated (ERNO = -1).

An access to the Flash EPROM is only permitted by using the function blocks FDEL, FWR and FRD.

NB blocks are copied from the SM address to the BNR block in segment SEG.

32 binary data or 16 word data or 8 double word data are read per block.

One block contains the following 34 bytes:
32 bytes of data
1 byte for CRC checksum
1 byte for the identifier ‘written’
(see also the table at the end of this block description)

After a writing procedure of a data set has been started (FALSE/TRUE edge at the FREI input), the data of the data set must not be altered until the procedure has been finished (RDY = TRUE). Storing the data set in the Flash EPROM can take several PLC cycles.

A FALSE/TRUE edge at the FREI input triggers the writing procedure of the data set once. As long as the procedure is being performed, the FREI input is not evaluated.

After the writing procedure has completed, the block outputs RDY, ERR and ERNO are updated.
If RDY = TRUE and ERR = FALSE, the procedure was successful. If RDY = TRUE and ERR = TRUE, an error has occurred. The ERNO output indicates the error code (error number).
After storing a data set in the Flash EPROM, the block outputs RDY, ERR and ERNO are set to zero by a FALSE signal at the FREI input. A new FALSE/TRUE edge starts a new writing procedure.

Since no new data can be written to data blocks which already contain data without deleting the entire data segment, the BNR input must point to the next free block before the next writing procedure can be performed.

**FREI**  
Processing of the block is controlled by the status at the FREI input.

**FREI = FALSE:**  
The outputs RDY, ERR and ERNO are set to "0" resp. "FALSE". This is not valid when a writing procedure is being performed.

**FREI = FALSE/TRUE edge:**  
A FALSE/TRUE edge at the FREI input triggers the writing procedure of the data set once. As long as the procedure is being performed, the FREI input is not evaluated.

**FREI = TRUE:**  
The block is not processed, i.e. the block no longer changes its outputs. This is not valid when a writing procedure is being performed.

**NB**  
The number of data set blocks is specified at the NB input. 32 binary data or 16 word data or 8 double word data are stored per block.

Valid values: 0..1927

Example:

**SM = MW 02,00 / %MW1002.0 and NB = 1:** writing the data from MW 02,00 / %MW1002.0 to MW 01,15 / %MW1001.15  
(1 block = 16 word data)

**SM = MW 02,00 / %MW1002.0 and NB = 2:** writing the data from MW 02,00 / %MW1002.0 to MW 03,15 / %MW1003.15  
(2 blocks = 32 word data)

**SEG**  
The number of the data segment in the Flash EPROM is specified at the SEG input. There are two data segments available in the 07 KT 98 basic units.

Valid values: 0 and 1

**BNR**  
The block number in the data segment is specified at the BNR input.

Valid values: 0 ... 1926

**SM DWORD**  
By means of an ADR operator, the address of the first variable for the storage of the data set is specified at the SM input. After a writing procedure of a data set has been started (FALSE/TRUE edge at the FREI input), the data of the data set must not be altered until the procedure has been finished (RDY = TRUE).

**RDY**  
The RDY output indicates that the writing procedure of the data set has been completed. This output has always to be considered together with the ERR output.

The following applies:

**RDY = TRUE and ERR = FALSE:**  
The writing procedure has been completed. The data set was stored in the Flash EPROM.

**RDY = TRUE and ERR = TRUE:**  
An error occurred during the writing procedure. The ERNO output indicates the error code.

**ERR**  
The ERR output indicates whether an error occurred during the writing operation. This output has always to be considered together with the RDY output. If an error has occurred, the following applies:  
**RDY = TRUE and ERR = TRUE:**  
The ERNO output indicates the error code.
WRITE DATA SET TO FLASH EPROM

**ERNO**

The ERNO output indicates an error code. This output has always to be considered together with the outputs RDY and ERR.

The error code is encoded binary. The following applies:

**ERNO = 0:**

No error occurred.

**ERNO = 1 (bit 0 = 1):**

Block number and number of blocks is greater than 1926.

**ERNO = 2 (bit 1 = 1):**

Data segment is greater than 1.

**ERNO = 4 (bit 2 = 1):**

Block not programmable or checksum error.

**ERNO = 8 (bit 3 = 1):**

Block already contains data.

**ERNO = 16 (bit 4 = 1):**

Block cannot be deleted.

**ERNO = -1 bits 0...15 = 1):**

Block cannot be run on this PLC.

The inputs and outputs can neither be duplicated nor inverted.

The following table shows the structure of a Flash segment.

<table>
<thead>
<tr>
<th>Byte offset</th>
<th>Block No.</th>
<th>Word 1</th>
<th>Word 2</th>
<th>Word 3</th>
<th>...</th>
<th>Word 15</th>
<th>Word 16</th>
<th>CRC</th>
<th>‘Written’ identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65450</td>
<td>1925</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65484</td>
<td>1926</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Function call in IL**

```
CAL FWR1(FREI := FWR_FREI,
         NB := FWR_NB, SEG := FWR_SEG,
         BNR := FWR_BNR, SM := FWR_SM)
LD   FWR1.RDY
ST   FWR1.RDY
LD   FWR1.ERR
ST   FWR1.ERR
LD   FWR1.ERNO
ST   FWR1.ERNO
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
FWR1(FREI := FWR_FREI,
    NB := FWR_NB, SEG := FWR_SEG,
    BNR := FWR_BNR, SM := FWR_SM);
FWR_RDY:=FWR1.RDY;
FWR_ERR:=FWR1.ERR;
FWR_ERNO:=FWR1.ERNO;
```
DELETE DATA SEGMENT IN SMARTMEDIA CARD

This function block deletes a data segment in the SmartMedia Card. All data in this data segment are lost after deletion.

Block data
Available as of PLC run-time system:
V4.0 for 07 MC 90 R0101 (2 MB)
V5.0 for 07 MC 90 R0201 (8 MB)
and 07 KT 9x R02xx

Remark:
Contained in library:
ABB-BIB4.LIB
Datenablage_S90_V41.LIB

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instanz</th>
<th>FCDEL</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Deletion of a data segment by a FALSE/TRUE edge</td>
</tr>
<tr>
<td>SEG</td>
<td>INT</td>
<td>Number of the data segment</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Deletion procedure completed</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error occurred, data segment cannot be deleted</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description
This function block deletes a data segment in the SmartMedia Card. All data in this data segment are lost after deletion.

Important note:
An access to the SmartMedia Card is only permitted by using the function blocks FCWR and FCRD.

The input SEG defines the data segment in the SmartMedia Card. The deletion of a data segment in the SmartMedia Card can take several PLC cycles.

With a FALSE/TRUE edge at input FREI the data segment deletion is performed once. Until the deletion procedure has not been finished (RDY = TRUE), input FREI will not be evaluated again.

After completion of the deletion procedure all function block outputs are updated. The deletion operation was successful, if RDY = TRUE and ERR = FALSE.

If the outputs show RDY = TRUE and ERR = TRUE, the data segment could not be deleted.

FREI

Processing of the block is controlled via input FREI.

FREI = FALSE:
All outputs are set to the value "FALSE". This is not valid during a deletion procedure.

FREI = FALSE/TRUE edge:
Deletion of the data segment is started once. Until the deletion procedure has not been finished (RDY = TRUE), input FREI will not be evaluated again.

FREI = TRUE:
The block is not processed, i.e., it no longer changes its outputs. This is not valid during a deletion procedure.
DELETE DATA SEGMENT IN SMARTMEDIA CARD

SEG INT
The number of the data segment in the SmartMedia Card is specified at input SEG. The following applies:
07 MC 90 R0101 (2 MB): 1...250
07 MC 90 R0201 (8 MB): 1...1023
(as of run-time system V5.0 and 07 KT 9x R02xx)

RDY BOOL
Output RDY indicates that the data segment deletion has been completed. This output has always to be considered together with output ERR.
The following applies:
RDY = TRUE and ERR = FALSE:
The deletion procedure has been completed. The data segment has been deleted successfully.
RDY = TRUE and ERR = TRUE:
An error occurred during the deletion procedure. The data segment could not be deleted.

ERR BOOL
Output ERR indicates whether an error occurred during the deletion operation. This output has always to be considered together with output RDY. If the data segment could not be deleted, the following is valid: RDY = TRUE and ERR = TRUE. Output ERNO signalizes the error number.

ERNO INT
At output ERNO an error number is output. This output has always to be considered together with the outputs RDY and ERR. The error number is binary encoded and valid for all FC* blocks.
The following applies:
ERNO = 0 (bit 0...15 = 0):
No error occurred
ERNO = 1 (bit 0 = 1):
Block number and number of blocks is greater than 127.
ERNO = 2 (bit 1 = 1):
Data segment is greater than maximum possible.
ERNO = 4 (bit 2 = 1):
Block not programmable or check sum error.
ERNO = 8 (bit 3 = 1):
Block already contains data
ERNO = 16 (bit 4 = 1):
Segment not deletable or too many defective blocks.
ERNO = 32 (bit 5 = 1):
SmartMedia Card not initialized.
ERNO = 64 (bit 6 = 1):
SmartMedia Card not inserted or wrong card.
The operating system executes the functions FCDEL, FCINIT, FCWR and FCRD in the background. This procedures can take quite a long time (FCINIT > 10s) because the PLC user program is performed with priority. At output ERNO then it is displayed, in which processing phase the block is.
ERNO = 301hex
– the block has detected the edge and waits for enabling by the operating system;
ERNO = 302hex
– the operating system has assumed the task;
ERNO = 303hex
– the task is being performed.
During this phase the outputs ERR and RDY are set to FALSE.
The inputs and outputs can neither be duplicated nor inverted.
DELETE DATA SEGMENT IN SMARTMEDIA CARD

**Function call in IL**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>FCDEL1(FREI := FCDEL_FREI, SEG := FCDEL_SEG)</td>
</tr>
<tr>
<td>LD</td>
<td>FCDEL1.RDY</td>
</tr>
<tr>
<td>ST</td>
<td>FCDEL_RDY</td>
</tr>
<tr>
<td>LD</td>
<td>FCDEL1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>FCDEL_ERR</td>
</tr>
<tr>
<td>LD</td>
<td>FCDEL1.ERNO</td>
</tr>
<tr>
<td>ST</td>
<td>FCDEL_ERNO</td>
</tr>
</tbody>
</table>

**Function call in ST**

FCDEL1(FREI := FCDEL_FREI, SEG := FCDEL_SEG);
FCDEL_RDY:=FCDEL1.RDY;
FCDEL_ERR:=FCDEL1.ERR;
FCDEL_ERNO:=FCDEL1.ERNO;

Note: In IL, the function call has to be performed in one line.
The FCINIT block formats the SMC card similar to the formatting of a PC floppy disk using the command FORMAT. Then the card is deleted completely and prepared as a data card. The block does not check, if the card is already formatted and possibly contains data. An operating system card is also deleted.

**Block data**

<table>
<thead>
<tr>
<th>Available as of PLC run-time system:</th>
<th>V4.0 for 07 MC 90 R0101 (2 MB)</th>
<th>V5.0 for 07 MC 90 R0201 (8 MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and 07 KT 9x R02xx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contained in library:</td>
<td>ABB-BIB4.LIB</td>
<td>Datenablage_S90_V41.LIB</td>
</tr>
</tbody>
</table>

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instanz</th>
<th>FCINIT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Initializing the Flash Card by FALSE/TRUE edge</td>
</tr>
<tr>
<td>AUTO</td>
<td>INT</td>
<td>Without effect, reserved</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Initialization process completed</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error occurred, Flash Card can not be initialized</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

**Description**

The FCINIT block formats the SMC card similar to the formatting of a PC floppy disk using the command FORMAT. Then the card is deleted completely and prepared as a data card. The block does not check, if the card is already formatted and possibly contains data. An operating system card is also deleted.

The FCINIT block must be called once in order to use the card as a data card using the blocks FCWR, FCRD and FCDEL. After formatting, the following space is available on the card:

**07 MC 90 R0101 (2 MB):**
250 segments with 128 blocks of 64 bytes each, i.e. 2000 kB can be stored in total.

**07 MC 90 R0201 (8 MB):**
1023 segments with 128 blocks of 64 bytes each, i.e. 8184 kB can be stored in total (as of run-time system V5.0 and 07 KT 9x R02xx).

**FREI**

Processing of the block is controlled via input FREI.

**FREI = FALSE:**
All outputs are set to the value "FALSE". This is not valid during a deletion procedure.

**FREI = FALSE/TRUE edge:**
Deletion of the data segment is started once. Until the deletion procedure has not been finished (RDY = TRUE), input FREI will not be evaluated again.

**FREI = TRUE:**
The block is not processed, i.e. it no longer changes its outputs. This is not valid during a deletion procedure.
**INITIALIZE SMARTMEDIA CARD**

**FCINIT**

**AUTO**

Without effect, reserved

**RDY**

Output RDY indicates that the data segment deletion has been completed. This output has always to be considered together with output ERR.

The following applies:

RDY = TRUE and ERR = FALSE:
The deletion procedure has been completed. The data segment has been deleted successfully.

RDY = TRUE und ERR = TRUE:
An error occurred during the deletion procedure. The data segment could not be deleted.

**ERR**

Output ERR indicates whether an error occurred during the deletion operation. This output has always to be considered together with output RDY. If the data segment could not be deleted, the following is valid: RDY = TRUE and ERR = TRUE. Output ERNO signalizes the error number.

**ERNO**

At output ERNO an error number is output. This output has always to be considered together with the outputs RDY and ERR. The error number is binary encoded and valid for all FC* blocks.

The following applies:

ERNO = 0 (bits 0...15 =0):
No error occurred

ERNO = 1 (bit 0 = 1):
Block number and number of blocks is greater than 127.

ERNO = 2 (bit 1 = 1):
Data segment is greater than maximum possible.

ERNO = 4 (bit 2 = 1):
Block not programmable or check sum error.

ERNO = 8 (bit 3 = 1):
Block already contains data.

ERNO = 16 (bit 4 = 1):
Sector not deletable or too many defective blocks.

ERNO = 32 (bit 5 = 1):
SmartMedia Card not initialized.

ERNO = 64 (bit 6 = 1):
SmartMedia Card not inserted or wrong card.

ERNO = 128 (bit 7 = 1):
Block is empty.

ERNO = 272 (bit 7 = 1 and bit 4 = 1):
Card has too many defective blocks.

The operating system executes the functions FCDEL, FCINIT, FCWR and FCRD in the background. This procedures can take quite a long time (FCINIT > 10s) because the PLC user program is performed with priority. At output ERNO then it is displayed, in which processing phase the block is.

ERNO = 301hex
– the block has detected the edge and waits for enabling by the operating system;

ERNO = 302hex
– the operating system has assumed the task.

ERNO = 303hex
– the job is being performed.

During this phase the outputs ERR and RDY are set to FALSE.

The inputs and outputs can neither be duplicated nor inverted.
Data_storage_S90 Library 16

 function call in IL

CAL FCINIT1(FREI := FCINIT_FREI,
AUTO := FCINIT_AUTO)

LD FCINIT1.RDY
ST FCINIT_RDY
LD FCINIT1.ERR
ST FCINIT_ERR
LD FCINIT1.ERNO
ST FCINIT_ERNO

Note: In IL, the function call has to be performed in one line.

function call in ST

FCINIT1(FREI := FCINIT_FREI,
AUTO := FCINIT_AUTO);
FCINIT_RDY:=FCINIT1.RDY;
FCINIT_ERR:=FCINIT1.ERR;
FCINIT_ERNO:=FCINIT1.ERNO;
The function block reads a data set from a data segment of the SmartMedia Card and stores the read data set beginning at the start flag defined by SM.

### Parameters

<table>
<thead>
<tr>
<th>Instanz</th>
<th>FCRD</th>
<th>FCRD1</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>R_Y</td>
<td>Storage of a data set by a FALSE/TRUE edge</td>
</tr>
<tr>
<td>NB</td>
<td>INT</td>
<td>R_M</td>
<td>Number of data set blocks</td>
</tr>
<tr>
<td>SEG</td>
<td>INT</td>
<td>R_S</td>
<td>Number of the data segment</td>
</tr>
<tr>
<td>BNR</td>
<td>INT</td>
<td>R_B</td>
<td>Number of the block in the data segment</td>
</tr>
<tr>
<td>SM</td>
<td>INT</td>
<td>R_S</td>
<td>Start flag of the data set</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>R_DY</td>
<td>Writing procedure completed</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>R_ERR</td>
<td>Error has occurred</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>R_ERNO</td>
<td>Error number</td>
</tr>
</tbody>
</table>

### Description

The function block reads a data set from a data segment of the SmartMedia Card and stores the read data set beginning at the start flag defined by SM. The data of the data set have been stored in the SmartMedia Card by the function block FCWR.

**Important note:**

An access to the SmartMedia Card is only permitted by using the function blocks FCDEL, FCINIT, FCWR and FCRD.

The inputs SM and NB define which data are read from the SmartMedia Card. The input SEG defines the SmartMedia Card data segment. The number of data which are read from a block depends on input SM.

The following applies: 32 word data are read per block. The data of each block are secured by a checksum.

The reading procedure of a data record is carried out once by a FALSE/TRUE edge at input FREI. If no error occurred when reading the data, output RDY is set to "TRUE" and the outputs ERR and ERNO are set to zero. The data set is stored beginning at the defined start flag SM. Storing the data set can take several PLC cycles.

If an error occurs during the reading procedure, RDY and ERR are both set to TRUE. The error type is indicated at output ERNO.

The outputs RDY, ERR and ERNO are set to zero by a zero signal at input FREI.
FREI

Processing of the block is controlled via input FREI.
The following applies:
FREI = FALSE:
The outputs RDY, ERR and ERNO are set to "0" or "FALSE", respectively.
FREI = FALSE/TRUE edge:
The reading procedure of the data set is carried out once.
FREI = TRUE:
The block is not processed, i.e. it no longer changes its outputs.

NB

The number of the data set blocks is specified at input NB. The number of data which are read from a block depends on input SM. 32 word data are read per block.
Valid values: 1 ... 128

Example:
- SM = MW 02,00 / %MW1002.0 and NB = 1:
  Storing the data from MW 02,00 / %MW1002.0 to MW 03,15 / %MW1003.15
  (1 block = 32 word data)
- SM = MW 02,00 / %MW1002.0 and NB = 2:
  Storing the data from MW 02,00 / %MW1002.0 to MW 05,15 / %MW1005.15
  (2 blocks = 64 word data)

SEG

The number of the data segment of the SmartMedia Card is specified at input SEG. The following applies:
07 MC 90 R0101 (2 MB): 1...250
07 MC 90 R0201 (8 MB): 1...1023
(as of run-time system V5.0 and 07 KT 9x R02xx)

BNR

The block number in the data segment is specified at input BNR. Valid values: 0 ... 127

SM

The first INT flag for storing the data set is specified at input SM.

RDY

Output RDY indicates that reading the data record has been completed. This output has always to be considered together with output ERR.

The following applies:
RDY = TRUE and ERR = FALSE: The reading operation has been completed. The data set is stored beginning at the defined input SM.
RDY = TRUE and ERR = TRUE: An error occurred during the reading procedure. Output ERNO signalizes the error number.

ERR

Output ERR indicates whether an error occurred during the reading operation. This output has always to be considered together with output RDY. If an error has occurred, the following applies: RDY = TRUE and ERR = TRUE. Output ERNO signalizes the error number.

ERNO

At output ERNO an error number is output. This output has always to be considered together with the outputs RDY and ERR.
The error number is binary encoded and valid for all FC* blocks. The following applies:
ERNO = 0 (bits 0...15 = 0): No error occurred
ERNO = 1 (bit 0 = 1): Block number and number of blocks is greater than 127.
ERNO = 2 (bit 1 = 1): Data segment is greater than maximum possible.
ERNO = 4 (bit 2 = 1): Block not programmable or check sum error.
ERNO = 8 (bit 3 = 1): Block already contains data.
ERNO = 16 (bit 4 = 1): Segment not deletable or too many defective blocks.
ERNO = 32 (bit 5 = 1): SmartMedia Card not initialized.
ERNO = 64 (bit 6 = 1): SmartMedia Card not inserted or wrong card.
ERNO = 128 (bit 7 = 1): The block is empty.
ERNO = 272 (bit 7 = 1 and bit 4 = 1): The card has too many defective blocks.

The inputs and outputs can neither be duplicated nor inverted.
Function call in IL

CAL  FCRD1(FREI := FCRD_FREI, 
     NB := FCRD_NB, SEG := FCRD_SEG, 
     BNR := FCRD_BNR, SM := FCRD_SM)

LD   FCRD1.RDY
ST   FCRD_RDY
LD   FCRD1.ERR
ST   FCRD_ERR
LD   FCRD1.ERNO
ST   FCRD_ERNO

Note: In IL, the function call has to be performed in one line.

Function call in ST

FCRD1(FREI := FCRD_FREI, 
     NB := FCRD_NB, SEG := FCRD_SEG, 
     BNR := FCRD_BNR, SM := FCRD_SM);

FCRD_RDY:=FCRD1.RDY;
FCRD_ERR:=FCRD1.ERR;
FCRD_ERNO:=FCRD1.ERNO;
WRITE DATA SET TO SMARTMEDIA CARD

The function block writes a data set to a data segment in the SmartMedia Card.

Block data
Available as of PLC run-time system:
- V4.0 for 07 MC 90 R0101 (2 MB)
- V5.0 for 07 MC 90 R0201 (8 MB)
and 07 KT 9x R02xx

Contained in library:
- ABB-BIB4.LIB
- Datenablage_S90_V41.LIB

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>FCWR</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Storage of a data set by a FALSE/TRUE edge</td>
</tr>
<tr>
<td>NB</td>
<td>INT</td>
<td>Number of data set blocks</td>
</tr>
<tr>
<td>SEG</td>
<td>INT</td>
<td>Number of the data segment</td>
</tr>
<tr>
<td>BNR</td>
<td>INT</td>
<td>Number of the block in the data segment</td>
</tr>
<tr>
<td>SM</td>
<td>INT</td>
<td>Start flag of the data set</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
<td>Writing procedure completed</td>
</tr>
<tr>
<td>ERR</td>
<td>BOOL</td>
<td>Error has occurred</td>
</tr>
<tr>
<td>ERNO</td>
<td>INT</td>
<td>Error number</td>
</tr>
</tbody>
</table>

Description
The function block writes a data set to a data segment in the SmartMedia Card. A deletion procedure (function block FCDEL) always deletes a complete data segment. The following space is available on the card after formatting:

07 MC 90 R0101 (2 MB):
250 segments with 128 blocks of 64 bytes each, i.e. 2000 kB can be stored in total.

07 MC 90 R0201 (8 MB):
1023 segments with 128 blocks of 64 bytes each, i.e. 8184 kB can be stored in total (as of run-time system V5.0 and 07 KT 9x R02xx).

After deleting, each of this 128 blocks can be written only once. If a block, already containing data, is to be overwritten, the entire data segment must be deleted beforehand.

When doing so, all data is lost in this data segment.

Important note:
An access to the SmartMedia Card is only permitted by using the function blocks FCDEL, FCINIT, FCWR and FCRD.

The inputs SM and NB define which data are written to the SmartMedia Card. The input SEG defines the data segment in the SmartMedia Card. The number of data which can be stored in a block depends on input SM.

The following applies: 32 word data are written per block. The data of each block are secured by a checksum.

When a writing procedure of a data set is started (FALSE/TRUE edge at input FREI), the data of the data set must not be changed until the end of the writing procedure (RDY = TRUE). Storing a data set in the SmartMedia Card can take several PLC cycles.
WRITE DATA SET TO SMARTMEDIA CARD

With a FALSE/TRUE edge at input FREI the data set writing is performed once. Until storing has not been finished (RDY = TRUE), input FREI will not be evaluated again.

Important note:
Prior to the writing procedure the entire segment is read and deleted. Then it is written. If power goes down during the writing procedure, all data is lost.
After the writing operation has been completed, the block outputs RDY, ERR and ERNO are updated. The storage was successful, if RDY = TRUE and ERR = FALSE. In the case of RDY = TRUE and ERR = TRUE an error occurred. The error type is signaled at output ERNO.

After storing the data set in the SmartMedia Card, the block outputs RDY, ERR and ERNO are set to zero by a signal "FALSE" at input FREI. A new FALSE/TRUE edge at input FREI starts a new writing procedure. Since without a previous deletion of the data segment no new data can be written to blocks which already contain data, the input BNR must point to the next free block for the next writing procedure.

FREI  BOOL
Processing of the block is controlled via input FREI.
FREI = FALSE:
The outputs RDY, ERR and ERNO are set to "0" or "FALSE", respectively. This is not valid during a writing procedure.
FREI = FALSE/TRUE edge:
Writing of the data set is started once. Until the writing procedure has not been finished (RDY = TRUE), input FREI must remain TRUE.

NB  INT
The number of the data set blocks is specified at input NB. The number of data which can be stored in a block depends on input SM. 32 word data are stored per block.
Valid values: 1 ... 128
Example:
- SM = MW 02,00 / %MW1002.0 and NB = 1:
  Writing the data from MW 02,00 / %MW1002.0 to MW 03,15 / %MW1003.15
  (1 block = 32 word data)
- SM = MW 02,00 / %MW1002.0 and NB = 2:
  Writing the data from MW 02,00 / %MW1002.0 to MW 05,15 / %MW1005.15
  (2 blocks = 64 word data)

SEG  INT
The number of the data segment in the SmartMedia Card is specified at input SEG. The following applies:
07 MC 90 R0101 (2 MB): 1...250
07 MC 90 R0201 (8 MB): 1...1023
(as of run-time system V5.0 and 07 KT 9x R02xx)

BNR  INT
The block number in the data segment is specified at input BNR.
Valid values: 0 ... 127

SM  INT
The first INT flag of the data set is specified at input SM. When a writing procedure of a data set is started (FALSE/TRUE edge at input FREI), the data of the data set must not be changed until the end of the writing procedure (RDY = TRUE).

RDY  BOOL
Output RDY indicates that writing of data record has been completed. This output has always to be considered together with output ERR.
The following applies:
RDY = TRUE and ERR = FALSE: The writing operation has been completed. The data set has been stored in the SmartMedia Card.
RDY = TRUE and ERR = TRUE: An error occurred during the writing procedure. Output ERNO signalizes the error number.

ERR  BOOL
Output ERR indicates whether an error occurred during the writing procedure. This output has always to be considered together with output RDY. If an error has occurred, the following applies: RDY = TRUE and ERR = TRUE. Output ERNO signalizes the error number.
WRITE DATA SET TO SMARTMEDIA CARD

ERNO

At output ERNO an error number is output. This output has always to be considered together with the outputs RDY and ERR.

The error number is binary encoded and valid for all FC\(^*\) blocks.

The following applies:

- **ERNO = 0 (bits 0..15 = 0):** No error occurred
- **ERNO = 1 (bit 0 = 1):** Block number and number of blocks is greater than 127.
- **ERNO = 2 (bit 1 = 1):** Data segment is greater than maximum possible.
- **ERNO = 4 (bit 2 = 1):** Block not programmable or check sum error.
- **ERNO = 8 (bit 3 = 1):** Block already contains data.
- **ERNO = 16 (bit 4 = 1):** Segment not deletable or too many defective blocks.
- **ERNO = 32 (bit 5 = 1):** SmartMedia Card not initialized.
- **ERNO = 64 (bit 6 = 1):** SmartMedia Card not inserted or wrong card.
- **ERNO = 128 (bit 7 = 1):** Block is empty.
- **ERNO = 272 (bit 7 = 1 and bit 4 = 1):** The card has too many defective blocks.

The inputs and outputs can neither be duplicated nor inverted.

**Function call in IL**

```
CAL FCWR1(FREI := FCWR_FREI, 
        NB := FCWR_NB, SEG := FCWR_SEG, 
        BNR := FCWR_BNR, SM := FCWR_SM)

LD FCWR1.RDY
ST FCWR_RDY
LD FCWR1.ERR
ST FCWR_ERR
LD FCWR1.ERNO
ST FCWR_ERNO
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
FCWR1(FREI := FCWR_FREI, 
      NB := FCWR_NB, SEG := FCWR_SEG, 
      BNR := FCWR_BNR, SM := FCWR_SM);
FCWR_RDY:=FCWR1.RDY;
FCWR_ERR:=FCWR1.ERR;
FCWR_ERNO:=FCWR1.ERNO;
```
Glossary

BOOL

Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT

DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

DWORD

DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

INT

INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

WORD

WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

Function blocks

Function blocks are subroutines which can have as much inputs, outputs and internal variables as required. They are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. This allows for example to realize counters which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters. A function has maximally one output parameter. Note that the output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
COMPONENTS OF THE DATA STORAGE LIBRARY 2

D
Data storage S90
FCDEL 11
FCINIT 14
FCRD 17
FCWR 20
FDEL 3
FRD 5
FWR 8

F
FCDEL S90 11
FCINIT S90 14
FCRD S90 17
FCWR S90 20
FDEL S90 3
FRD S90 5
FWR S90 8
Contents

The System Info Library 2
PRECONDITION FOR THE USE OF THE SYSTEM INFO LIBRARY ........................................ 2
COMPONENTS OF THE SYSTEM INFO LIBRARY ............................................................... 2

Block overview arranged according to their call names 2
INTERROGATION OF INFORMATION ABOUT THE USER PROGRAMM STACK
GET_STACKINFO ... 3
INTERROGATION OF INFORMATION ABOUT THE HEAP
HEAP_INFO.......... 4
INTERROGATION OF INFORMATION ABOUT THE PLC
IDENT................... 6
INTERROGATION OF INFORMATION ABOUT THE USER PROGRAM
PROJEKTINFO ......... 8
READING THE SYSTEM TIME
SYS_TIME............... 9
INTERROGATION, WHETHER AN EXCEPTION OCCURRED DURING FLOATING-POINT CALCULATION
FPUEXINFO.......... 10
INTERROGATION OF THE STARTING ADDRESSES OF THE DUAL-PORT RAM
HW_INFO ............... 12

Glossary 14

Index 16
The System Info Library

PRECONDITION FOR THE USE OF THE SYSTEM INFO LIBRARY

Note:
The blocks of the System Info Library only run in the RUN mode of the PLC, not in the simulation mode.

COMPONENTS OF THE SYSTEM INFO LIBRARY

The following function blocks are contained in the System Info Library:

- GET_STACKINFO - Interrogation of information about the user program stack
- HEAP_INFO - Interrogation of information about the heap
- IDENT - Interrogation of information about the PLC
- PROJEKTINFO - Interrogation of information about the user program
- SYS_TIME - Reading the system time
- FPUEXINFO - Interrogation, whether an exception occurred during floating-point calculation
- HW_INFO - Interrogation of the starting addresses of the dual-port RAM

Detailed descriptions of the blocks can be found in the following sections.

Block overview arranged according to their call names

Explanation of the abbreviations:
FBmV … Function block with historical values
FBoV … Function block without historical values
F … Function

<table>
<thead>
<tr>
<th>CE Name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPUEXINFO</td>
<td>FBoV</td>
<td>Interrogation, whether an exception occurred during floating-point calculation</td>
<td>10</td>
</tr>
<tr>
<td>GET_STACKINFO</td>
<td>FBoV</td>
<td>Interrogation of information about the user program stack</td>
<td>3</td>
</tr>
<tr>
<td>HEAP_INFO</td>
<td>FBoV</td>
<td>Interrogation of information about the heap</td>
<td>4</td>
</tr>
<tr>
<td>HW_INFO</td>
<td>FBoV</td>
<td>Interrogation of the starting addresses of the dual-port RAM</td>
<td>12</td>
</tr>
<tr>
<td>IDENT</td>
<td>FBoV</td>
<td>Interrogation of information about the PLC</td>
<td>6</td>
</tr>
<tr>
<td>PROJEKTINFO</td>
<td>F</td>
<td>Interrogation of information about the user program</td>
<td>8</td>
</tr>
<tr>
<td>SYS_TIME</td>
<td>F</td>
<td>Reading the system time</td>
<td>9</td>
</tr>
</tbody>
</table>
INTERROGATION OF INFORMATION ABOUT THE USER PROGRAM STACK

GET_STACKINFO

A separate stack is established for every PLC user program. The GET_STACKINFO block interrogates the amount of the stack used (or free).

Block data
Available as of PLC run-time system: V4.0
Contained in library: Systeminfo_S90_V41.LIB and Systeminfo_S90_V42.LIB

Remark:

Block type
Function block without historical values

Parameters
- Instance GET_STACKINFO Instance name
- EN BOOL Enabling of the block processing
- ACT_FREI INT Current number of free bytes
- MIN_FREI INT Number of free bytes since the user program was created

Description
A separate stack is established for every PLC user program. The GET_STACKINFO block interrogates the amount of the stack used (or free).

EN BOOL
The EN input controls the processing of the block.

EN FALSE:
All block outputs remain unchanged.

EN TRUE:
The block is processed, i.e. the outputs are updated with every cycle.

Function call in IL
CAL GET_STACKINFO1(EN := TRUE)
LD GET_STACKINFO1.ACT_FREI
ST STACK_ACT_FREI
LD GET_STACKINFO1.MIN_FREI
ST STACK_MIN_FREI

Function call in ST
GET_STACKINFO1(EN := TRUE);
ACT_FREI := GET_STACKINFO1.ACT_FREI;
MIN_FREI := GET_STACKINFO1.MIN_FREI;

ACT_FREI INT
The ACT_FREI output indicates the current number of free bytes in the stack.

MIN_FREI INT
The MIN_FREI output indicates how many bytes were at least free in the stack since the user program was created, i.e. since the download and the first RUN or since the user program was loaded from the Flash EPROM.
This block enables the user to get information about the heap.

**Block data**

- Available as of PLC run-time system: V4.0
- Contained in library: Systeminfo_S90_V41.LIB and Systeminfo_S90_V42.LIB

**Block type**

Function block without historical values

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>HEAP_INFO</td>
<td>Instance name</td>
</tr>
<tr>
<td>TotalFree</td>
<td>DWORD</td>
<td>Total amount of memory space, currently free</td>
</tr>
<tr>
<td>TotalUsed</td>
<td>DWORD</td>
<td>Total amount of memory space, currently used</td>
</tr>
<tr>
<td>FreeBlocks</td>
<td>DWORD</td>
<td>Number of free blocks</td>
</tr>
<tr>
<td>UsedBlocks</td>
<td>DWORD</td>
<td>Number of reserved blocks</td>
</tr>
<tr>
<td>LargestBlock</td>
<td>DWORD</td>
<td>Largest coherent free memory space</td>
</tr>
</tbody>
</table>

**Description**

The heap is an area in the RAM usable for the operating system for varying tasks (and varying need of space), i.e. this area is not allocated at the beginning. For instance, the tasks "Send program" or "Online change" use memory space temporarily in the heap and free the space again after use.

The ARCNET blocks also reserve memory space in the heap for buffers temporarily used. For the new ARCNET blocks, the size of these buffers is user-defined.

Since depending on the application (e.g. how many PLC tasks were created) the free memory space is different in size, the HEAP_INFO function block can be used to find out the size of free memory in the heap.

The heap can be more or less fragmented, i.e. it is possible that there are 20 Kbytes free in total, but there is no coherent memory space of 20 Kbytes for creating such a block.

If more memory space is demanded than really exists, an error is output (FK3 - error code 181).
**TotalFree**
DWord
The TotalFree output indicates the total amount of free memory space in bytes currently available.

**TotalUsed**
DWord
The TotalUsed output indicates the total amount of currently used memory space in bytes.

**FreeBlocks**
DWord
The FreeBlocks output indicates the number of free blocks.

**UsedBlocks**
DWord
The UsedBlocks output indicates the number of reserved (used) blocks.

**LargestFree**
DWord
The LargestFree output indicates the largest coherent free memory space in bytes.

---

**Function call in IL**

CAL HEAP_INFO1()
LD HEAP_INFO1.TotalFree
ST TotalFree
LD HEAP_INFO1.TotalUsed
ST TotalUsed
LD HEAP_INFO1.FreeBlocks
ST FreeBlocks
LD HEAP_INFO1.UsedBlocks
ST UsedBlocks
LD HEAP_INFO1.LargestFree
ST LargestFree

**Function call in ST**

HEAP_INFO1();
TotalFree := HEAP_INFO1.TotalFree
TotalUsed := HEAP_INFO1.TotalUsed
FreeBlocks := HEAP_INFO1.FreeBlocks
UsedBlocks := HEAP_INFO.UsedBlocks
LargestFree := HEAP_INFO.LargestFree
INTERROGATION OF INFORMATION ABOUT THE PLC

IDENT

The IDENT function block supplies information about the PLC from which the PLC program was started.

Block data
Available as of PLC run-time system: V4.0
Contained in library: Systeminfo_S90_V41.LIB and Systeminfo_S90_V42.LIB

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>TYP</td>
<td>INT</td>
<td>PLC type</td>
</tr>
<tr>
<td>RUB</td>
<td>INT</td>
<td>PLC version</td>
</tr>
<tr>
<td>VER</td>
<td>INT</td>
<td>Software version as a 3-digit number</td>
</tr>
</tbody>
</table>

Description
The IDENT function block supplies information about the PLC from which the PLC program was started.

FREI       BOOL
Enabling for block processing.

FREI = TRUE
If FREI = TRUE, the outputs of the block are updated.

FREI = FALSE
The block is not processed. The outputs remain unchanged.

TYP        INT
If FREI = TRUE, the TYP output indicates the PLC type.
95         means 07 KT 95
96         means 07 KT 96
97         means 07 KT 97 or 07 SL 97
98         means 07 KT 98

VER        INT
The VER output indicates the software version as a 3-digit number: e.g. 416 means Version V4.16.

RUB        INT
The RUB output indicates the PLC version R0xxx:

<table>
<thead>
<tr>
<th>Version</th>
<th>PLC, equipped with the couplers...</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 / 200</td>
<td>no internal couplers</td>
</tr>
<tr>
<td>120 / 220</td>
<td>PROFIBUS DP coupler</td>
</tr>
<tr>
<td>160 / 260</td>
<td>ARCNET coupler</td>
</tr>
<tr>
<td>162 / 262</td>
<td>ARCNET + PROFIBUS DP coupler</td>
</tr>
<tr>
<td>165 / 265</td>
<td>ARCNET + DeviceNet coupler</td>
</tr>
<tr>
<td>268</td>
<td>ARCNET + CANopen coupler</td>
</tr>
<tr>
<td>270</td>
<td>Ethernet coupler</td>
</tr>
<tr>
<td>272</td>
<td>Ethernet + PROFISUB DP coupler</td>
</tr>
<tr>
<td>276</td>
<td>Ethernet + ARCNET coupler</td>
</tr>
<tr>
<td>278</td>
<td>Ethernet + CANopen coupler</td>
</tr>
<tr>
<td>280</td>
<td>CANopen coupler</td>
</tr>
</tbody>
</table>
Function call in IL

CAL IDENT1(FREI := TRUE)

LD IDENT1.TYP
ST IDENT_TYP

LD IDENT1.RUB
ST IDENT_RUB

LD IDENT1.VER
ST IDENT VER

Function call in ST

IDENT1(FREI := TRUE);

IDENT_TYP := IDENT1.TYP;
IDENT_RUB := IDENT1.RUB;
IDENT_VER := IDENT1.VER;
The ProjektInfo function supplies information about the user program started on the PLC. The information entered in the menu "Project/Project information" of 907 AC 1131 are output as a string.

**Block data**

Available as of PLC run-time system: V4.0
Contained in library: Systeminfo_S90_V41.LIB and Systeminfo_S90_V42.LIB

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREI</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>ProjektInfo</td>
<td>STRING[255]</td>
<td>Output string</td>
</tr>
</tbody>
</table>

**Description**

The ProjektInfo function supplies information about the user program started on the PLC. The information entered in the menu "Project/Project information" of 907 AC 1131 are output as a string.

**FREI**

Enabling of the block processing.

- **FREI** TRUE
  - If FREI = TRUE the output string is updated.
- **FREI** FALSE
  - The block is not processed. The output string is deleted.

**Function block in IL**

LD ProjektInfo_Frei
ProjektInfo
ST ProjektInfo

**Projectinfo**

STRING[255]

The output string contains the following information:

- the Directory\Projectname of the PLC program
- the date of last change
- information entered under Designation/Author/Version/Description

The different entries are delimited by spaces (blanks).

A maximum of 155 bytes is output. If the project information is shorter than 155 bytes, the output string is filled with zero bytes.

**Function call in ST**

projInfo := Projektinfo(FREI := ProjektInfo_Frei);
READING THE SYSTEM TIME

The SYS_TIME function supplies the system tick as a double word in milliseconds.

Block data
Available as of PLC run-time system: V4.0
Contained in library: Systeminfo_S90_V41.LIB and Systeminfo_S90_V42.LIB

Remark:

Block type
Function

Parameters
FREI BOOL Enabling of the block processing
SYS_TIME DWORD System tick in ms

Description
The SYS_TIME function supplies the system tick as a double word. The system tick is resolved into milliseconds. Among other things, the system tick is used as time base for both the user program and all time-dependent function blocks. After a RESET, the system tick begins counting with 0. The first overflow is reached after 49 days. With an overflow, the counter begins at 0 again.

FREI BOOL Enabling of the block processing.
FREI TRUE If FREI = TRUE, the block output is updated.
FREI FALSE The block is not processed. The output remains unchanged since the program start.

SYS_TIME DWORD System tick in milliseconds.

Function call in IL
LD TRUE
SYS_TIME
ST Systemtick

Function call in ST
Systemtick := SYS_TIME(FREI := TRUE);
INTERROGATION, WHETHER AN EXCEPTION OCCURRED DURING FLOATING-POINT CALCULATION

Using the FPUEXINFO function block, it can be found out whether an exception occurred during a floating-point calculation.

**Block data**

- Available as of PLC run-time system: V4.16
- Contained in library: Systeminfo_S90_V42.LIB

**Block type**

Function block without historical values

**Parameters**

- **Instanz**: FPUEXINFO Instance name
- **ERR**: BOOL ERR=TRUE: Exception after floating-point calculation
- **INFO**: STRING(255) Information about the calculation

**Description**

Using the FPUEXINFO function block, it can be found out whether an exception occurred during a floating-point calculation. The setting of the reaction to floating-point exceptions is performed with the system constant KW81.08 / %MW3081.8.

The following applies:

\[
\text{KW81.08 AT } \%\text{MW3081.8} : \text{INT} := 0; \text{ (* default *)}
\]

- The FK2 error \(182_{\text{Dec}} = B6_{\text{Hex}}\) is triggered with a floating-point exception (e.g. with SQRT(-1)). The PLC turns to STOP. Using the calling hierarchy of the 907 AC 1131, it can be found out where the error occurred.

\[
\text{KW81.08 AT } \%\text{MW3081.8} : \text{INT} := 1;
\]

In case of a floating-point exception, no error is triggered. Using the FPUEXINFO function block, it can be found out, whether an floating-point exception occurred within the calculation. The user can now decide, whether to continue with default values or to shut down the machine specifically.

**ERR**

- **BOOL**
  - The ERR output indicates whether a floating-point exception has occurred.
  - ERR=TRUE The previous calculation has triggered a floating-point exception.
  - ERR=FALSE The previous calculation did not trigger a floating-point exception.

**INFO**

- **STRING(255)**
  - The INFO output string supplies further information about the calculation.
INTERROGATION, WHETHER AN EXCEPTION OCCURRED DURING FLOATING-POINT CALCULATION

### Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>FPUEXINFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>FPUEXINFO1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>FPUEXINFO1.ERR</td>
</tr>
<tr>
<td>LD</td>
<td>FPUEXINFO1.INFO</td>
</tr>
<tr>
<td>ST</td>
<td>FPUEXINFO1.INFO</td>
</tr>
</tbody>
</table>

### Function call in ST

<table>
<thead>
<tr>
<th>CAL</th>
<th>FPUEXINFO1.ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>FPUEXINFO1.ERR</td>
</tr>
<tr>
<td>ST</td>
<td>FPUEXINFO1.INFO</td>
</tr>
</tbody>
</table>

### Programming example:

Under "Global Variables" / "System constants", the initialization value of KW81_08 is set to 1:

KW81_08 AT %MW3081.8 : INT := 1; (* Do not trigger an FK2 error *)

PROGRAM PLC_PRG
VAR
  FPUEXINFO1 : FPUEXINFO;
  rV1 : REAL := -1.0;
  rV2 : REAL;
  bError : BOOL;
  bWarning : BOOL;
END_VAR

bWarning := bError := FALSE;
(* Floating-point calculation *)

rV2 := SQRT(rV1);
(* Check, whether an exception has occurred *)

FPUEXINFO1();

IF FPUEXINFO1.ERR=TRUE THEN
  (* here: evaluation of the exception *)
  (* e.g. shut down machine or continue with standard values or corrected values *)
  rV1 := 1.0;
  bWarning := TRUE;
  (* same calculation with corrected values *)
  rV2 := SQRT(rV1);
  (* check again.. *)
  FPUEXINFO1();
  IF FPUEXINFO1.ERR = TRUE THEN
    bError := TRUE;
  END_IF
END_IF
(* here: e.g. evaluation bWarning, bError.. *)
The HW_INFO function block supplies the start addresses of the dual-port RAM (DPR) of the PLC.

**Parameters**

- **Instanz**: HW_INFO
- **EN**: BOOL
- **DPR_EXT**: DWORD
- **DPR_EA**: DWORD
- **DPR_CS31**: DWORD
- **DPR_INT**: DWORD

**Description**

The HW_INFO function block supplies the start addresses of the dual-port RAM (DPR) of the PLC.

- **EN**: BOOL
  - Enabling of the block processing.
  - FREI: TRUE
    - If FREI = TRUE, the block outputs are updated.
  - FREI: FALSE
    - The block is not processed. The outputs remain unchanged since the program start.

- **DPR_EXT**: DWORD
  - At the DPR_EXT output the start address of the dual-port RAM of the external networking interface is available.

- **DPR_EA**: DWORD
  - At the DPR_EA output the start address of the dual-port RAM of the onboard IO processor is available.

- **DPR_CS31**: DWORD
  - At the DPR_CS31 output the start address of the dual-port RAM of the CS31 bus processor is available.

- **DPR_INT**: DWORD
  - At the DPR_INT output the start address of the dual-port RAM of the internal couplers is available.
## Function call in IL

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>HW_INFO(EN := TRUE)</td>
</tr>
<tr>
<td>LD</td>
<td>HW_INFO1.DPR_EXT</td>
</tr>
<tr>
<td>ST</td>
<td>HW_INFO_DPR_EXT</td>
</tr>
<tr>
<td>LD</td>
<td>HW_INFO1.DPR_EA</td>
</tr>
<tr>
<td>ST</td>
<td>HW_INFO_DPR_EA</td>
</tr>
<tr>
<td>LD</td>
<td>HW_INFO1.DPR_CS31</td>
</tr>
<tr>
<td>ST</td>
<td>HW_INFO_DPR_CS31</td>
</tr>
<tr>
<td>LD</td>
<td>HW_INFO1.DPR_INT</td>
</tr>
<tr>
<td>ST</td>
<td>HW_INFO_DPR_INT</td>
</tr>
</tbody>
</table>

## Function call in ST

```plaintext
HW_INFO1(EN := TRUE);
HW_INFO_DPR_EXT := HW_INFO1.DPR_EXT
HW_INFO_DPR_EA := HW_INFO1.DPR_EA
HW_INFO_DPR_CS31 := HW_INFO1.DPR_CS31
HW_INFO_DPR_INT := HW_INFO1.DPR_INT
```
Glossary

**BOOL**

Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

**DINT**

DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**DWORD**

DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**INT**

INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**WORD**

WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.
**Functions**

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

**Function blocks**

Function blocks are subroutines which can have as much inputs, outputs and internal variables as required. They are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. This allows for example to realize counters which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters. A function has maximally one output parameter. Note that the output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

   For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

   For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

   The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
Components of the System Info Library 2

F
FPUEXINFO S90 10

G
GET_STACKINFO S90 3

H
HEAP_INFO S90 4
HW_INFO S90 12

I
IDENT S90 6

P
Projektinfo S90 8

S
SYS_TIME S90 9
SystemInfo S90
  FPUEXINFO 10
  GET_STACKINFO 3
  HEAP_INFO 4
  HW_INFO 12
  IDENT 6
  Projektinfo 8
  SYS_TIME 9
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

PROFIBUS
Function Block Library
90 Series
Contents

The PROFIBUS library 2

PREREQUISITES FOR THE USAGE OF THE LIBRARY 2
COMPONENTS OF THE LIBRARIES 2
PROFIBUS-DP MASTER 3
PROFIBUS-DP SLAVE 4
ERR - ERROR MESSAGES OF THE PROFIBUS BLOCKS 5

Overview of blocks arranged according to their call names 6

READING OUT INFORMATION ABOUT THE INSTALLED PROFIBUS COUPLERS PROFI_INFO 7
POLLING DETAILED DIAGNOSTIC DATA OF A DP SLAVE DPM_SLVDIAG 10
READING OUT THE PROFIBUS COUPLER STATUS DPM_STAT 19
READING OUT THE STATUS SURVEY OF ALL DP SLAVES DPM_SYSĐIAG 25
SENDING GLOBAL CONTROL COMMANDS TO THE DP SLAVE DPM_CTRL 27
SENDING USER PARAMETERS TO A DP SLAVE DPM_SETPRM 32
READING THE INPUT DATA OF A SLAVE WHICH IS NOT ASSIGNED TO THE MASTER DPM_READ_INPUT 34
READING THE OUTPUT DATA OF A SLAVE WHICH IS NOT ASSIGNED TO THE MASTER DPM_READ_OUTPUT 36
READING OUT THE STATUS OF THE PROFIBUS COUPLER DPS_STAT 38
SENDING THE ERROR FLAGS OF THE BASIC UNIT FOR AN EXTENDED DIAGNOSIS DPS_EXTĐIAG 42
READING OUT THE USER PARAMETERS DPS_GETPRM 45

Glossary 49

Index 51
The PROFIBUS library

PREREQUISITES FOR THE USAGE OF THE LIBRARY

The PROFIBUS couplers of the AC31 series 90 controllers can be operated as DP master or as DP slave (exception: 07 SL 97 can only be used as DP master). Up to version V4.2 (incl.) the function blocks for both operation modes were summarized in one common 907 AC 1131 library named PROFIBUS_S90_VXX.LIB. Because version V4.3 contains additional new function blocks, the libraries are now divided to PROFIBUS_Master_S90_VXX.LIB and PROFIBUS_Slave_S90_VXX.LIB according to the operation mode. The library PROFIBUS_S90_VXX.LIB can still be used, if only the functions contained in this library are used within existing projects. However, it is recommended to include the new PROFIBUS library according to the used operation mode instead of using the old common library. For new projects, only the new libraries should be used which are separated according to the operation mode.

The function blocks contained in the PROFIBUS libraries access to both, the PLC run time system and directly to the coupler. The definitions and functions required for this are stored in the internal library COUPLER_S90_V41.LIB.

When using the PROFIBUS libraries, the library COUPLER_S90_V41.LIB must be included additionally in the project.

Note:

Neither the blocks of the PROFIBUS library nor the PROFIBUS communication can run in simulation mode. The PROFIBUS communication only runs in the RUN mode of the PLC, not with single cycle, step and breakpoint.

COMPONENTS OF THE LIBRARIES

For normal cyclic data exchange via PROFIBUS-DP it is not necessary to include the libraries. The libraries contain additional function blocks which allow an easy handling of the PROFIBUS coupler in the individual operation modes. Additionally, various data types are defined in this library. These structures enable a clear presentation of data sets.
PROFIBUS-DP MASTER

Function blocks

The library PROFIBUS_Master_S90_Vxx.LIB contains the following function blocks:

Group: General
PROFI_INFO - Reading information about the installed couplers

Group: Status / Diagnostics
DPM_SLVDIAG - Polling diagnostic data from a DP slave
DPM_STAT - Reading out the state of the PROFIBUS coupler operating as DP master
DPM_SYSDIAG - Displaying status surveys of all DP slaves

Group: Control system
DPM_CTRL - Sending control commands to the DP slaves

Group: Parameters
DPM_SETPRM - Sending user parameters to a DP slave during runtime

Group: Acyclic reading of I/O data
DPM_READ_INPUT - Reading the input data of a slave which is not assigned to the master
DPM_READ_OUTPUT - Reading the output data of a slave which is not assigned to the master

Detailed information about the various blocks can be found in the following sections.

Data types

In the library for the operation mode PROFIBUS-DP master the following data types (structures) are defined:

DPM_COM_ERR_TYPE - Communication error
DPM_STATE_BITS_TYPE - Bits for coupler state description
STATIONSTATUS_1_TYPE - Stationstatus_1 (DP slave diagnosis according to standard)
STATIONSTATUS_2_TYPE - Stationstatus_2 (DP slave diagnosis according to standard)
STATIONSTATUS_3_TYPE - Stationstatus_3 (DP slave diagnosis according to standard)

For detailed information about the different data types please refer to the particular description of the corresponding block.
PROFIBUS-DP SLAVE

Function blocks

**Note:** The block PROFI_INFO contained in the PROFIBUS master library can also be used in the coupler operation mode DP slave. For this purpose, the corresponding master library must be loaded in addition to the PROFIBUS slave library.

**Group:** Status / Diagnostics

- **DPS_STAT** - Reading out the state of the PROFIBUS coupler operating as DP slave
- **DPS_EXTDIAG** - Sending the error flags of the basic unit for an extended diagnosis

**Group:** Parameters

- **DPS_GETPRM** - Reading out the user parameters

Detailed information about the various blocks can be found in the following sections.

**Data types**

In the library for the operation mode PROFIBUS-DP slave the following data types (structures) are defined:

- **DPS_STATE_BITS_TYPE** - Bits for coupler state description

For detailed information about the different data types please refer to the particular description of the corresponding block.
ERR - ERROR MESSAGES OF THE PROFIBUS BLOCKS

If the output ERR of a PROFIBUS block is set to a value unequal to 0, this means that either one of the block input values is invalid or an internal error occurred during the block execution. The error cause can be determined with the help of the following tables.

<table>
<thead>
<tr>
<th>ERR</th>
<th>DEC</th>
<th>HEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8193</td>
<td>2001</td>
<td>CONO</td>
<td>Invalid value at block input CONO</td>
</tr>
<tr>
<td>8194</td>
<td>2002</td>
<td>CONO</td>
<td>Coupler model in slot CONO does not correspond to the block type</td>
</tr>
<tr>
<td>8195</td>
<td>2003</td>
<td>CONO</td>
<td>No DP master coupler in slot CONO</td>
</tr>
<tr>
<td>8196</td>
<td>2004</td>
<td>CONO</td>
<td>DP master coupler in slot CONO is not in OPERATE state</td>
</tr>
<tr>
<td>8197</td>
<td>2005</td>
<td>CONO</td>
<td>Invalid status survey type</td>
</tr>
<tr>
<td>8198</td>
<td>2006</td>
<td>CONO</td>
<td>Invalid slave address at block input</td>
</tr>
<tr>
<td>8199</td>
<td>2007</td>
<td>CONO</td>
<td>Invalid module number at block input</td>
</tr>
<tr>
<td>8200</td>
<td>2008</td>
<td>CONO</td>
<td>Invalid module index at block input</td>
</tr>
<tr>
<td>8203</td>
<td>200B</td>
<td>CONO</td>
<td>No DP slave coupler in slot CONO</td>
</tr>
<tr>
<td>9377</td>
<td>24A1</td>
<td></td>
<td>Coupler reports invalid slave address</td>
</tr>
</tbody>
</table>
Overview of blocks arranged according to their call names

Abbreviations:
- FBmV … Function block with historical values
- FBoV … Function block without historical values
- F … Function

<table>
<thead>
<tr>
<th>CE name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPM_CTRL</td>
<td>FBoV</td>
<td>Sending control information to installed couplers</td>
<td>27</td>
</tr>
<tr>
<td>DPM_READ_INPUT</td>
<td>FBmV</td>
<td>Reading the input data of a slave which is not assigned to the master</td>
<td>34</td>
</tr>
<tr>
<td>DPM_READ_OUTPUT</td>
<td>FBmV</td>
<td>Reading the output data of a slave which is not assigned to the master</td>
<td>36</td>
</tr>
<tr>
<td>DPM_SETPRM</td>
<td>FBmV</td>
<td>Sending of user parameters to a DP slave during the runtime</td>
<td>32</td>
</tr>
<tr>
<td>DPM_SLVDIAG</td>
<td>FBmV</td>
<td>Polling of detailed diagnostic data of a DP slave</td>
<td>10</td>
</tr>
<tr>
<td>DPM_STAT</td>
<td>FBoV</td>
<td>Reading out the status of the PROFIBUS coupler</td>
<td>19</td>
</tr>
<tr>
<td>DPM_SYSDIAG</td>
<td>FBoV</td>
<td>Reading out the status survey of all DP slaves</td>
<td>25</td>
</tr>
<tr>
<td>DPS_STAT</td>
<td>FBoV</td>
<td>Reading out the status of the PROFIBUS coupler</td>
<td>27</td>
</tr>
<tr>
<td>DPS_EXTDIAG</td>
<td>FBmV</td>
<td>Sending the error flags of the basic unit for an extended diagnosis</td>
<td>42</td>
</tr>
<tr>
<td>DPS_GETPRM</td>
<td>FBmV</td>
<td>Reading out the user parameters</td>
<td>45</td>
</tr>
<tr>
<td>PROFI_INFO</td>
<td>FBoV</td>
<td>Reading information about the installed couplers</td>
<td>7</td>
</tr>
</tbody>
</table>
READING OUT INFORMATION ABOUT THE INSTALLED PROFIBUS COUPLERS

The block PROFI_INFO outputs coupler related information. The following items are displayed: Coupler type and model, operation mode, manufacturing date, device number and serial number as well as firmware designation and firmware version.

Block data
Available as of PLC runtime system: V4.0
Included in library:
- PROFI40.LIB
- PROFIBUS_S90_V41.LIB
- PROFIBUS_S90_V42.LIB
- PROFIBUS_Master_S90_V43.LIB

CONO, ERR - type changed

Block type
Function block without historical values

Parameters
Instance | PROFI_INFO | Instance name
CONO | BYTE | Slot (module number) of the coupler
ERR | WORD | Error message
TYP | STRING(17) | Coupler type
MODEL | STRING(22) | Coupler model
OP_MODE | STRING(22) | Operation mode of the coupler
MAN_DATE | DATE | Manufacturing date of the coupler
DEV_NO | DWORD | Device number of the coupler
SER_NO | DWORD | Serial number of the coupler
FW_NAME | STRING(17) | Designation of the coupler firmware
FW_VER | STRING(17) | Version of the coupler firmware

Description
The block PROFI_INFO is always active. It reads the slot number at the block input CONO and outputs the corresponding information about the selected coupler at the outputs.

PROFI_INFO recognizes different coupler types. For that reason, the block outputs do not only relate to the PROFIBUS coupler.

The block is not intended for usage in normal user programs. It should be used to support fault diagnosis and maintenance operations.

CONO | BYTE
At input CONO the slot (module number) of the coupler is applied for which the information should be polled. The module number depends on the PLC category.
R012X | CONO = 1
R01X2 | CONO = 2

Note: Up to version V4.2 (incl.) the data type of CONO was INT.
An error identification is output at output ERR, if an invalid value is applied at input CONO. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained at the beginning of these descriptions.

Note: Up to version V4.2 (incl.) the data type of ERR was INT.

The output TYP displays the coupler type (i.e. the coupler design). Different types are recognized by the block.

For the PROFIBUS coupler used in 07 KT 9X controllers the output TYP is 'dual mode' (two operation modes). For the coupler used in 07 SL 97 controllers the output TYP is 'single mode' (one operation mode, only master).

If a coupler type can not be determined the output TYP is set to 'unknown type'.

The output MODEL displays the transfer protocols supported by the coupler as well as the subscriber type.

For the PROFIBUS coupler used in 07 KT 9X controllers the model identifier is MODEL = 'PROFIBUS Master/Slave'. For the coupler used in 07 SL 97 controllers the model identifier is MODEL = 'PROFIBUS DP Master'.

If the block detects an unknown model, the output MODEL displays 'unknown model'.

The output OP_MODE displays the current setting for the coupler operation mode. This output is only applicable for couplers with switchable operation modes.

For the PROFIBUS coupler the following operation modes are defined:

OP_MODE = 'none': The coupler has not yet received any configuration data. Hence, no operation mode is set.

OP_MODE = 'PROFIBUS-DP Master' The coupler was configured as a PROFIBUS DP master via the 907 AC 1131 control configuration or via the 907 FB 1131.

OP_MODE = 'PROFIBUS-DP Slave' The coupler was configured as a PROFIBUS DP slave via the 907 AC 1131 control configuration or via the 907 FB 1131.

The output MAN_DATE displays the manufacturing date of the coupler. The date is a variable of the data type DATE and has the format D#YYYY-MM-DD. The initial value is D#2000-01-01.

The output DEV_NO outputs the device number of the coupler.

The output SER_NO outputs the serial number of the coupler.

At the output FW_NAME the designation of the coupler firmware is applied. The firmware designation of the PROFIBUS coupler with switchable operation mode is e.g. 'DPMDPS C04DPABB'.

The output FW_VER displays the firmware release number (version) and the firmware issue date of the coupler. These data are represented as a string (e.g. 'V1.000 15.10.99').
Function call in IL

CAL INFO(CONO := INFO_CONO)

LD INFO.ERR
ST INFO.ERR
LD INFO.TYP
ST INFO.TYP
LD INFO.MODEL
ST INFO.MODEL
LD INFO.OP_MODE
ST INFO.OP_MODE
LD INFO.MAN_DATE
ST INFO.MAN_DATE
LD INFO.DEV_NO
ST INFO.DEV_NO
LD INFO.SER_NO
ST INFO.SER_NO
LD INFO.FW_NAME
ST INFO.FW_NAME
LD INFO.FW_VER
ST INFO.FW_VER

Note: In IL, the function call has to be performed in one line.

Function call in ST

INFO(CONO := INFO_CONO);

INFO.ERR := INFO.ERR;
INFO.TYP := INFO.TYP;
INFO.MODEL := INFO.MODEL;
INFO.OP_MODE := INFO.OP_MODE;
INFO.MAN_DATE := INFO.MAN_DATE;
INFO.DEV_NO := INFO.DEV_NO;
INFO.SER_NO := INFO.SER_NO;
INFO.FW_NAME := INFO.FW_NAME;
INFO.FW_VER := INFO.FW_VER;
The block DPM_SLVDIAG reads the detailed diagnostic data of a DP slave.

Block data

Available as of PLC runtime system: V4.0
Included in library:
- PROFIBUS_S90_V41.LIB
- PROFIBUS_S90_V42.LIB
- PROFIBUS_Master_S90_V43.LIB

Remark:
- CONO, SLV, ERR, MSTR
- EXT_DIAG_DAT - type changed

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DPM_SLVDIAG</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>SLV</td>
<td>BYTE</td>
<td>Bus address of the specific DP slave</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>STAT_1</td>
<td>STATIONSTATUS_1_TYPE</td>
<td>Stationstatus_1 according to standard</td>
</tr>
<tr>
<td>STAT_2</td>
<td>STATIONSTATUS_2_TYPE</td>
<td>Stationstatus_2 according to standard</td>
</tr>
<tr>
<td>STAT_3</td>
<td>STATIONSTATUS_3_TYPE</td>
<td>Stationstatus_3 according to standard</td>
</tr>
<tr>
<td>MSTR</td>
<td>BYTE</td>
<td>Bus address of the related DP master</td>
</tr>
<tr>
<td>EXT_DIAG_LEN</td>
<td>BYTE</td>
<td>Length of the extended diagnostic data</td>
</tr>
<tr>
<td>EXT_DIAG_DAT</td>
<td>ARRAY [1..238] OF BYTE</td>
<td>Extended diagnostic data according to standard</td>
</tr>
</tbody>
</table>

Description

The block DPM_SLVDIAG implements the PROFIBUS function Slave_Diag.

Every time a FALSE → TRUE edge is applied to the input EN, DPM_SLVDIAG reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until the processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.
POLLING DETAILED DIAGNOSTIC DATA OF A DP SLAVE

CONO BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R012X CONO = 1
R01X2 CONO = 2

Note: Up to version V4.2 (incl.) the data type of CONO was INT.

SLV BYTE
At input SLV the bus address of the DP slave is applied, for which diagnostic data are requested.

Note: Up to version V4.2 (incl.) the data type of SLV was INT.

DONE BOOL
The output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing is finished, DONE is set to TRUE.

The values applied at the block outputs are only valid if DONE = TRUE. Additionally, the diagnostic data are only valid if ERR = 0. If ERR is not 0, an error occurred.

ERR WORD
At the output ERR an error identifier is applied if an invalid value was applied to an input or if an error occurred during the request processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained at the beginning of this description.

Note: Up to version V4.2 (incl.) the data type of ERR was INT.

STAT_1 STATIONSTATUS_1_TYPE
At output STAT_1 the first octet of the DP slave diagnostic data is output. STAT_1 is only valid if DONE = TRUE and ERR = 0.

The structure of the type STATIONSTATUS_1_TYPE corresponds to the octet Stationstatus_1 defined in the standard and is described below.

STAT_2 STATIONSTATUS_2_TYPE
At output STAT_2 the second octet of the DP slave diagnostic data is output. STAT_2 is only valid if DONE = TRUE and ERR = 0.

The structure of the type STATIONSTATUS_2_TYPE corresponds to the octet Stationstatus_2 defined in the standard and is described below.

STAT_3 STATIONSTATUS_3_TYPE
At output STAT_3 the third octet of the DP slave diagnostic data is output. STAT_3 is only valid if DONE = TRUE and ERR = 0.

The structure of the type STATIONSTATUS_3_TYPE corresponds to the octet Stationstatus_3 defined in the standard and is described below.

MSTR BYTE
At the output MSTR the bus address of the DP master is applied, to which the DP slave is assigned. MSTR is only valid if DONE = TRUE and ERR = 0.

Using the block DPM_SLVDIAG, diagnostic data for all DP slaves can be polled which are connected to the bus. If the diagnostic data are requested by a DP slave which was assigned to the PLC during the configuration, the PLC bus address is output at MSTR. In multi-master systems it is possible that diagnostic data are also requested by DP slaves which are assigned to other DP masters. In this case, MSTR outputs the bus address of the DP master to which the requesting DP slave is assigned.

Note: Up to version V4.2 (incl.) the data type of MSTR was INT.

EXT_DIAG_LEN BYTE
At the output EXT_DIAG_LEN the number of valid bytes, following in EXT_DIAG_DAT are output. If EXT_DIAG_LEN = 0, no extended diagnostic data are available. Otherwise, EXT_DIAG_DAT[1] to EXT_DIAG_DAT[EXT_DIAG_LEN] contain the extended diagnostic data reported by the DP slave. EXT_DIAG_LEN is only valid if DONE = TRUE and ERR = 0.

Note: Up to version V4.2 (incl.) the data type of EXT_DIAG_LEN was INT.
EXT_DIAG_DAT  ARRAY [1..238] OF BYTE

At the output EXT_DIAG_DAT the extended diagnostic data which were reported from the DP slave are applied as a byte array. The data in EXT_DIAG_DAT are only valid if DONE = TRUE, ERR = 0 and EXT_DIAG_LEN > 0.

The extended diagnostic data are structured according to the standard. Since the meaning of these data strongly depends on the used devices, no automatic interpretation by the block is possible at this point. The evaluation of the data must be performed by the user with the aid of the particular device description and the GSD file respectively. The generally applicable structure of the extended diagnostic data (according to the standard) is described below.

---

**Function call in IL**

```plaintext
CAL  DIAG
    (EN := DIAG_EN,
     CONO := DIAG_CONO,
     SLV  := DIAG_SLV)

LD  DIAG.Done
ST  DIAG.Done
LD  DIAG.Err
ST  DIAG.Err
LD  DIAG.Stat_1
ST  DIAG.Stat_1
LD  DIAG.Stat_2
ST  DIAG.Stat_2
LD  DIAG.Stat_3
ST  DIAG.Stat_3
LD  DIAG.Mstr
ST  DIAG.Mstr
LD  DIAG.Ext_Diag.Len
ST  DIAG.Ext_Diag.Len
LD  DIAG.Ext_Diag.Dat
ST  DIAG_EXT_DIAG_DAT

Note: In IL, the function call has to be performed in one line.
```

**Function call in ST**

```plaintext
DIAG
    EN := DIAG_EN,
    CONO := DIAG_CONO,
    SLV  := DIAG_SLV);

DIAG.Done := DIAG.Done;
DIAG.Err := DIAG.Err;
DIAG.Stat_1 := DIAG.Stat_1;
DIAG.Stat_2 := DIAG.Stat_2;
DIAG.Stat_3 := DIAG.Stat_3;
DIAG.Mstr := DIAG.Mstr;
DIAG.Ext_Diag.Len := DIAG.Ext_Diag.Len;
DIAG_EXT_DIAG_DAT := DIAG_EXT_DIAG_DAT;
```

---

**Structure of DP slave diagnostic data**

The block DPM_SLVDIAG divides the diagnostic data of a DP slave into sections and applies them to the corresponding outputs. The structure of the DP slave diagnostic data is prescribed by the standard as follows:

<table>
<thead>
<tr>
<th>Octet</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stationstatus_1</td>
<td>STAT_1</td>
</tr>
<tr>
<td>2</td>
<td>Stationstatus_2</td>
<td>STAT_2</td>
</tr>
<tr>
<td>3</td>
<td>Stationstatus_3</td>
<td>STAT_3</td>
</tr>
<tr>
<td>4</td>
<td>Master_Add</td>
<td>MSTR</td>
</tr>
<tr>
<td>5 to 6</td>
<td>Ident_Number</td>
<td></td>
</tr>
<tr>
<td>7 to 244</td>
<td>Ext_Diag_Data</td>
<td>EXT_DIAG_DAT[1] to EXT_DIAG_DAT[EXT_DIAG_LEN]</td>
</tr>
</tbody>
</table>
The diagnostic byte Stationstatus_1 (defined within the standard) is applied to the output STAT_1 of the block DPM_SLVDIAG. It is represented as a structure of the data type STATIONSTATUS_1_TYPE. Within the PROFIBUS master library the structure STATIONSTATUS_1_TYPE is declared as follows:

```c
TYPE STATIONSTATUS_1_TYPE:
    STRUCT
        NON_EXISTENT: BOOL;
        NOT_READY: BOOL;
        CFG_FAULT: BOOL;
        EXT_DIAG: BOOL;
        NOT_SUPPORTED: BOOL;
        INVALID_RESPONSE: BOOL;
        PRM_FAULT: BOOL;
        MASTER_LOCK: BOOL;
    END_STRUCT
END_TYPE
```

### NON_EXISTENT
This bit is set, the PROFIBUS coupler has not found a DP slave at the bus address which was applied at the block input SLV.

### NOT_READY
This bit is set to TRUE by the DP slave, if the DP slave is not ready for I/O data exchange.

### CFG_FAULT
This bit is set to TRUE by the DP slave, if the configuration data (nominal configuration) received from the DP master do not match the data stored in the DP slave (actual configuration).

### EXT_DIAG
If this bit is set to TRUE, extended diagnostic data are available in EXT_DIAG_DAT. If this bit is not set (FALSE) and EXT_DIAG_LEN > 0, possibly a status message is available in EXT_DIAG_DAT. The meaning of such a status message is device-dependent.

### NOT_SUPPORTED
This bit is set to TRUE by the DP slave, if an unsupported function was requested before.

### INVALID_RESPONSE
If this bit is set to TRUE, the PROFIBUS coupler has not received a plausible response from the requested DP slave.

### PRM_FAULT
The DP slave sets this bit to TRUE, if the parameter data received last were faulty.

### MASTER_LOCK
This bit is set to TRUE, if the DP slave is assigned to another DP master. In this case MSTR contains the bus address of this DP master.
### Stationstatus_2 STAT_2 STATIONSTATUS_2_TYPE

The diagnostic byte Stationstatus_2 (defined within the standard) is applied to the output STAT_2 of the block DPM_SLVDIAG. It is represented as a structure of the data type STATIONSTATUS_2_TYPE. Within the PROFINET master library the structure STATIONSTATUS_2_TYPE is declared as follows:

```c
TYPE STATIONSTATUS_2_TYPE:
  STRUCT
    PRM_REQ: BOOL;
    STAT_DIAG: BOOL;
    DP_Slave: BOOL;
    WD_ON: BOOL;
    FREEZE_MODE: BOOL;
    SYNC_MODE: BOOL;
    reserved: BOOL;
    DEACTIVATED: BOOL;
  END_STRUCT
END_TYPE
```

- **DP slave**
  - **BOOL**
  - This bit is permanently set to TRUE.

- **WD_ON**
  - **BOOL**
  - This bit is set to TRUE by the DP slave, if the slave response monitoring is active.

- **FREEZE_MODE**
  - **BOOL**
  - This bit is set to TRUE by the DP slave, if the slave is currently running in Freeze mode.

- **SYNC_MODE**
  - **BOOL**
  - This bit is set to TRUE by the DP slave, if the slave is currently running in Sync mode.

- **reserved**
  - **BOOL**
  - This bit is reserved and currently not used.

- **DEACTIVATED**
  - **BOOL**
  - This bit is set to TRUE, if the DP slave is not marked as active in the DP master’s configuration data and has been taken out of the cyclic I/O data exchange (please refer to 907 AC 1131 and 907 FB 1131, 'Slave active in current configuration').

- **PRM_REQ**
  - **BOOL**
  - This bit is set to TRUE by the DP slave, if it is required to re-parameterize and re-configure the slave (e.g. when adding an additional I/O module). The bit remains set until the re-parameterization is done.

- **STAT_DIAG**
  - **BOOL**
  - This bit is set to TRUE by the DP slave, if the slave has a static diagnosis. A DP slave with static diagnosis is not ready for I/O data exchange.
The diagnostic byte Stationstatus_3 (defined within the standard) is applied to the output STAT_3 of the block DPM_SLVDIAG. It is represented as a structure of the data type STATIONSTATUS_3_TYPE. Within the PROFIBUS master library the structure STATIONSTATUS_3_TYPE is declared as follows:

```plaintext
TYPE STATIONSTATUS_3_TYPE:
STRUCT
  reserved0 : BOOL;
  reserved1 : BOOL;
  reserved2 : BOOL;
  reserved3 : BOOL;
  reserved4 : BOOL;
  reserved5 : BOOL;
  reserved6 : BOOL;
  EXT_DIAG_OVERFLOW : BOOL;
END_STRUCT
END_TYPE
```

- **reserved0 - reserved6** (BOOL)
  These bits are reserved and currently not used.

- **EXT_DIAG_OVERFLOW** (BOOL)
  If this bit is set to TRUE, more diagnostic data are available as entered in EXT_DIAG_DAT. For example, the DP slave sets this bit, if currently more diagnostic messages are queued, than it is able to transfer within one telegram (see entry Max_Diag_Data_Len in the GSD file).

**Master_Add**

Into this octet the DP slave enters the address of the DP master from which it was parameterized (i.e. to which it is assigned). If the DP slave was not yet parameterized by any DP master, the following is valid: MSTR = 255.

**Ident_Number**

At this point the DP slave enters its identification number in the diagnosis telegram. The identification number is assigned by the PNO (PROFIBUS Nutzerorganisation e.V. = PROFIBUS user organization) for each device type. This number is a firm component of the GSD file. The block DPM_SLVDIAG does not output the device identification number, because the number is not necessary for evaluating the diagnostic data.
The above described six bytes of standard diagnostic data have to be supported by each DP slave. Optionally, a DP slave can additionally provide extended diagnostic data. This is the case if a value greater than 6 is assigned to the item Max_Diag_Data_Len in the GSD file of the DP slave. The format of the extended diagnosis is determined in the standard. Since the extended diagnostic data are not static on the one side and can contain manufacturer-specific entries on the other side, no automatic data evaluation can be performed by the block DM_SLVDIAG.

The evaluation of extended diagnostic data must be performed by the user with the aid of the GSD file for the particular DP slave and the description of the generally applicable data format given below.

The extended diagnostic data are divided into three ranges:
- device-related diagnosis
- module-related diagnosis
- channel-related diagnosis

Device-related diagnosis
The device-related diagnosis contains general diagnostic information, such as overtemperature or undervoltage. This range is initiated by a header byte, in which the highest two bits are permanently set to 00. The lower six bits indicate the length of the following block, including the header byte itself.

<table>
<thead>
<tr>
<th>Device-related range</th>
<th>Length</th>
<th>Header</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0010</td>
<td>00</td>
<td>10</td>
</tr>
</tbody>
</table>

The device-related diagnostic data are defined by the manufacturer. For detailed information about their meaning please refer to the particular device documentation.

Module-related diagnosis
The module-related diagnosis contains diagnostic information which can be assigned directly to the particular I/O modules of the device. The range containing the module-related diagnosis starts with a header byte, in which the highest two bits are permanently set to 01. The lower six bits indicate the length of the following block, including the header byte itself.

<table>
<thead>
<tr>
<th>Module-related range</th>
<th>Length</th>
<th>Header</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0110</td>
<td>01</td>
<td>20</td>
</tr>
</tbody>
</table>

In the following block one single bit is assigned to each module. The module index is represented by the bit offset within the block (please refer to the example). If a bit is set to TRUE, this means that a diagnosis has to be done in the related I/O module.

Channel-related diagnosis
In the channel-related diagnosis range, the diagnosed channels and the cause for the diagnosis are entered one after the other. Each entry consists of three bytes and starts with a header byte, in which the highest two bits are permanently set to 10. The lower six bits contain the index of the module, for which the following diagnosis is performed.

<table>
<thead>
<tr>
<th>Channel-related range</th>
<th>Module</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>012100</td>
</tr>
</tbody>
</table>

The lower six bits of the following byte contain the number of the channel which reports a diagnosis. The highest two bits indicate whether the specific channel is an input channel, an output channel or an I/O channel.

<table>
<thead>
<tr>
<th>Channel</th>
<th>Direction</th>
<th>Module</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Reserved</td>
<td>00</td>
<td>102100</td>
</tr>
<tr>
<td>01</td>
<td>Input</td>
<td>01</td>
<td>012100</td>
</tr>
<tr>
<td>10</td>
<td>Output</td>
<td>10</td>
<td>112100</td>
</tr>
<tr>
<td>11</td>
<td>Input/output</td>
<td>11</td>
<td>112100</td>
</tr>
</tbody>
</table>

The direction identifier in the bits 6 and 7 is encoded as follows:

<table>
<thead>
<tr>
<th>Direction</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>Input</td>
<td>01</td>
</tr>
<tr>
<td>Output</td>
<td>10</td>
</tr>
<tr>
<td>Input/output</td>
<td>11</td>
</tr>
</tbody>
</table>

The third byte of each entry contains the channel type in its upper three bits and the error type in the lower five bits.

<table>
<thead>
<tr>
<th>Channel type</th>
<th>Error type</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>110</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
<td>102100</td>
</tr>
</tbody>
</table>

The channel type (channel width) is encoded as follows:

<table>
<thead>
<tr>
<th>Channel type</th>
<th>Error type</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>110</td>
<td>000</td>
<td>102100</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
<td>102100</td>
</tr>
</tbody>
</table>

The channel type (channel width) is encoded as follows:
The encoding of the error type is as follows:

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Binary</th>
<th>Decoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 reserved</td>
<td>0 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 short circuit</td>
<td>0 0 0 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>2 undervoltage</td>
<td>0 0 0 1 0 0</td>
<td>1</td>
</tr>
<tr>
<td>3 overvoltage</td>
<td>0 0 0 1 1 0</td>
<td>3</td>
</tr>
<tr>
<td>4 overload</td>
<td>0 0 1 0 0 0</td>
<td>4</td>
</tr>
<tr>
<td>5 overtemperature</td>
<td>0 0 1 0 1 0</td>
<td>5</td>
</tr>
<tr>
<td>6 cable break</td>
<td>0 0 1 1 0 0</td>
<td>6</td>
</tr>
<tr>
<td>7 upper limit</td>
<td>0 0 1 1 1 0</td>
<td>7</td>
</tr>
<tr>
<td>exceeded</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 lower limit</td>
<td>0 1 0 0 0 0</td>
<td>8</td>
</tr>
<tr>
<td>exceeded</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 Error</td>
<td>0 1 0 0 1 0</td>
<td>9</td>
</tr>
<tr>
<td>10 reserved</td>
<td>0 1 0 1 0 0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 reserved</td>
<td>0 1 1 1 1 1</td>
<td>15</td>
</tr>
<tr>
<td>16 manufacturer-</td>
<td>1 0 0 0 0 0</td>
<td>16</td>
</tr>
<tr>
<td>specific</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 manufacturer-</td>
<td>1 1 1 1 1 1</td>
<td>31</td>
</tr>
<tr>
<td>specific</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The valid length of the complete extended diagnostic data is indicated at output EXT_DIAG_LEN of the block DPM_SLV_DIAG. When evaluating the diagnosis only data have to be considered which are contained in the range EXT_DIAG_DAT[1] to EXT_DIAG_DAT[EXT_DIAG_LEN].
Example for extended diagnostic data

EXT_DIAG_LEN = 15

| EXT_DIAG_DAT[1] | 0 0 0 0 0 1 0 0 | Device-related diagnosis Length: 4 bytes incl. header byte |
| EXT_DIAG_DAT[2] | X X X X X X X X | Device-related diagnosis Length: 3 bytes |
| EXT_DIAG_DAT[3] | X X X X X X X X | Meaning of the data is manufacturer-specific |
| EXT_DIAG_DAT[4] | X X X X X X X X | |
| EXT_DIAG_DAT[5] | 0 1 0 0 0 1 0 1 | ID-related diagnosis Length: 5 bytes incl. header byte |
| EXT_DIAG_DAT[6] | 0 0 0 0 0 1 0 0 | Module 0 with diagnosis |
| EXT_DIAG_DAT[7] | 0 0 1 0 0 1 0 0 | Overload, channel organized bit-wise |
| EXT_DIAG_DAT[8] | 0 0 1 0 0 1 0 0 | Channel-related diagnosis Module 0 |
| EXT_DIAG_DAT[9] | 0 0 1 0 0 1 0 0 | Channel-related diagnosis Module 12 |
| EXT_DIAG_DAT[10] | 1 0 0 0 0 0 0 0 | Overload, channel organized bit-wise |
| EXT_DIAG_DAT[11] | 0 0 0 0 0 1 0 | Module 12 |
| EXT_DIAG_DAT[12] | 0 0 0 0 0 1 0 0 | Channel-related diagnosis Module 12 |
| EXT_DIAG_DAT[13] | 1 0 0 0 1 1 0 1 | Upper limit exceeded, channel organized word-wise |

Example program for evaluating extended diagnostic data

A detailed example program for the evaluation of extended diagnostic data can be found on the CD-ROM for 907 AC 1131 and in the programming system online help.
READING OUT THE PROFIBUS COUPLER STATUS

DPM_STAT outputs the status of a PROFIBUS coupler in operation mode DP master. At the outputs, information about the communication state and the error events are output.

Block data
Available as of PLC runtime system: V4.0
Included in library:
- PROF40.LIB
- PROFIBUS_S90_V41.LIB
- PROFIBUS_S90_V42.LIB
- PROFIBUS_Master_S90_V43.LIB

Remark:
- CONO, ERR, DPM_STATE, COM_ERR, BUS_ERR, TIME_OUT - type changed

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DPM_STAT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>STAT_BITS</td>
<td>DPM_STATE_BITS_TYPE</td>
<td>Atypical communication states</td>
</tr>
<tr>
<td>DPM_STATE</td>
<td>WORD</td>
<td>DP master state according to standard</td>
</tr>
<tr>
<td>COM_ERR</td>
<td>DPM_COM_ERR_TYPE</td>
<td>Communication error</td>
</tr>
<tr>
<td>BUS_ERR</td>
<td>WORD</td>
<td>Number of serious bus errors</td>
</tr>
<tr>
<td>TIME_OUT</td>
<td>WORD</td>
<td>Number of timeouts</td>
</tr>
</tbody>
</table>

Description
The block DPM_STAT outputs the current state of a PROFIBUS coupler in operation mode DP master. DPM_STAT is active if input EN = TRUE. If the block is active, the current values are always displayed at the outputs.

EN
The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are available at the outputs.

CONO
At input CONO the slot (module number) of the coupler is applied, for which the state should be read. The module number depends on the PLC category.

ERR
An error identification is output at output ERR, if an invalid value is applied at a block input. ERR has always to be considered together with the input EN. The value applied at ERR is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The error messages encoding at the output ERR applies to all PROFIBUS blocks and is explained at the beginning of this description.

Note: Up to version V4.2 (incl.) the data type of ERR was INT.
STATE_BITS   DPM_STATE_BITS_TYPE
At output STATE_BITS atypical communication states at the PROFIBUS are displayed. STAT_BITS is only valid if EN = TRUE and ERR = 0.

The structure of the type DPM_STATE_BITS_TYPE is defined in the PROFIBUS master library and is described below.

DPM_STATE   BYTE
At output DPM_STATE, the DP master state is output according to the standard. The following states are defined:

OFFLINE 00_hex / 00_dec
STOP 40_hex / 64_dec
CLEAR 80_hex / 128_dec
OPERATE C0_hex / 192_dec

DPM_STATE = OFFLINE
The PROFIBUS coupler is in the initialization state if DPM_STATE has the value OFFLINE. After the initialization phase is completed the coupler changes to STOP state.

DPM_STATE = STOP
The coupler is completely initialized, if DPM_STATE has the value STOP. In this state the coupler is ready to receive configuration data. No data exchange with the DP slaves is performed. The coupler has this state if no user program is running.

DPM_STATE = CLEAR
If the user program is started, the coupler changes from STOP state into CLEAR state and begins, via the bus, to parameterize (set into operation) the DP slaves which were assigned during the configuration. After the setup has been completed successfully, the coupler changes into OPERATE state. If an error occurs during the parameter setting, the coupler changes back to STOP state.

DPM_STATE = OPERATE
Normally the coupler is in OPERATE state while a user program is running. In this state the DP master exchanges I/O data with the DP slaves. If an error occurs during this process and ‘Auto Clear Mode’ was selected during the configuration, the coupler changes back to CLEAR state and tries again to parameterize the DP slaves. If ‘Auto Clear Mode’ was not selected, the coupler remains (in case of an error) in OPERATE state. If the user program is stopped, the coupler also changes back to STOP state.

DPM_STATE is only valid if EN = TRUE and ERR = 0.

Note: Up to version V4.2 (incl.) the data type of DPM_STATE was INT.

COM_ERR   DPM_COM_ERR_TYPE
At output COM_ERR possible communication errors are displayed. COM_ERR is only valid if EN = TRUE and ERR = 0.

The structure of the type DPM_COM_ERR_TYPE is defined in the PROFIBUS master library and is described below together with the possible errors.

Note: Up to version V4.2 (incl.) the data type of COM_ERR was COM_ERR_TYPE.

BUS_ERR   WORD
At output BUS_ERR the number of serious bus errors, e.g. short circuits on the transmission line which occurred since the system start are output. BUS_ERR is only valid if EN = TRUE and ERR = 0.

Note: Up to version V4.2 (incl.) the data type of BUS_ERR was INT.

TIME_OUT   WORD
At output TIME_OUT the number of timeout errors which occurred since the system start are output. A timeout error occurs, if a DP slave does not response to a DP master request within the configured time. TIME_OUT is only valid if EN = TRUE and ERR = 0.

Note: Up to version V4.2 (incl.) the data type of TIME_OUT was INT.
Function call in IL

CAL STAT
(EN := STAT_EN,
CONO := STAT_CONO)

LD STAT.ERR
ST STAT.ERR
LD STAT.STATE_BITS
ST STAT.STATE_BITS
LD STAT.DPM_STATE
ST STAT.DPM_STATE
LD STAT.COM_ERR
ST STAT.COM_ERR
LD STAT.BUS_Err
ST STAT.BUS_Err
LD STAT.TIME_OUT
ST STAT.TIME_OUT

Function call in ST

STAT
(EN := STAT_EN,
CONO := STAT_CONO);

STAT.ERR := STAT.ERR;
STAT.STATE_BITS := STAT.STATE_BITS;
STAT.DPM_STATE := STAT.DPM_STATE;
STAT.COM_ERR := STAT.COM_ERR;
STAT.BUS_Err := STAT.BUS_Err;
STAT.TIME_OUT := STAT.TIME_OUT;

Note: In IL, the function call has to be performed in
one line.
### STATE_BITS DPM_STATE_BITS_TYPE

The structure STATE_BITS includes four boolean variables which display different communication states. Within the PROFIBUS master library the data type DPM_STATE_BITS_TYPE is declared as follows:

```plaintext
TYPE DPM_STATE_BITS_TYPE:
  STRUCT
    CTRL: BOOL;
    AUTO_CLR: BOOL;
    NO_EXCH: BOOL;
    FATAL: BOOL;
  END_STRUCT
END_TYPE
```

- **CTRL** BOOL
  If this bit is set to TRUE, a parameter setting error occurred. In normal operation CTRL should be FALSE. If not, the parameter and configuration data have to be checked.

- **AUTO_CLR** BOOL
  This bit is only valid if ‘Auto Clear Mode’ was selected during the configuration. If AUTO_CLR is set to TRUE, an error occurred during the communication with at least one DP slave. As a result, the coupler stopped the data exchange with all DP slaves and changed back to CLEAR state (see DPM_STATE).

- **NO_EXCH** BOOL
  This bit is set to TRUE, if the process data exchange with one or several DP slaves is not possible. The error cause can be found in the configuration data as well as in the DP slaves themselves.

- **FATAL** BOOL
  If FATAL is set to TRUE, no communication via the PROFIBUS is possible due to a serious bus error (e.g. bus line short circuit). In this case, all bus lines have to be checked.
COM_ERR  DPM_COM_ERR_TYPE

Communication errors can be located precisely via COM_ERR. The output COM_ERR is represented as a structure of the type DPM_COM_ERR_TYPE. Within the PROFIBUS master library this data type is declared as follows:

```
TYPE DPM_COM_ERR_TYPE:
  STRUCT
    ADDRESS: BYTE;
    EVENT: BYTE;
  END_STRUCT
END_TYPE
```

ADDRESS = 255  Coupler error

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>USR_INTF task not found</td>
<td>Coupler</td>
<td>Contact ABB</td>
</tr>
<tr>
<td>51</td>
<td>No global data field</td>
<td>Coupler</td>
<td>Contact ABB</td>
</tr>
<tr>
<td>52</td>
<td>FDL task not found</td>
<td>Coupler</td>
<td>Contact ABB</td>
</tr>
<tr>
<td>53</td>
<td>PLC task not found</td>
<td>Coupler</td>
<td>Contact ABB</td>
</tr>
<tr>
<td>54</td>
<td>No master parameter record</td>
<td>Configuration</td>
<td>Generate configuration in the project and reload program to controller</td>
</tr>
<tr>
<td>55</td>
<td>Faulty value in master parameter record</td>
<td>Configuration</td>
<td>Check configuration data for coupler in the project and/or reload program to controller</td>
</tr>
<tr>
<td>56</td>
<td>No slave parameter records</td>
<td>Configuration</td>
<td>Add DP slaves to configuration data and reload program to controller</td>
</tr>
<tr>
<td>57</td>
<td>Faulty value in a slave parameter record</td>
<td>Configuration</td>
<td>Check configuration data of subordinate DP slaves in the project and/or reload program to controller</td>
</tr>
<tr>
<td>58</td>
<td>Doubled slave address</td>
<td>Configuration</td>
<td>Check configuration data of subordinate DP slaves in the project for doubled bus addresses and/or reload program to controller</td>
</tr>
<tr>
<td>59</td>
<td>Invalid offset address output data</td>
<td>Configuration</td>
<td>Check configuration data of subordinate DP slaves in the project for invalid IEC addresses and/or reload program to controller</td>
</tr>
<tr>
<td>60</td>
<td>Invalid offset address input data</td>
<td>Configuration</td>
<td>Check configuration data of subordinate DP slaves in the project for invalid IEC addresses and/or reload program to controller</td>
</tr>
<tr>
<td>61</td>
<td>Range overlapping in output data</td>
<td>Configuration</td>
<td>Check configuration data of subordinate DP slaves in the project for overlapping IEC address ranges and/or reload program to controller</td>
</tr>
<tr>
<td>62</td>
<td>Range overlapping in input data</td>
<td>Configuration</td>
<td>Check configuration data of subordinate DP slaves in the project for overlapping IEC address ranges and/or reload program to controller</td>
</tr>
<tr>
<td>63</td>
<td>Unknown process data handshake</td>
<td>Control system</td>
<td>Supply voltage OFF/ON, otherwise contact ABB</td>
</tr>
<tr>
<td>64</td>
<td>Insufficient memory</td>
<td>Coupler</td>
<td>Contact ABB</td>
</tr>
<tr>
<td>65</td>
<td>Faulty slave parameter record</td>
<td>Configuration</td>
<td>Check configuration data of subordinate DP slaves in the project and/or reload program to controller</td>
</tr>
<tr>
<td>202</td>
<td>No segment available</td>
<td>Coupler</td>
<td>Contact ABB</td>
</tr>
<tr>
<td>212</td>
<td>Error reading database</td>
<td>Configuration/Coupler</td>
<td>Reload program with configuration data to controller</td>
</tr>
</tbody>
</table>

ADDRESS  BYTE

In case of an error, ADDRESS contains the bus address of the faulty subscriber (0 to 255). If ADDRESS has the value 255, the error is located in the coupler itself.

EVENT  BYTE

EVENT displays the cause of an error. The following tables show the encoding of the various errors.
### ADDRESS = 0..125  Error at subscriber with bus address ADDRESS

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Subscriber reports overflow</td>
<td>DP master</td>
<td>Check configuration data of subordinate DP slave in the project and/or reload program to controller</td>
</tr>
<tr>
<td></td>
<td></td>
<td>telegram</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Subscriber does not support requested function</td>
<td>DP master</td>
<td>Check DP slave for conformity according to PROFIBUS standard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>telegram</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>No data in response telegram</td>
<td>DP slave</td>
<td>Compare configuration data of subordinate DP slave in the project with actual configuration and reload program to controller if necessary</td>
</tr>
<tr>
<td>17</td>
<td>Subscriber does not response</td>
<td>DP slave</td>
<td>Check bus line and DP slave bus address</td>
</tr>
<tr>
<td>18</td>
<td>DP master not in logical token ring</td>
<td>DP master</td>
<td>Check the configured DP master bus address, the highest station address (HSA) in the other system DP masters and/or bus line for short circuits</td>
</tr>
<tr>
<td>21</td>
<td>Faulty parameter in request telegram</td>
<td>DP master</td>
<td>Contact ABB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>telegram</td>
<td></td>
</tr>
</tbody>
</table>
**READING OUT THE STATUS SURVEY OF ALL DP SLAVES**

The block DPM_SYSDIAG outputs a bit field as a state survey of all DP slaves at output SLV. Three different surveys can be selected via input TYP.

### Block data

- **Available as of PLC runtime system:** V4.0
- **Included in library:**
  - PROFI40.LIB
  - PROFIBUS_S90_V41.LIB
  - PROFIBUS_S90_V42.LIB
  - PROFIBUS_Master_S90_V43.LIB
- **Remark:** CONO, TYP, ERR - type changed

### Block type

Function block without historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DPM_SYSDIAG</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>TYP</td>
<td>BYTE</td>
<td>Selection of the survey type</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>SLV</td>
<td>ARRAY [0..127] OF BOOL</td>
<td>DP slaves status survey</td>
</tr>
</tbody>
</table>

### Description

The block DPM_SYSDIAG outputs different status surveys of all DP slaves. Three survey types can be selected:

- configuration survey
- I/O data exchange survey
- diagnosis survey

**EN**

The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are available at the outputs.

**CONO**

At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

- R012X CONO = 1
- R01X2 CONO = 2

**Note:** Up to version V4.2 (incl.) the data type of CONO was INT.

**TYP**

The input TYP is used to select which status survey should be displayed at output SLV.

- TYP = 1 configuration survey
- TYP = 2 data exchange survey
- TYP = 3 diagnosis survey

**BYTE**

At the output SLV it is displayed which DP slaves the DP master has configured successfully (i.e. set into operation). Please note that the DP master sets only those DP slaves into operation which were assigned when generating the configuration data.

At output SLV is displayed with which DP slaves the DP master exchanges data. The data exchange can only be performed with DP slaves which were configured by the DP master itself. The data exchange survey can only be requested, if the coupler is in OPERATE state.

At output SLV it is displayed which DP slaves have signaled an available diagnosis. The diagnosis survey can only be requested if the coupler is in OPERATE state.
READING OUT THE STATUS SURVEY OF ALL
DP SLAVES

Note: Up to version V4.2 (incl.) the data type of TYP was INT.

ERR WORD
An error identification is applied at the output ERR if an invalid value is applied at an input. ERR has always to be considered together with the input EN. The value applied at ERR is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The error messages encoding at the output ERR applies to all PROFIBUS blocks and is explained at the beginning of this description.

Note: Up to version V4.2 (incl.) the data type of ERR was INT.

SLV ARRAY [0..127] OF BOOL
At output SLV, the status survey is output as a bit field. Every bit within this field represents a DP slave. With this, the index corresponds to the DP slave bus address. A bit which is set to TRUE means that the state selected via TYP is applicable for the corresponding DP slave.

If e.g. TYP = 1 is selected and SLV[2] = TRUE, the DP slave was successfully configured with bus address 2 by the DP master. If SLV[2] = FALSE, the configuration of the specific DP slave has not yet been completed or the DP slave is not part of the DP master configuration data.

If TYP = 2 was selected and SLV[2] = TRUE, this means that the DP master exchanges I/O data with the DP slave which has bus address 2. However, if SLV[2] = FALSE, the DP master currently does not exchange I/O data with the DP slave. The DP master is only able to exchange data with those DP slaves which it has successfully set into operation before.

For TYP = 3 e.g. SLV[2] = TRUE means that the DP slave with bus address 2 has signaled a diagnosis. The detailed diagnosis can be requested by using the block DPM_SLVDIAG.

The bit field output at SLV is only valid if EN = TRUE and ERR = 0.

Function call in IL
CAL SYSDIAG
  (EN := SYSDIAG_EN,
   CONO := SYSDIAG_CONO,
   TYP := SYSDIAG_TYP)
LD SYSDIAG.ERR
ST SYSDIAG.ERR
LD SYSDIAG.SLV
ST SYSDIAG.SLV

Note: In IL, the function call has to be performed in one line.

Function call in ST
SYSDIAG
  (EN := SYSDIAG_EN,
   CONO := SYSDIAG_CONO,
   TYP := SYSDIAG_TYP);
SYSDIAG_ERR := SYSDIAG.ERR;
SYSDIAG_SLV := SYSDIAG.SLV;
SENDING GLOBAL CONTROL COMMANDS TO THE DP SLAVE

The block DPM_CTRL sends Global Control commands for the time synchronization of process data of several DP slaves.

Block data
Available as of PLC runtime system: V4.0
Included in library: PROFIBUS_Master_S90_V43.LIB

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>DPM_CTRL</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>SLV</td>
<td>BYTE</td>
<td>Selection of the called DP slaves</td>
</tr>
<tr>
<td>GROUP_SEL</td>
<td>BYTE</td>
<td>Selection of the called slave groups</td>
</tr>
<tr>
<td>CLR_DATA</td>
<td>BOOL</td>
<td>Resetting the output data of the DP slaves</td>
</tr>
<tr>
<td>UNFREEZE</td>
<td>BOOL</td>
<td>Termination of the input data synchronization</td>
</tr>
<tr>
<td>FREEZE</td>
<td>BOOL</td>
<td>Synchronization of the input data</td>
</tr>
<tr>
<td>UNSYNC</td>
<td>BOOL</td>
<td>Termination of the output data synchronization</td>
</tr>
<tr>
<td>SYNC</td>
<td>BOOL</td>
<td>Synchronization of the output data</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

Description
The block DPM_CTRL implements the PROFIBUS function Global_Control. Global_Control is a broadcast function.

Using Global Control commands the output data of one, several or all slaves can be reset or the input or output data of the slaves can be synchronized. The commands are selected via combinations of the outputs CLR_DATA, FREEZE / UNFREEZE and SYNC / UNSYNC. The called slaves are selected using three parameters. First, during the project planning, the slaves can be divided into logical groups (refer to AC1131 Configurator and FB1131 documentation). Then the slaves can be specifically called individually or in groups via the block inputs SLV and GROUP_SLV during running operation. Every time a FALSE → TRUE edge is applied to input EN, DPM_CTRL reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

EN \(\rightarrow\) BOOL

If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE. While the request is processed, state changes at input EN are recognized but not evaluated.
SENDING GLOBAL CONTROL COMMANDS TO THE DP SLAVE

CONO BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R012X CONO = 1
R01X2 CONO = 2

SLV BYTE
At input SLV the bus address of the DP slave is applied, to which a Global Control command shall be sent. With SLV = 0..126, an individual slave with the corresponding bus address is directly called, independent of the group it was assigned during configuration and independent of the value applied at block input GROUP_SEL. If SLV = 127 and GROUP_SEL = 0, all slaves are called simultaneously. The selection of individual slave groups is done with SLV = 127 and a combination of GROUP_SEL and the group assignment during configuration. This is why the block outputs SLV and GROUP_SEL always have to be considered together with the group assignment performed during configuration. The possible combinations are explained following the description of the block inputs.

GROUP_SEL BYTE
At input GROUP_SEL the slave groups are selected, to which a Global Control command shall be sent. With SLV = 0..126, an individual slave with the corresponding bus address is directly called, independent of the group assignment during configuration and independent of the value applied at block input GROUP_SEL. If GROUP_SEL = 0 and SLV = 127, all slaves are called simultaneously. The selection of individual slave groups is done with SLV = 127 and a combination of GROUP_SEL and the group assignment during configuration. This is why the block outputs SLV and GROUP_SEL always have to be considered together with the group assignment performed during configuration. The possible combinations are explained following the description of the block inputs.

CLR_DATA BOOL
With the input CLR_DATA the output data of the on-site slaves can be reset. The called slave resets its outputs on-site to 0 when it receives a Global Control command with CLR_DATA = TRUE. If the CLR_DATA part of a command is FALSE, the outputs of the called slaves keep their present state. Slaves which are not called ignore the whole command. The possible combinations of individual commands are explained following the description of the block inputs.

UNFREEZE BOOL
When input UNFREEZE is TRUE the synchronization mode for the input data of the called slaves is terminated, regardless whether the input FREEZE is TRUE or FALSE at this moment. From now on, the called slaves forward their input values immediately to the master again. If the UNFREEZE part of a command is FALSE, the called slaves keep their present state. The UNFREEZE command is ignored if the called slave is not in the FREEZE state. Slaves which are not called ignore the whole command. UNFREEZE has always to be considered together with FREEZE. The possible combinations of individual commands are explained following the description of the block inputs.

FREEZE BOOL
When input FREEZE is TRUE and at the same time input UNFREEZE is FALSE, the called slaves are operated in the input data synchronization mode. This mode is activated with the first FREEZE command. As a result, the called slaves simultaneously freeze the present values applied at their local inputs and temporarily store them. During the subsequent process data cycles the temporarily stored input values are transmitted to the master, regardless whether the actual input values have changed in the mean time or not. When another FREEZE command is received, the temporarily stored input values are updated, i.e. the called slaves again simultaneously store the present input values into an internal buffer and then transmit these values to the master during the subsequent cycles. FREEZE has always to be considered together with UNFREEZE. The possible combinations of individual commands are explained following the description of the block inputs.

UNSYNC BOOL
When input UNSYNC is TRUE the synchronization mode for the output data of the called slaves is terminated, regardless whether the input SYNC is TRUE or FALSE at this moment. From now on, the called slaves forward the output data received from the master immediately to their own outputs again. If the UNSYNC part of a command is FALSE, the called slaves keep their present state. The UNSYNC command is ignored if the called slave is not in the SYNC state. Slaves which are not called ignore the whole command. UNSYNC has always to be considered together with SYNC. The possible combinations of individual commands are explained following the description of the block inputs.
SENDING GLOBAL CONTROL COMMANDS TO THE DP SLAVE

SYNC

When input SYNC is TRUE and at the same time input UNSYNC is FALSE, the called slaves are operated in the output data synchronization mode. This mode is activated with the first SYNC command. As a result, the called slaves freeze the present states of their local outputs. The output data sent during the subsequent process data cycles are first only locally stored in these slaves. Then, when another SYNC command is received, the slaves simultaneously apply these temporarily stored values to their outputs. SYNC has always to be considered together with UNSYNC. The possible combinations of individual commands are explained following the description of the block inputs.

DONE

The output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing is finished, DONE is set to TRUE. The values applied at the block outputs are only valid if DONE = TRUE. The Global Control command was only executed correctly if also ERR = 0. If ERR is not 0, an error occurred.

ERR

At the output ERR an error identifier is applied if an invalid value was applied to an input or if an error occurred during the request processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained following the block descriptions.

Function call in IL

CAL CTRL
(EN := CTRL_EN,
CONO := CTRL_CONO,
SLV := CTRL_SLV
GROUP_SEL :=CTRL_GROUP_SEL
CLR_DATA := TRL_CLR_DATA,
UNFREEZE := CTRL_UNFREEZE,
FREEZE := CTRL_FREEZE,
UNSYNC := CTRL_UNSSYNC,
SYNC := CTRL_SYNC)

LD CTRL.DONE
ST CTRL_DONE
LD CTRL.ERR
ST CTRL_ERR

Note: In IL, the function call has to be performed in one line.

Function call in ST

CTRL
(EN := CTRL_EN,
CONO := CTRL_CONO,
SLV := CTRL_SLV,
GROUP_SEL := CTRL_GROUP_SEL,
CLR_DATA := TRL_CLR_DATA,
UNFREEZE := CTRL_UNFREEZE,
FREEZE := CTRL_FREEZE,
UNSYNC := CTRL_UNSSYNC,
SYNC := CTRL_SYNC);

CTRL_DONE := CTRL_DONE;
CTRL_ERR := CTRL_ERR;
Selection of the called slaves

During offline configuration of the master using 907 AC 1131 or 907 FB 1131 the master is first informed with which slaves it should exchange process data. Here, the inserted slaves are not yet assigned to a particular group and can only be called individually via their bus address. In a further step the slaves can be combined to logical groups (refer to AC1131 and FB1131 documentation). For this purpose, first the global properties of each single group must be determined. Each group can have only SYNC properties, only FREEZE properties or both properties. Up to eight groups can be defined. Then, as part of the slave configuration or of the slave properties, it must be determined to which group or groups the individual slave is to be assigned. The group assignment itself does not influence the cyclic exchange of process data. It becomes only effective in conjunction with Global Control commands.

During the boot process of the bus and the parameter assignment of the slaves performed by the master, each single slave is informed about its group assignment. This information is combined in one byte with each single bit representing one of the eight possible groups.

```
G  8 G  7 G  6 G  5 G  4 G  3 G  2 G  1
7  6  5  4  3  2  1  0
```

For instance, if a slave shall be assigned to the groups 7 and 1, the master sends a byte with the decimal value 65 = binary value 01000001 to this slave.

The selection which slaves shall be called by a Global Control command during the running operation is done by the block inputs SLV and GROUP_SEL. With SLV = 0..126 only the slave with a bus address = SLV is called, independent of the group assignment and of the value applied at input GROUP_SEL. All other slaves discard this telegram. With SLV = 127 and GROUP_SEL = 0 all slaves on the bus are called, independent of their group assignment. The group assignment performed during the configuration is only considered by the slaves if they receive a Global Control command with SLV = 127 and GROUP_SEL unequal to 0. In this case, the slave compares the GROUP_SEL value with the group assignment byte received during the parameter assignment. The slave accepts the command if a bit-wise collation of the two values delivers a value unequal to 0 and discards the command if the collation delivers a value of 0. For instance, if a slave which is assigned to groups 1 and 7 (see above) receives a Global Control command with SLV = 127 and GROUP_SEL = 64 dec. (= 01000000 bin.), the command is also (among others) addressed to this slave.

```
Group assignment
GROUP_SEL AND
Result
```

The following table provides an overview of possible combinations of the three parameters group assignment, SLV and GROUP_SEL and the slaves called with these combinations.

<table>
<thead>
<tr>
<th>SLV</th>
<th>GROUP_SEL</th>
<th>Group assignment</th>
<th>Called slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 126</td>
<td>X</td>
<td>X</td>
<td>Only slave with bus address = SLV</td>
</tr>
<tr>
<td>127</td>
<td>0</td>
<td>X</td>
<td>All slaves</td>
</tr>
<tr>
<td>127</td>
<td>1 - 255</td>
<td>1 - 255</td>
<td>Slaves for which the bit-wise collation of group assignment and GROUP_SEL delivers a value unequal to 0.</td>
</tr>
</tbody>
</table>
SENDING GLOBAL CONTROL COMMANDS TO THE DP SLAVE

Possible combinations of Global Control commands

During the process data exchange, the master cyclically transmits the output data to the corresponding slave which applies these data immediately to its outputs. In return a slave transmits the values presently applied at its inputs to the master. During this process, the slaves are called one after the other within one bus cycle of the master. As a result, a small time difference appears between the moments at which the individual slaves output the received data at their local outputs. In the same way the instants of the acquisition of values at the slave inputs and their transmission to the master differ. A time-synchronization of the inputs or outputs is achieved with the help of Global Control commands. While a CLR_DATA causes all called slaves to set their outputs to 0 once and at the same time, the combinations of SYNC / UNSYNC or FREEZE / UNFREEZE must be considered together. The following table shows the possible combinations within a Global Control command and explains their effects.

<table>
<thead>
<tr>
<th>CLR_DATA</th>
<th>SYNC</th>
<th>UNSYNC</th>
<th>FREEZE</th>
<th>UNFREEZE</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>All called slaves set their outputs to 0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>No effect to SYNC mode</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>SYNC mode for output data is terminated</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>SYNC mode; the output data received last are applied to the outputs</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>SYNC mode for output data is terminated</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>No effect to FREEZE mode</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>FREEZE mode for input data is terminated</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>FREEZE mode; present input values are stored and transmitted to the master during the subsequent cycles</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>FREEZE mode for input data is terminated</td>
</tr>
</tbody>
</table>
SENDING USER PARAMETERS TO A DP SLAVE

With the block DPM_SETPRM the user parameters of a slave which were preset during the configuration can be modified during the running operation. The new parameters are immediately sent to the corresponding slave. After re-starting the controller or the slave, the values preset in the configuration are used again.

### Block data

**Available as of PLC runtime system:** V4.0

**Included in library:** PROFIBUS_Master_S90_V43.LIB

**Remark:**

### Block type

Function block with historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DPM_SETPRM</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>SLV</td>
<td>BYTE</td>
<td>Bus address of the called DP slave</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Memory address of the parameters (via ADR operator)</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of parameter data to be sent (byte value)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

### Description

The block DPM_SETPRM implements the PROFIBUS function Set_Prm.

With the function Set_Prm the user parameters of a slave can be modified during the running operation. The parameters are immediately sent to the slave. They are valid until modified parameters are transmitted once again. After re-starting the controller or the slave, the values preset in the configuration are used again.

Format and length of the user parameters are slave-specific. Due to this, the block DPM_SETPRM has inputs where only the variable address and the length of the user parameters to be sent must be specified. It is in the responsibility of the user that the data comply with the requirements of the corresponding device concerning format and length (e.g. by defining a structure).

Every time a FALSE → TRUE edge is applied to input EN, DPM_SETPRM reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until the processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

<table>
<thead>
<tr>
<th>EN</th>
<th>BOOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.</td>
<td></td>
</tr>
</tbody>
</table>

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE. While the request is processed, state changes at input EN are recognized but not evaluated.

<table>
<thead>
<tr>
<th>CONO</th>
<th>BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.</td>
<td></td>
</tr>
</tbody>
</table>

| R012X | CONO = 1 |
| R01X2 | CONO = 2 |

<table>
<thead>
<tr>
<th>SLV</th>
<th>BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>At input SLV the bus address of the DP slave is applied, to which the user parameters shall be sent. Valid addresses are values between 0 and 126.</td>
<td></td>
</tr>
</tbody>
</table>
DATA DWORD
At input DATA the address of the variable from which on the parameters to be sent are stored is specified via the address operator ADR. The data format and the length of the variable itself must be chosen this way that they correspond to the structure of the user parameters of the slave. It is recommended to define a STRUCT data type which represents an image of the slave's structure first and then to declare a variable of this type. Please note that at this point only the user parameters are to be specified. The standard parameters cannot be modified during the running operation. The standard parameter settings specified during configuration are automatically completed by the block.

DATA_LEN BYTE
The input DATA_LEN is used to inform the block about the length (number of bytes) of the user parameters to be transmitted. The maximum length is 237 bytes.

DONE BOOL
The output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing is finished, DONE is set to TRUE.

The values applied at the block outputs are only valid if DONE = TRUE. The parameters were only transmitted correctly if also ERR = 0. If ERR is not 0, an error occurred.

ERR WORD
At the output ERR an error identifier is applied if an invalid value was applied to an input or if an error occurred during the request processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained following the block descriptions.

Function call in IL
LD SETPRM_DATA
ADR
ST SETPRM_DATA_ADR
CAL SETPRM
(EN := SETPRM_EN,
CONO := SETPRM_CONO,
SLV := SETPRM_SLV,
DATA := SETPRM_DATA,
DATA_LEN := SETPRM_DATA_LEN)
LD SETPRM_DONE
ST SETPRM_DONE
LD SETPRM_ERR
ST SETPRM_ERR

Note: In IL, the function call has to be performed in one line.

Function call in ST
SETPRM
(EN := SETPRM_EN,
CONO := SETPRM_CONO,
SLV := SETPRM_SLV,
DATA := SETPRM_DATA,
DATA_LEN := SETPRM_DATA_LEN);
SETPRM_DONE := SETPRM_DONE;
SETPRM_ERR := SETPRM_ERR;
With the block DPM_READ_INPUT even the input data of those slaves can be read which are not assigned to the master, i.e. which were not configured by the master.

### Block data

**Available as of PLC runtime system:** V4.0

**Included in library:** PROFIBUS_Master_S90_V43.LIB

**Remark:**

### Block type

Function block with historical values

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>DPM_READ_INPUT</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>SLV</td>
<td>BYTE</td>
<td>DP slave address</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Memory address for input data (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of the read input data</td>
</tr>
</tbody>
</table>

### Description

The block DPM_READ_INPUT implements the acyclic PROFIBUS function DDLM_Read_Input. Using this function the master can also read the input data of those slaves which are assigned to other masters. DPM_READ_INPUT works outside the cyclic process data exchange.

Every time a FALSE → TRUE edge is applied to input EN, DPM_READ_INPUT reads the data at the inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until the processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

**CONO**

BYTE

At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R012X CONO = 1

R01X2 CONO = 2

**SLV**

BYTE

At input SLV the bus address of the DP slave whose input data are to be read is applied.

**DATA**

DWORD

The address of the variable which shall be used to store the received input data is specified at the input DATA via the address operator ADR. The size of the variable itself must be chosen accordingly so that it is able to hold all input data of the slave (e.g. BYTE array). Furthermore, the format (BYTE, WORD, etc.) of the slave inputs must be considered. If the slave has mixed inputs of different types, it is recommended to first define a STRUCT data type which represents an image of the slave's structure (see I/O configuration of the slave) and then to declare a variable of this type.
DONE
The output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing is finished, DONE is set to TRUE.

The values applied at the block outputs and the input data of the slave are only valid if DONE = TRUE. The input data were only read correctly if also ERR = 0. If ERR is not 0, an error occurred.

ERR
At the output ERR an error identifier is applied if an invalid value was applied to an input or if an error occurred during the request processing. ERR has always to be considered together with the output done. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained after the block descriptions.

DATA_LEN
The output DATA_LEN displays the length (in bytes) of the input data read by the slave. DATA_LEN is only valid if DONE = TRUE and ERR = 0. If DATA_LEN contains a value X which is not 0, the block has stored X bytes of input data in the variable specified with DATA. For instance, if DATA is a byte array with a start index of 1, the valid input data of the slave are contained in the entries DATA[1] to DATA[X].

Function call in IL

LD READ_INPUT_DATA
ADR
ST READ_INPUT_DATA_ADR
CAL READ_INPUT
(EN := READ_INPUT_EN,
CONO := READ_INPUT_CONO,
SLV := READ_INPUT_SLV,
DATA := READ_INPUT_DATA,)

LD READ_INPUT_DONE
ST READ_INPUT_DONE
LD READ_INPUT_ERR
ST READ_INPUT_ERR
LD READ_INPUT_DATA_LEN
ST READ_INPUT_DATA_LEN

Note: In IL, the function call has to be performed in one line.

Function call in ST

READ_INPUT
(EN := READ_INPUT_EN,
CONO := READ_INPUT_CONO,
SLV := READ_INPUT_SLV,
DATA := ADR(READ_INPUT_DATA));

READ_INPUT_DONE := READ_INPUT_DONE;
READ_INPUT_ERR := READ_INPUT_ERR;
READ_INPUT_DATA_LEN := READ_INPUT_DATA_LEN;
READING THE OUTPUT DATA OF A SLAVE WHICH IS NOT ASSIGNED TO THE MASTER

With the block DPM_READ_OUTPUT even the output data of those slaves can be read which are not assigned to the master, i.e. which were not configured by the master. Writing of the output data is not possible.

Block data
Available as of PLC runtime system: V4.0
Included in library: PROFIBUS_Master_S90_V43.LIB

Block type
Function block with historical values

Parameters
<table>
<thead>
<tr>
<th>Instance</th>
<th>DPM_READ_OUTPUT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>SLV</td>
<td>BYTE</td>
<td>Bus address of the called DP slave</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Memory address for output data (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of the read output data</td>
</tr>
</tbody>
</table>

Description
The block DPM_READ_OUTPUT implements the acyclic PROFIBUS function DDLM_Read_Output. Using this function the master can also read the output data of those slaves which are assigned to other masters. Write access to the output data of the slaves is not possible. DPM_READ_OUTPUT works outside the cyclic process data exchange.

Every time a FALSE → TRUE edge is applied to input EN, DPM_READ_OUTPUT reads the data at the inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until the processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

EN
If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

CONO
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.
R012X CONO = 1
R01X2 CONO = 2

SLV
At input SLV the bus address of the DP slave whose output data are to be read is applied.

DATA
The address of the variable which shall be used to store the received output data is specified at the input DATA via the address operator ADR. The size of the variable itself must be chosen accordingly so that it is able to hold all output data of the slave (e.g. BYTE array). Furthermore, the format (BYTE, WORD, etc.) of the slave outputs must be considered. If the slave has mixed outputs of different types, it is recommended to
first define a STRUCT data type which represents an image of the slave's output structure (see I/O configuration of the slave) and then to declare a variable of this type.

**DONE**

The output DONE reflects the state of the request processing. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing is finished, DONE is set to TRUE.

The values applied at the block outputs and the output data of the slave are only valid if DONE = TRUE. The output data were only read correctly if also ERR = 0. If ERR is not 0, an error occurred.

**ERR**

At the output ERR an error identifier is applied if an invalid value was applied to an input or if an error occurred during the request processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained after the block descriptions.

**DATA_LEN**

The output DATA_LEN displays the length (in bytes) of the output data read by the slave. DATA_LEN is only valid if DONE = TRUE and ERR = 0. If DATA_LEN contains a value X which is not 0, the block has stored X bytes of output data in the variable specified with DATA. For instance, if DATA is a byte array with a start index of 1, the valid output data of the slave are contained in the entries DATA[1] to DATA[X].

### Function call in IL

```
LD READ_OUTPUT_DATA
ADR
ST READ_OUTPUT_DATA_ADR
CAL READ_OUTPUT
  (EN := READ_OUTPUT_EN,
   CONO := READ_OUTPUT_CONO,
   SLV := READ_OUTPUT_SLV,
   DATA := READ_OUTPUT_DATA),
LD READ_OUTPUT_DONE
ST READ_OUTPUT_DONE
LD READ_OUTPUT.ERR
ST READ_OUTPUT.ERR
LD READ_OUTPUT_DATA_LEN
ST READ_OUTPUT_DATA_LEN
```

Note: In IL, the function call has to be performed in one line.

### Function call in ST

```
READ_OUTPUT
  (EN := READ_OUTPUT_EN,
   CONO := READ_OUTPUT_CONO,
   SLV := READ_OUTPUT_SLV,
   DATA := ADR(READ_OUTPUT_DATA));

READ_OUTPUT_DONE
 := READ_OUTPUT_DONE;
READ_OUTPUT_ERR
 := READ_OUTPUT_ERR;
READ_OUTPUT_DATA_LEN
 := READ_OUTPUT_DATA_LEN;
```
The block DPS_STAT outputs the state of a PROFIBUS coupler in operation mode DP slave.

**Block data**

- **Available as of PLC runtime system:** V4.0
- **Included in library:**
  - PROFI40.LIB
  - PROFIBUS_S90_V41.LIB
  - PROFIBUS_S90_V42.LIB
  - PROFIBUS_Slave_S90_V43.LIB

**Remark:**
- CONO, ERR, ERR_CNT, LAST
- ERR - type changed

**Block type**

Function block without historical values

**Parameters**

- **Instance**:
  - Instance name: DPS_STAT

- **EN**
  - BOOL: Enabling the block processing

- **CONO**
  - BYTE: Slot (module number) of the coupler

- **ERR**
  - WORD: Error message

- **STATE_BITS**
  - DPS_STATE_BITS_TYPE: Communication states

- **ERR_CNT**
  - WORD: Number of errors occurred since system start

- **LAST_ERR**
  - BYTE: Identifier of the last occurred error

**Description**

The block DPS_STAT outputs the current PROFIBUS coupler state in operation mode DP slave.

**EN**

The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are applied to the outputs.

**CONO**

At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R012X CONO = 1
R01X2 CONO = 2

**Note:** Up to version V4.2 (incl.) the data type of CONO was INT.

**ERR**

An error identification is applied at the output ERR if an invalid value is applied at an input. ERR has always to be considered together with the input EN. The value applied at ERR is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The error messages encoding at the output ERR applies to all PROFIBUS blocks and is explained at the beginning of this description.

**Note:** Up to version V4.2 (incl.) the data type of ERR was INT.

**STATE_BITS**

At output STATE_BITS the current communication states at the PROFIBUS are displayed. STATE_BITS is only valid if EN = TRUE and ERR = 0.

The structure of the type DPS_STATE_BITS_TYPE is defined in the PROFIBUS slave library and is described below.
ERR_CNT WORD
At output ERR_CNT the number of errors occurred since the last coupler reset is output. ERR_CNT is only valid if EN = TRUE and ERR = 0.

Note: Up to version V4.2 (incl.) the data type of ERR_CNT was INT.

LAST_ERR BYTE
At output LAST_ERR the identifier of the last occurred error is output. The error encoding in LAST_ERR is described below. LAST_ERR is only valid if EN = TRUE and ERR = 0.

Note: Up to version V4.2 (incl.) the data type of LAST_ERR was INT.

### Function call in IL

CAL STAT
(EN := STAT_EN,
CONO := STAT_CONO)
LD STAT.ERR
ST STAT.ERR
LD STAT.STATE_BITS
ST STAT.STATE_BITS
LD STAT.ERR_CNT
ST STAT.ERR_CNT
LD STAT.LAST_ERR
ST STAT.LAST_ERR

Note: In IL, the function call has to be performed in one line.

### Function call in ST

STAT
(EN := STAT_EN,
CONO := STAT_CONO);
STAT.ERR := STAT.ERR;
STAT.STATE_BITS := STAT.STATE_BITS;
STAT.ERR_CNT := STAT.ERR_CNT;
STAT.LAST_ERR := STAT.LAST_ERR;
STATE_BITS DPS_STATE_BITS_TYPE

The structure STATE_BITS includes five boolean variables which display the different coupler states. Within the PROFIBUS slave library the data type DPS_STATE_BITS_TYPE is declared as follows:

TYPE DPS_STATE_BITS_TYPE:
  STRUCT
    INIT: BOOL;
    RUN: BOOL;
    STATIC_DIAG: BOOL;
    DATA_EXCH: BOOL;
    NEW_PRM: BOOL;
  END_STRUCT
END_TYPE

INIT BOOL

If this bit is set to TRUE, an initialization error occurred. In normal operation INIT should be FALSE. If this is not the case, the parameter and configuration data have to be checked.

RUN BOOL

This bit is set to TRUE, if the initialization phase has been completed successfully and the coupler is ready for communication via PROFIBUS. An error has occurred, if RUN is FALSE for longer time.

STATIC_DIAG BOOL

This bit is set to TRUE, if the coupler itself has a static diagnosis according to the PROFIBUS standard. If at least one standard diagnosis cause according to the standard occurred (see block DPM_SLVDIAG), the coupler independently sends a diagnosis message via the bus. Until a static diagnosis of the coupler exists, no I/O data are exchanged with the DP master.

DATA_EXCH BOOL

If this bit is set to TRUE, the cyclic I/O data exchange between coupler and DP master is performed. In normal operation DATA_EXCH should always be TRUE provided that STATIC_DIAG is not set to TRUE.

NEW_PRM BOOL

NEW_PRM is set to TRUE, if the coupler has received new parameter data from the DP master.
### Encoding of the error messages

<table>
<thead>
<tr>
<th>LAST_ERR</th>
<th>Meaning</th>
<th>Error source</th>
<th>Remedy</th>
</tr>
</thead>
</table>
| 52       | Invalid bus address configured                    | Configuration     | Valid bus addresses 1 - 125  
Check configuration data for coupler in the project and/or reload program to controller |
| 53       | Expect warm start                                 | Coupler           | Contact ABB                                                                             |
| 54       | Received configuration data contain invalid I/O module | Configuration     | An I/O module was configured which is not entered in the GSD file  
Check configuration data for coupler in the project and/or reload program to controller |
| 55       | Received configuration data contain I/O module with invalid length | Configuration     | An I/O module was configured without length  
Check configuration data for coupler in the project and/or reload program to controller |
| 61       | No address switches                               | Coupler           | Contact ABB                                                                             |
| 63       | Unknown process data handshake                    | Control system    | Supply voltage OFF/ON, otherwise contact ABB                                             |
| 70       | Received configuration data contain invalid I/O data length | Configuration     | Add I/O modules to coupler configuration data (if not configured) or reduce I/O data overall length (if overall length exceeded) and reload program to controller |
| 72       | Invalid DPV1 buffer for DP master class 1 communication | Coupler           | Contact ABB                                                                             |
| 73       | Invalid DPV1 buffer for DP master class 2 communication | Coupler           | Contact ABB                                                                             |
| 74       | ASIC not in offline mode during initialization    | Coupler           | Contact ABB                                                                             |
| 75       | ASIC insufficient memory                          | Coupler           | Contact ABB                                                                             |
| 76       | Invalid size I/O data buffer                      | Coupler           | Contact ABB                                                                             |
| 77       | Invalid size diagnosis buffer                     | Coupler           | Contact ABB                                                                             |
| 78       | Invalid size parameter buffer                     | Coupler           | Contact ABB                                                                             |
| 79       | Invalid size configuration data buffer            | Coupler           | Contact ABB                                                                             |
| 80       | Invalid size SSA buffer                           | Coupler           | Contact ABB                                                                             |
| 210      | Error while reading database                      | Configuration/ Coupler | Reload program with configuration data to controller                                  |
SENDING THE ERROR FLAGS OF THE BASIC UNIT FOR AN EXTENDED DIAGNOSIS

The block DPS_EXTDIAG generates an extended diagnosis message from the error flags of the basic unit and sends this message to the master.

### Block data

- Available as of PLC runtime system: V4.0
- Included in library: PROFIBUS_Slave_S90_V43.LIB

### Block type

Function block without historical values

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>DPS_EXTDIAG</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

### Description

The block DPS_EXTDIAG generates an extended diagnosis message according to the PROFIBUS standard. For this, it monitors the state of the sum error flag %MX255.10. If this flag is set (TRUE), the detailed information is entered into the diagnosis telegram according to the states of the error class flags %MX255.11 to %MX255.14. If the corresponding flag is set, the detailed information is entered into the telegram. Otherwise the corresponding range in the diagnosis data is set to zero. Furthermore, the status word %IW1007.15 is always sent. The extended diagnosis data are marked as device-specific data and have always a total length of 26 bytes. The PROFIBUS standard diagnosis data are automatically completed.

If a FK1 or FK2 error occurred (%MX255.11 = TRUE or %MX255.12 = TRUE), the diagnosis information is marked as an error message and can be evaluated accordingly in the master. If only a light error (FK3, %MX255.13 = TRUE) or a warning (FK4, %MX255.14 = TRUE) occurred, the diagnosis information is marked as a status message.

The extended diagnosis is sent until the sum error flag is acknowledged, i.e. until %MX255.10 is set to FALSE. The structure of the diagnosis data is explained later in this section.

- **EN** (BOOL): The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the state of the sum error flag %MX255.10 is monitored. If necessary, extended diagnosis information are sent to the master.

- **CONO** (BYTE): At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

- **DONE** (BOOL): The output DONE indicates whether the block has sent (TRUE) an extended diagnosis message during the current program cycle or not (FALSE).

---

**Remark:**

Included in library: PROFIBUS_Slave_S90_V43.LIB
An error identification is applied at the output ERR if an invalid value is applied at an input. ERR has always to be considered together with the input EN. The value applied at ERR is only valid if EN = TRUE. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained following the block descriptions.

**Function call in IL**

```
CAL  EXTDIAG
(EN := EXTDIAG_EN,
CONO := EXTDIAG_CONO)

LD  EXTDIAG_DONE
ST  EXTDIAG_DONE
LD  EXTDIAG_ERR
ST  EXTDIAG_ERR
```

Note: In IL, the function call has to be performed in one line.

**Function call in ST**

```
EXTDIAG
(EN := EXTDIAG_EN,
CONO := EXTDIAG_CONO);

EXTDIAG_DONE := EXTDIAG_DONE;
EXTDIAG_ERR := EXTDIAG_ERR;
```
LAST_ERR  Encoding of the error messages

The following table gives an overview of the structure of the diagnosis data according to PROFIBUS standard (6 bytes) and the extended diagnosis data (26 bytes).

**PROFIBUS standard diagnosis data**

<table>
<thead>
<tr>
<th>Byte</th>
<th>Designation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stationstatus_1</td>
<td>See DPM_SLVDIAG</td>
</tr>
<tr>
<td>2</td>
<td>Stationstatus_2</td>
<td>See DPM_SLVDIAG</td>
</tr>
<tr>
<td>3</td>
<td>Stationstatus_3</td>
<td>See DPM_SLVDIAG</td>
</tr>
<tr>
<td>4</td>
<td>Stationstatus_4</td>
<td>See DPM_SLVDIAG</td>
</tr>
<tr>
<td>5</td>
<td>Identification number, high byte</td>
<td>Identification number of the slave</td>
</tr>
<tr>
<td>6</td>
<td>Identification number, low byte</td>
<td>Identification number of the slave</td>
</tr>
</tbody>
</table>

**Extended diagnosis data**

<table>
<thead>
<tr>
<th>Byte</th>
<th>Designation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Header device-specific diagnosis</td>
<td>Identification and length of the device-specific diagnosis, see DPM_SLVDIAG</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>always 0</td>
</tr>
<tr>
<td>3</td>
<td>%MW1254.0 (low byte)</td>
<td>FK1 error identifier</td>
</tr>
<tr>
<td>4</td>
<td>%MW1254.0 (high byte)</td>
<td>FK1 error identifier</td>
</tr>
<tr>
<td>5</td>
<td>%MW1254.1 (low byte)</td>
<td>FK1 detailed information 1</td>
</tr>
<tr>
<td>6</td>
<td>%MW1254.1 (high byte)</td>
<td>FK1 detailed information 1</td>
</tr>
<tr>
<td>7</td>
<td>%MW1254.8 (low byte)</td>
<td>FK2 error identifier</td>
</tr>
<tr>
<td>8</td>
<td>%MW1254.8 (high byte)</td>
<td>FK2 error identifier</td>
</tr>
<tr>
<td>9</td>
<td>%MW1254.9 (low byte)</td>
<td>FK2 detailed information 1</td>
</tr>
<tr>
<td>10</td>
<td>%MW1254.9 (high byte)</td>
<td>FK2 detailed information 1</td>
</tr>
<tr>
<td>11</td>
<td>%MW1254.10 (low byte)</td>
<td>FK2 detailed information 2</td>
</tr>
<tr>
<td>12</td>
<td>%MW1254.10 (high byte)</td>
<td>FK2 detailed information 2</td>
</tr>
<tr>
<td>13</td>
<td>%MW1255.0 (low byte)</td>
<td>FK3 error identifier</td>
</tr>
<tr>
<td>14</td>
<td>%MW1255.0 (high byte)</td>
<td>FK3 error identifier</td>
</tr>
<tr>
<td>15</td>
<td>%MW1255.1 (low byte)</td>
<td>FK3 detailed information 1</td>
</tr>
<tr>
<td>16</td>
<td>%MW1255.1 (high byte)</td>
<td>FK3 detailed information 1</td>
</tr>
<tr>
<td>17</td>
<td>%MW1255.2 (low byte)</td>
<td>FK3 detailed information 2</td>
</tr>
<tr>
<td>18</td>
<td>%MW1255.2 (high byte)</td>
<td>FK3 detailed information 2</td>
</tr>
<tr>
<td>19</td>
<td>%MW1255.8 (low byte)</td>
<td>FK4 error identifier</td>
</tr>
<tr>
<td>20</td>
<td>%MW1255.8 (high byte)</td>
<td>FK4 error identifier</td>
</tr>
<tr>
<td>21</td>
<td>%MW1255.9 (low byte)</td>
<td>FK4 detailed information 1</td>
</tr>
<tr>
<td>22</td>
<td>%MW1255.9 (high byte)</td>
<td>FK4 detailed information 1</td>
</tr>
<tr>
<td>23</td>
<td>%MW1255.10 (low byte)</td>
<td>FK4 detailed information 2</td>
</tr>
<tr>
<td>24</td>
<td>%MW1255.10 (high byte)</td>
<td>FK4 detailed information 2</td>
</tr>
<tr>
<td>25</td>
<td>%IW1007.15 (low byte)</td>
<td>Status word</td>
</tr>
<tr>
<td>26</td>
<td>%IW1007.15 (high byte)</td>
<td>Status word</td>
</tr>
</tbody>
</table>
READING OUT THE USER PARAMETERS

With the block DPS_GETPRM the current user parameters of the slave can be read.

**Block data**

- Available as of PLC runtime system: V4.0
- Included in library: PROFIBUS_Slave_S90_V43.LIB

**Block type**

- Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>DPS_GETPRM</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the slave</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>BOOL1</td>
<td>BOOL</td>
<td>Switch 1, BOOL parameter</td>
</tr>
<tr>
<td>BOOL2</td>
<td>BOOL</td>
<td>Switch 2, BOOL parameter</td>
</tr>
<tr>
<td>BOOL3</td>
<td>BOOL</td>
<td>Switch 3, BOOL parameter</td>
</tr>
<tr>
<td>BOOL4</td>
<td>BOOL</td>
<td>Switch 4, BOOL parameter</td>
</tr>
<tr>
<td>BOOL5</td>
<td>BOOL</td>
<td>Switch 5, BOOL parameter</td>
</tr>
<tr>
<td>BOOL6</td>
<td>BOOL</td>
<td>Switch 6, BOOL parameter</td>
</tr>
<tr>
<td>BOOL7</td>
<td>BOOL</td>
<td>Switch 7, BOOL parameter</td>
</tr>
<tr>
<td>BOOL8</td>
<td>BOOL</td>
<td>Switch 8, BOOL parameter</td>
</tr>
<tr>
<td>BYTE1</td>
<td>BYTE</td>
<td>Value byte 1, BYTE parameter</td>
</tr>
<tr>
<td>BYTE2</td>
<td>BYTE</td>
<td>Value byte 2, BYTE parameter</td>
</tr>
<tr>
<td>INT1</td>
<td>INT</td>
<td>Integer value 1, INT parameter</td>
</tr>
<tr>
<td>INT2</td>
<td>INT</td>
<td>Integer value 1, INT parameter</td>
</tr>
</tbody>
</table>

**Description**

The block DPS_GETPRM implements the PROFIBUS function Get_Prm.

With the block DPS_GETPRM the current user parameters of the slave can be read. In addition to the 7 bytes of standard parameters the master sends 10 bytes of user parameters to the slave. The standard parameters are defined by the configuration and cannot be modified. The first 3 bytes of the user parameters are intended for internal use and cannot be modified as well. The remaining user parameters divide in 8 BOOL values, 2 BYTE values and 2 INT values. These parameters can be used as desired.

The default values of the user parameters are preset during the slave configuration in the master. These values are sent once to the slave every time the slave is taken into operation by the master (restart of the master or of the slave). With the help of the PROFIBUS function Set_Prm (see also DPM_SETPRM) the user parameters can be used by the master during the
running operation and thus they can be used as application specific commands.

The reception of new parameter data is indicated at the output NEW_PRM of the block DPS_STAT.

Every time a FALSE → TRUE edge is applied to input EN, DPM_GET_PRM reads the values at the inputs and then reads the current user parameters from the coupler. Further FALSE → TRUE edges at input EN are ignored until the processing of the active requests is finished. The completion of the request processing is indicated by DONE = TRUE.

**EN**
If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE. While the request is processed, state changes at input EN are recognized but not evaluated.

**CONO**
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R012X CONO = 1
R01X2 CONO = 2

**DONE**
The output DONE reflects the state of the request processing. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If the processing is finished, DONE is set to TRUE for one cycle.

The values applied at the block outputs are only valid if DONE = TRUE. The parameters applied at the block outputs are only valid if also ERR = 0. If ERR is not 0, an error occurred.

**ERR**
At the output ERR an error identifier is applied if an invalid value was applied to an input or if an error occurred during the request processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at the output ERR applies to all PROFIBUS blocks and is explained after the block descriptions.

**BOOL1 ... BOOL8**
At the outputs BOOL1 to BOOL8 the current states of the boolean user parameters (switches) are indicated. These are only valid if DONE = TRUE and ERR = 0.

**BYTE1 ... BYTE2**
At the outputs BYTE1 and BYTE2 the current values of the BYTE user parameters are displayed. These are only valid if DONE = TRUE and ERR = 0.

**INT1 ... INT2**
At the outputs INT1 and INT2 the current values of the INT user parameters are displayed. These are only valid if DONE = TRUE and ERR = 0.
Function call in IL

CAL GETPRM
  (EN := GETPRM.EN,
   CONO := GETPRM)

LD GETPRM_DONE
ST GETPRM_DONE
LD GETPRM.ERR
ST GETPRM.ERR
LD GETPRM.BOOL1
ST GETPRM.BOOL1
LD GETPRM.BOOL2
ST GETPRM.BOOL2
LD GETPRM.BOOL3
ST GETPRM.BOOL3
LD GETPRM.BOOL4
ST GETPRM.BOOL4
LD GETPRM.BOOL5
ST GETPRM.BOOL5
LD GETPRM.BOOL6
ST GETPRM.BOOL6
LD GETPRM.BOOL7
ST GETPRM.BOOL7
LD GETPRM.BOOL8
ST GETPRM.BOOL8
LD GETPRM.BYTE1
ST GETPRM.BYTE1
LD GETPRM.BYTE2
ST GETPRM.BYTE2
LD GETPRM.INT1
ST GETPRM.INT1
LD GETPRM.INT2
ST GETPRM.INT2

Note: In IL, the function call has to be performed in one line.

Function call in ST

GETPRM
  (EN := GETPRM.EN,
   CONO := GETPRM_CONO);

GETPRM_DONE := GETPRM_DONE;
GETPRM_ERR := GETPRM.ERR;
GETPRM_BOOL1 := GETPRM.BOOL1;
GETPRM_BOOL2 := GETPRM.BOOL2;
GETPRM_BOOL3 := GETPRM.BOOL3;
GETPRM_BOOL4 := GETPRM.BOOL4;
GETPRM_BOOL5 := GETPRM.BOOL5;
GETPRM_BOOL6 := GETPRM.BOOL6;
GETPRM_BOOL7 := GETPRM.BOOL7;
GETPRM_BOOL8 := GETPRM.BOOL8;
GETPRM_BYTE1 := GETPRM.BYTE1;
GETPRM_BYTE1 := GETPRM.BYTE1;
GETPRM_INT1 := GETPRM.INT1;
GETPRM_INT2 := GETPRM.INT2;
Glossary

BOOL
Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT
DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

DWORD
DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

INT
INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.
**WORD**

WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

**Functions**

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can be derived:

- Within functions, global variables can neither be read nor written.
- Within functions, absolute operands can neither be read nor written.
- Within functions, function blocks must not be called.

**Function blocks**

Function blocks are subroutines which can have as many inputs, outputs and internal variables as required. They are called from a program or from another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. Therefore e.g. counters can be realized which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters, a function only one. The output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.
Index

D
DPM_COM_ERR_TYPE 23
DPM_CTRL S90 27
DPM_READ_INPUT S90 34
DPM_READ_OUTPUT S90 36
DPM_SETPRM S90 32
DPM_SLVDIAG S90 10, 26, 40
DPM_STAT S90 19
DPM_STATE_BITS_TYPE 22
DPM_SYSDIAG S90 25
DPS_EXTDIAG S90 42
DPS_GETPRM S90 45
DPS_STAT S90 38

O
Output ERR of a PROFIBUS block 5

P
PROFI_INFO S90 7
PROFIBUS S90
  DPM_CTRL 27
  DPM_READ_INPUT 34
  DPM_READ_OUTPUT 36
  DPM_SETPRM 32
  DPM_SLVDIAG 10
  DPM_STAT 19
  DPM_SYSDIAG 25
  DPS_EXTDIAG 42
  DPS_GETPRM 45
  DPS_STAT 38
  PROFI_INFO 7

S
STATIONSTATUS_1_TYPE 13
STATIONSTATUS_2_TYPE 14
STATIONSTATUS_3_TYPE 15
## Contents

### DeviceNet library

- SPECIAL CHARACTERISTICS IN THE DEVICE NET LIBRARY .................................. 2
- COMPONENTS OF THE DEVICE NET LIBRARY .................................. 2
  - Function blocks .................................. 2
  - Data types .................................. 2
- ERR - ERROR MESSAGES OF THE DEVICE NET BLOCKS .................................. 3
  - Errors at block inputs .................................. 3
  - Execution errors, coupler or slave responses .................................. 3

### Overview of blocks arranged according to their call names

<table>
<thead>
<tr>
<th>Call Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>READING INFORMATION ABOUT THE INSTALLED DEVICE NET COUPLERS</td>
<td>DEVNET_INFO ....... 5</td>
</tr>
<tr>
<td>POLLING DIAGNOSTIC DATA FROM A SLAVE</td>
<td>DNM_DEVDIAG ...... 7</td>
</tr>
<tr>
<td>READING THE DEVICE NET COUPLER STATUS</td>
<td>DNM_STAT ........... 12</td>
</tr>
<tr>
<td>READING A STATUS SURVEY OF ALL SLAVES</td>
<td>DNM_SYSDIAG ....... 18</td>
</tr>
<tr>
<td>READING AN ATTRIBUTE FROM A SLAVE OBJECT</td>
<td>DNM_GET_ATTR... 20</td>
</tr>
<tr>
<td>RESETTING A SLAVE OBJECT</td>
<td>DNM_RESET_OBJ 23</td>
</tr>
<tr>
<td>WRITING AN ATTRIBUTE TO A SLAVE OBJECT</td>
<td>DNM_SET_ATTR ...25</td>
</tr>
</tbody>
</table>

### Glossary

- 28

### Index

- 30
DeviceNet library

SPECIAL CHARACTERISTICS IN THE DEVICENET LIBRARY

Note:

DeviceNet communication is only performed in RUN mode of the PLC and not in simulation mode.

Prerequisites for use of the library

The function blocks contained in the DeviceNet library access to both, the PLC run time system and directly to the coupler. The required definitions and functions are stored in the internal library Coupler_S90_V41.LIB.

When using the DeviceNet library DeviceNet_S90_V41.LIB, it is absolutely essential to insert also the library Coupler_S90_V41.LIB into the project.

COMPONENTS OF THE DEVICENET LIBRARY

The library contains function blocks which allow an easy handling of the DeviceNet coupler. Additionally various data types are defined in this library. These structures enable a clear presentation of data sets.

Function blocks

The DeviceNet library contains the following function blocks:

<table>
<thead>
<tr>
<th>Group</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVNET_INFO</td>
<td>- Reading information about installed couplers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group</th>
<th>DeviceNet master</th>
</tr>
</thead>
</table>

Status/Diagnostics:

- DNM_DEVDIA - Polling diagnostic data from a slave
- DNM_STAT - Reading the DeviceNet coupler status
- DNM_SYSDIA - Displaying status information about all slaves

Object and attribute handling:

- DNM_GET_ATTR - Reading an attribute from a slave object
- DNM_RESET_OBJ - Resetting a slave object
- DNM_SET_ATTR - Writing an attribute to a slave object

Detailed information about the various blocks can be found in the following sections.

Data types

The following data types (structures) are defined in the DeviceNet library:

- DNM_COM_ERR_TYPE - Communication error
- DNM_STATE_BITS_TYPE - Bits for coupler state description
- DEVICESTATUS_1_TYPE - Slave diagnosis

For detailed information about the different data types please refer to the specific description of the corresponding block.
ERR - ERROR MESSAGES OF THE DEVICENET BLOCKS

If the output ERR of a DeviceNet block is unequal to 0, either one of the block input values is invalid or an internal error occurred during block execution. The error cause can be determined with the help of the following tables.

Errors at block inputs

<table>
<thead>
<tr>
<th>ERR DEC</th>
<th>HEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8193</td>
<td>2001</td>
<td>Invalid value at block input CONO</td>
</tr>
<tr>
<td>8194</td>
<td>2002</td>
<td>Coupler model in slot CONO does not correspond to the block type</td>
</tr>
<tr>
<td>8195</td>
<td>2003</td>
<td>No DeviceNet master coupler in slot CONO</td>
</tr>
<tr>
<td>8196</td>
<td>2004</td>
<td>DeviceNet master coupler in slot CONO is not in OPERATE state</td>
</tr>
<tr>
<td>8197</td>
<td>2005</td>
<td>Invalid status survey type</td>
</tr>
<tr>
<td>8198</td>
<td>2006</td>
<td>Invalid slave address at block input</td>
</tr>
<tr>
<td>8199</td>
<td>2007</td>
<td>Invalid module number at block input</td>
</tr>
<tr>
<td>8200</td>
<td>2008</td>
<td>Invalid module index at block input</td>
</tr>
<tr>
<td>8201</td>
<td>2009</td>
<td>Invalid class ID at block input</td>
</tr>
<tr>
<td>8202</td>
<td>200A</td>
<td>Invalid data length at block input</td>
</tr>
</tbody>
</table>

Execution errors, coupler or slave responses

<table>
<thead>
<tr>
<th>ERR DEC</th>
<th>HEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>9218</td>
<td>2402</td>
<td>Slave does not have the required resources</td>
</tr>
<tr>
<td>9224</td>
<td>2408</td>
<td>Service unavailable in slave</td>
</tr>
<tr>
<td>9225</td>
<td>2409</td>
<td>Slave reports invalid attribute value</td>
</tr>
<tr>
<td>9227</td>
<td>240B</td>
<td>Slave already in request mode</td>
</tr>
<tr>
<td>9228</td>
<td>240C</td>
<td>Object status conflict in slave</td>
</tr>
<tr>
<td>9230</td>
<td>240E</td>
<td>Attribute not writable in slave</td>
</tr>
<tr>
<td>9231</td>
<td>240F</td>
<td>No access release in slave</td>
</tr>
<tr>
<td>9232</td>
<td>2410</td>
<td>Not possible to run service in current slave status</td>
</tr>
<tr>
<td>9233</td>
<td>2411</td>
<td>No slave response, timeout</td>
</tr>
<tr>
<td>9235</td>
<td>2413</td>
<td>Slave has received too few data</td>
</tr>
<tr>
<td>9236</td>
<td>2414</td>
<td>Attribute not supported in slave</td>
</tr>
<tr>
<td>9237</td>
<td>2415</td>
<td>Slave has received too many data</td>
</tr>
<tr>
<td>9238</td>
<td>2416</td>
<td>Object does not exist in slave</td>
</tr>
<tr>
<td>9239</td>
<td>2417</td>
<td>Too many data, overflow of internal buffer in slave</td>
</tr>
<tr>
<td>9264</td>
<td>2430</td>
<td>Slave not configured within the coupler, Explicit Message channel not available</td>
</tr>
<tr>
<td>9266</td>
<td>2432</td>
<td>Coupler reports format error in response telegram of slave</td>
</tr>
<tr>
<td>9270</td>
<td>2436</td>
<td>Coupler reports that a parallel service is already active</td>
</tr>
<tr>
<td>9271</td>
<td>2437</td>
<td>Slave address out of valid range</td>
</tr>
<tr>
<td>9273</td>
<td>2439</td>
<td>Sequence error in fragmented response sequence of slave</td>
</tr>
<tr>
<td>9377</td>
<td>24A1</td>
<td>Slave address out of valid range</td>
</tr>
<tr>
<td>9416</td>
<td>24C8</td>
<td>Slave not configured within the coupler, database not found</td>
</tr>
</tbody>
</table>
Overview of blocks arranged according to their call names

Character description:

FBmV  … Function block with historical values
FBoV  … Function block without historical values
F    … Function

<table>
<thead>
<tr>
<th>CE name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVNET_INFO</td>
<td>FBoV</td>
<td>Reading information about the installed couplers</td>
<td>5</td>
</tr>
<tr>
<td>DNM_GET_ATTR</td>
<td>FBmV</td>
<td>Reading an attribute from a slave object</td>
<td>20</td>
</tr>
<tr>
<td>DNM_DEVDIAG</td>
<td>FBmV</td>
<td>Reading diagnostic data from a slave (clients)</td>
<td>7</td>
</tr>
<tr>
<td>DNM_RESET_OBJ</td>
<td>FBmV</td>
<td>Resetting a slave object</td>
<td>23</td>
</tr>
<tr>
<td>DNM_SET_ATTR</td>
<td>FBmV</td>
<td>Writing an attribute to a slave object</td>
<td>25</td>
</tr>
<tr>
<td>DNM_STAT</td>
<td>FBoV</td>
<td>Reading the DeviceNet coupler status</td>
<td>12</td>
</tr>
<tr>
<td>DNM_SYSDIAG</td>
<td>FBoV</td>
<td>Reading status surveys about all slaves (clients)</td>
<td>18</td>
</tr>
</tbody>
</table>
READING INFORMATION ABOUT THE INSTALLED
DEVICENET COUPLERS

The block DEVNET_INFO outputs coupler related information. The following items are displayed:
coupler type and model, operating mode, manufacturing date, device number and serial number as well as firmware designation and firmware version.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DEVNET_INFO</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>TYP</td>
<td>STRING(17)</td>
<td>Coupler type</td>
</tr>
<tr>
<td>MODEL</td>
<td>STRING(22)</td>
<td>Coupler model</td>
</tr>
<tr>
<td>OP_MODE</td>
<td>STRING(22)</td>
<td>Operating mode of the coupler</td>
</tr>
<tr>
<td>MAN_DATE</td>
<td>DATE</td>
<td>Manufacturing date of the coupler</td>
</tr>
<tr>
<td>DEV_NO</td>
<td>DWORD</td>
<td>Device number of the coupler</td>
</tr>
<tr>
<td>SER_NO</td>
<td>DWORD</td>
<td>Serial number of the coupler</td>
</tr>
<tr>
<td>FW_NAME</td>
<td>STRING(17)</td>
<td>Designation of the coupler firmware</td>
</tr>
<tr>
<td>FW_VER</td>
<td>STRING(17)</td>
<td>Version of the coupler firmware</td>
</tr>
</tbody>
</table>

Description

The block DEVNET_INFO is always active. It reads the slot number at the block input CONO and outputs the corresponding information about the selected coupler at the outputs.

DEVNET_INFO recognizes different coupler types. For that reason, the block outputs do not only relate to the DeviceNet coupler.

The block is not intended for usage in normal user programs. It should be mainly used to support error diagnostics and maintenance operations.

CONO
The slot of the coupler (module number) whose information are to be requested is specified at input CONO. The module number depends on the PLC category.

07 SL 97 R015X CONO = 1
07 SL 97 R01X5 CONO = 2

ERR
ERR outputs an error ID, if an invalid value is applied to input CONO. The encoding of the error messages available at ERR applies to all DeviceNet blocks and is explained in section "ERR – Error messages of the DeviceNet blocks".

TYP

TYP STRING(17)
TYP outputs the coupler type (i.e. the coupler design). Different types are recognized by the block.

The DeviceNet coupler has the type TYP = 'single mode'.
If a coupler type cannot be determined, the output TYP is set to 'unknown'.
READING INFORMATION ABOUT THE INSTALLED
DEVICENET COUPLERS

MODEL STRING(22)
MODEL outputs the transfer protocol supported by the coupler as well as the subscriber type.

The model ID of the DeviceNet coupler is
MODEL = ‘DeviceNet Master’.

If the block detects an unknown model, MODEL outputs the value ‘unknown’.

OP_MODE STRING(22)
OP_MODE outputs the current coupler operating mode setting. This output is only applicable for couplers with switchable operating modes.

Therefore this output is not used for the DeviceNet coupler.

MAN_DATE DATE
MAN_DATE outputs the manufacturing date of the coupler. The date is a variable of the data type DATE and has the format D#YYYY-MM-DD. The initial value is D#2000-01-01.

DEV_NO DWORD
DEV_NO outputs the device number of the coupler.

SER_NO DWORD
SER_NO outputs the serial number of the coupler.

FW_NAME STRING(17)
FW_NAME outputs the name of the coupler firmware.

FW_VER STRING(17)
FW_VER outputs the release number (version) and issue date of the coupler firmware. These data are represented as a string (e.g., ‘V1.003 15.07.00’).

Note:
The inputs and outputs can neither be duplicated nor inverted.

Function call in IL
CAL INFO(CONO := INFO_CONO)
LD INFO.ERR
ST INFO_ERR
LD INFO_TYP
ST INFO_TYP
LD INFO.MODEL
ST INFO_MODEL
LD INFO.OP_MODE
ST INFO_OP_MODE
LD INFO.MAN_DATE
ST INFO_MAN_DATE
LD INFO.DEV_NO
ST INFO_DEV_NO
LD INFO.SER_NO
ST INFO_SER_NO
LD INFO.FW_NAME
ST INFO_FW_NAME
LD INFO.FW_VER
ST INFO_FW_VER

Note:
In IL, the function call has to be performed in one line.

Function call in ST
INFO(CONO := INFO_CONO);
INFO.ERR := INFO.ERR;
INFO_TYP := INFO_TYP;
INFO_MODEL := INFO_MODEL;
INFO.OP_MODE := INFO.OP_MODE;
INFO.MAN_DATE := INFO.MAN_DATE;
INFO.DEV_NO := INFO.DEV_NO;
INFO.SER_NO := INFO.SER_NO;
INFO.FW_NAME := INFO.FW_NAME;
INFO.FW_VER := INFO.FW_VER;
POLLING DIAGNOSTIC DATA FROM A SLAVE

The block IBM_DEVDIAG reads the diagnostic data of a slave.

```
+-------------------------------+-------------------------------+
|    DIAG                       |    DIAG                       |
|  DIAG_EN→                    |  DIAG_ERR→                   |
|  DIAG_CONO→                  |  DIAGSTAT_1→                 |
|  DIAG_MAC_ID→                |  DIAG_DEVSTATE→              |
|  STAT_1→                     |  DIAG_ONL_ERR→               |
|  DEV_STATE→                  |  DIAG_GEN_ERR→               |
|  ONL_ERR→                    |  DIAG_ADD_ERR→               |
|  GEN_ERR→                    |  DIAG_HEART_TO→              |
|  ADD_ERR→                    |  DONE→                       |
|  HEART_TO→                   |  DIAG_DONE→                  |
+-------------------------------+-------------------------------+
```

**Block type**

Function block with historical values

**Parameters**

- **Instance**: DNM_DEVDIAG
- **EN**: BOOL
- **CONO**: BYTE
- **MAC_ID**: BYTE
- **DONE**: BOOL
- **ERR**: WORD
- **STAT_1**: DEVICESTATUS_1_TYPE
- **DEV_STATE**: BYTE
- **ONL_ERR**: BYTE
- **GEN_ERR**: BYTE
- **ADD_ERR**: BYTE
- **HEART_TO**: WORD

**Description**

Using the block DNM_DEVDIAG, diagnostic data of individual slaves can be requested.

Every time a FALSE → TRUE edge is applied to the input EN, DNM_DEVDIAG reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

For a FALSE → TRUE edge at input EN, all further inputs are applied.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

**CONO**

The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

- 07 SL 97 R015X \ CONO = 1
- 07 SL 97 R01X5 \ CONO = 2

**MAC_ID**

The MAC ID of the slave, which diagnostic data are to be requested, is specified at input MAC_ID.
DONE BOOL

The output DONE reflects the processing state. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE.

The values at the block outputs are only valid if DONE = TRUE. The diagnostic data are only valid if also ERR = 0. If ERR is not equal to 0, an error occurred.

ERR WORD

ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all DeviceNet blocks and is explained in section "ERR – Error messages of the DeviceNet blocks".

STAT_1 DEVICESTATUS_1_TYPE

STAT_1 outputs different bits as a structure of the type DEVICESTATUS_1_TYPE which display error states of the slave. STAT_1 is only valid if DONE = TRUE and ERR = 0.

The format of the structure of the type DEVICESTATUS_1_TYPE is defined in the DeviceNet library (see description below).

DEV_STATE BYTE

DEV_STATE outputs the current state of the coupler's internal state machine for the relevant slave. This value provides information about the current phase of communication with the device. DEV_STATE is only valid if DONE = TRUE and ERR = 0.

The IDs of the individual states are described on the following pages.

ONL_ERR BYTE

ONL_ERR outputs a value which describes possible existing communication errors between master coupler and slave. ONL_ERR is only valid if DONE = TRUE and ERR = 0.

The error IDs of ONL_ERR correspond to the IDs of the DNM_STAT block output COM_ERR.EVENT. They are described in the table provided in the DNM_STAT block description.

GEN_ERR BYTE

GEN_ERR outputs the ID of the last error telegram of the slave in accordance to the DeviceNet specification.

GEN_ERR refers only to connection errors (ONL_ERR = 35, see below). Therefore the error ID of GEN_ERR has always to be considered in conjunction with ONL_ERR. GEN_ERR is only valid if DONE = TRUE, ERR = 0 and ONL_ERR = 35.

The GEN_ERR IDs are described on the following pages.

ADD_ERR BYTE

ADD_ERR outputs additional information about the connection error reported at GEN_ERR. Therefore ADD_ERR has always to be considered in conjunction with GEN_ERR. ADD_ERR is only valid if DONE = TRUE, ERR = 0 and GEN_ERR <> 0.

HEART_TO WORD

Output HEART_TO indicates how often response monitoring for this slave was performed.

The DeviceNet bus parameters contain a configurable Heartbeat timer. This timer determines at which time intervals the master should check the availability of the slaves. HEART_TO contains the number of timeouts for the requested slave. Therefore, HEART_TO can be used to make a statement about the quality of the connection to this slave. The value of this input is only valid for DONE = TRUE and ERR = 0.

Note:
The inputs and outputs can neither be duplicated nor inverted.
Function call in IL

CAL     DIAG
(EN := DIAG_EN,
CONO := DIAG_CONO,
MAC_ID := DIAG_MAC_ID)

LD     DIAG_DONE
ST     DIAG_DONE
LD     DIAG_ERR
ST     DIAG_ERR
LD     DIAG_STAT_1
ST     DIAG_STAT_1
LD     DIAG_DEV_STATE
ST     DIAG_DEV_STATE
LD     DIAG_ONL_ERR
ST     DIAG_ONL_ERR
LD     DIAG_GEN_ERR
ST     DIAG_GEN_ERR
LD     DIAG_ADD_ERR
ST     DIAG_ADD_ERR
LD     DIAG_HEART_TO
ST     DIAG_HEART_TO

Function call in ST

DIAG
(EN := DIAG_EN,
CONO := DIAG_CONO,
MAC_ID := DIAG_MAC_ID);

DIAG_DONE := DIAG_DONE;
DIAG_ERR := DIAG_ERR;
DIAG_STAT_1 := DIAG_STAT_1;
DIAG_DEV_STATE := DIAG_DEV_STATE;
DIAG_ONL_ERR := DIAG_ONL_ERR;
DIAG_GEN_ERR := DIAG_GEN_ERR;
DIAG_ADD_ERR := DIAG_ADD_ERR;
DIAG_HEART_TO := DIAG_HEART_TO;

Note:
In IL, the function call has to be performed in one line.

Devicestatus_1 STAT_1 DEVICESTATUS_1_TYPE

The output STAT_1 of DNM_DEVDIAG displays different diagnostic bits as a structure of the type DEVICESTATUS_1_TYPE. Within the DeviceNet library the structure DEVICESTATUS_1_TYPE is declared as follows:

TYPE DEVICESTATUS_1_TYPE:
STRUCT
NO_RESPONSE: BOOL;
reserved1: BOOL;
PRM_FAULT: BOOL;
CFG_FAULT: BOOL;
UCMM_Support: BOOL;
reserved2: BOOL;
reserved3: BOOL;
DEACTIVATED: BOOL;
END_STRUCT
END_TYPE

NO_RESPONSE BOOL
If this bit is set, the slave with the MAC ID specified for the block input MAC_ID does not respond to the master requests. Normally NO_RESPONSE should be set to FALSE.

reserved1 BOOL
This bit is reserved and currently not used.

PRM_FAULT BOOL
This bit is set, if the write access to at least one configured attribute of the slave was refused. In this case a parameterization error occurred.
CFG_FAULT BOOL
This bit is set, if the actual length of the transmitted data (actual configuration) differs from the configured length (nominal configuration). In this case a configuration error occurred.

UCMM_Support BOOL
This bit is reserved and currently not used.

reserved2 BOOL
This bit is reserved and currently not used.

reserved3 BOOL
This bit is reserved and currently not used.

DEACTIVATED BOOL
This bit is set to TRUE, if the slave defined in the configuration data of the master is deactivated and not processed.

DEV_STATE
DEV_STATE outputs the current status of the master’s internal state automation of a slave. The value shows the phase of communication between master and requested slave. The following table describes the possible values of DEV_STATE and the meaning from the master’s point of view.

<table>
<thead>
<tr>
<th>DEV_STATE</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial state of the state automation</td>
</tr>
<tr>
<td>1</td>
<td>Slave inactive and not processed</td>
</tr>
<tr>
<td>2</td>
<td>Waiting for requests of all slaves to check for duplicate MAC IDs</td>
</tr>
<tr>
<td>3</td>
<td>Initialization of the internal predefined master slave structures</td>
</tr>
<tr>
<td>4</td>
<td>Request for allocation of predefined master slave connection sets</td>
</tr>
<tr>
<td>5</td>
<td>Waiting for slave request after request for allocation of predefined master slave connection sets</td>
</tr>
<tr>
<td>6</td>
<td>Request for release of predefined master slave connection sets</td>
</tr>
<tr>
<td>7</td>
<td>Waiting for slave request after request for release of predefined master slave connection sets</td>
</tr>
<tr>
<td>8</td>
<td>Initialization of internal I/O configuration structures</td>
</tr>
<tr>
<td>9</td>
<td>Request for allocation of I/O configuration structures</td>
</tr>
<tr>
<td>10</td>
<td>Waiting for slave request after request for allocation of I/O configuration structures</td>
</tr>
<tr>
<td>11</td>
<td>Request for release of I/O configuration structures</td>
</tr>
<tr>
<td>12</td>
<td>Waiting for slave request after request for release of I/O configuration structures</td>
</tr>
<tr>
<td>13</td>
<td>Request for length of the connection consumed by the slave (output configuration)</td>
</tr>
<tr>
<td>14</td>
<td>Waiting for slave request after request for length of the connection consumed by the slave</td>
</tr>
<tr>
<td>15</td>
<td>Comparison of actual output configuration received by the slave with the internal nominal output configuration</td>
</tr>
<tr>
<td>16</td>
<td>Request for length of the connection produced by the slave (input configuration)</td>
</tr>
<tr>
<td>17</td>
<td>Waiting for slave request after request for length of the connection produced by the slave</td>
</tr>
<tr>
<td>18</td>
<td>Comparison of actual input configuration received by the slave with the internal nominal input configuration</td>
</tr>
<tr>
<td>19</td>
<td>Configuration and registration of I/O connection structures</td>
</tr>
<tr>
<td>20</td>
<td>Setting the expected packet rate in the slave</td>
</tr>
<tr>
<td>21</td>
<td>Waiting for slave request after setting the expected packet rate</td>
</tr>
<tr>
<td>22</td>
<td>Sending the first I/O poll request to the slave</td>
</tr>
<tr>
<td>23</td>
<td>Waiting for slave request after I/O poll request</td>
</tr>
<tr>
<td>24</td>
<td>Sending the second I/O poll request to the slave</td>
</tr>
<tr>
<td>25</td>
<td>Waiting for slave request after I/O poll request</td>
</tr>
<tr>
<td>26</td>
<td>Sending the third I/O poll request to the slave</td>
</tr>
<tr>
<td>27</td>
<td>Waiting for slave request after I/O poll request</td>
</tr>
<tr>
<td>28</td>
<td>Transmission of Heartbeat timeout to slave</td>
</tr>
<tr>
<td>29</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>First opening of connectionless communication</td>
</tr>
<tr>
<td>31</td>
<td>Waiting for slave request after opening the connectionless communication</td>
</tr>
<tr>
<td>32</td>
<td>Second opening of connectionless communication</td>
</tr>
<tr>
<td>33</td>
<td>Waiting for slave request after opening the connectionless communication</td>
</tr>
</tbody>
</table>
### DEV_STATE

<table>
<thead>
<tr>
<th>DEV_STATE</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>Closing the connectionless communication</td>
</tr>
<tr>
<td>35</td>
<td>Waiting for slave request after closing the connectionless communication</td>
</tr>
<tr>
<td>36</td>
<td>Enabling of all established connections</td>
</tr>
<tr>
<td>37</td>
<td>Waiting for slave request after enabling of all established connections</td>
</tr>
<tr>
<td>38</td>
<td>Opening the connectionless user-specific communication</td>
</tr>
<tr>
<td>39</td>
<td>Waiting for slave request after opening the connectionless user-specific communication</td>
</tr>
<tr>
<td>40</td>
<td>Request for allocation of user-specific predefined master slave connections</td>
</tr>
<tr>
<td>41</td>
<td>Waiting for slave request after request for allocation of user-specific master slave connections</td>
</tr>
<tr>
<td>42</td>
<td>Closing the connectionless user-specific communication</td>
</tr>
<tr>
<td>43</td>
<td>Waiting for slave request after request for closing the user-specific master slave connections</td>
</tr>
<tr>
<td>44</td>
<td>Reading or writing user-specific attribute</td>
</tr>
<tr>
<td>45</td>
<td>Waiting for request after reading or writing the user-specific attribute</td>
</tr>
<tr>
<td>46</td>
<td>Sending or waiting for request after fragmentally reading or writing attributes</td>
</tr>
</tbody>
</table>

### GEN_ERR

The output GEN_ERR is only valid for the communication error ONL_ERR = 35 (slave rejects requested command with error message). In this case, GEN_ERR outputs additional information about the occurred error in accordance to the DeviceNet specification. The following table shows the possible values and their meanings.

<table>
<thead>
<tr>
<th>ONL_ERR</th>
<th>GEN_ERR</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>Resource not available</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Service not supported</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Invalid attribute value</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Request mode already active</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Object status conflict</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>Attribute cannot be changed</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Privilege violation</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>Device status conflict</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>Response data too long</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>Insufficient data</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>Attribute not supported</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>Too much data</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>Object does not exist</td>
</tr>
</tbody>
</table>

35: Slave rejects requested command with error message
READING THE DEVICENET COUPLER STATUS

DNM_STAT outputs the DeviceNet coupler status. The outputs provide information about the communication state and error events.

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STAT_EN</td>
<td>EN</td>
<td>ERR</td>
</tr>
<tr>
<td>DNM_STAT</td>
<td>STATE_BITS</td>
<td>STAT_STATE_BITS</td>
</tr>
<tr>
<td>CONO</td>
<td>STATE_BITS</td>
<td>STAT_STATE_BITS</td>
</tr>
<tr>
<td>DNM_STAT</td>
<td>COM_ERR</td>
<td>STAT_COM_ERR</td>
</tr>
<tr>
<td>BUS_ERR</td>
<td>BUS_OFF</td>
<td>STAT_BUS_ERR</td>
</tr>
<tr>
<td>BUS_OFF</td>
<td>BUS_OFF</td>
<td>STAT_BUS_OFF</td>
</tr>
</tbody>
</table>
```

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DNM_STAT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>STATE_BITS</td>
<td>DNM_STATE_BITS_TYPE</td>
<td>Atypical communication states</td>
</tr>
<tr>
<td>DNM_STATE</td>
<td>BYTE</td>
<td>General state of the DeviceNet master</td>
</tr>
<tr>
<td>COM_ERR</td>
<td>DNM_COM_ERR_TYPE</td>
<td>Communication error</td>
</tr>
<tr>
<td>BUS_ERR</td>
<td>WORD</td>
<td>Number of bus failures</td>
</tr>
<tr>
<td>BUS_OFF</td>
<td>WORD</td>
<td>Number of bus outages</td>
</tr>
</tbody>
</table>

Description

The block DNM_STAT outputs the current DeviceNet coupler status.

DNM_STAT is active if input EN = TRUE. If the block is active, always the current values are displayed at the outputs.

```
<table>
<thead>
<tr>
<th>EN</th>
<th>BOOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are available at the outputs.</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>CONO</th>
<th>BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>The slot of the coupler (module number) whose state is to be read is specified at input CONO. The module number depends on the PLC category.</td>
<td></td>
</tr>
</tbody>
</table>
```

```
| 07 SL 97 R015X | CONO = 1 |
| 07 SL 97 R01X5 | CONO = 2 |
```

```
<table>
<thead>
<tr>
<th>ERR</th>
<th>INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR outputs an error ID, if an invalid value was applied to a block input. ERR has always to be considered in conjunction with the input EN. The ERR output value is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The encoding of the error messages available at ERR applies to all DeviceNet blocks and is explained in section &quot;ERR – Error messages of the DeviceNet blocks&quot;.</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>STATE_BITS</th>
<th>DNM_STATE_BITS_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_BITS outputs atypical communication states of the DeviceNet coupler. STATE_BITS is only valid if EN = TRUE and ERR = 0.</td>
<td></td>
</tr>
</tbody>
</table>
```

The format of the structure of the type `DNM_STATE_BITS_TYPE` is defined in the DeviceNet library (see description below).
**READING THE DEVCICNET COUPLER STATUS**

**DNM_STATE**

DNM_STATE outputs the general communication state of the DeviceNet master. The following states are defined:

- **OFFLINE** 00 HEX / 00 DEC
- **STOP** 40 HEX / 64 DEC
- **CLEAR** 80 HEX / 128 DEC
- **OPERATE** C0 HEX / 192 DEC

**DNM_STATE = OFFLINE**

If DNM_STATE is set to OFFLINE, the DeviceNet coupler performs an initialization. After the initialization phase is completed the coupler changes to STOP state.

**DNM_STATE = STOP**

The coupler is completely initialized, if DNM_STATE has the value STOP. In this state the coupler is ready to receive configuration data. There is no data exchange with the slaves. The coupler is set to this state if no user program is running.

**DNM_STATE = CLEAR**

If the user program is started, the coupler changes from STOP into CLEAR and starts to establish the connections defined during configuration. After the setup has been successfully completed, the coupler changes into OPERATE state. If an error occurs during parameterization, the coupler changes back to STOP state.

**DNM_STATE = OPERATE**

Normally the coupler is in OPERATE state while an user program is running. In this state the master exchanges I/O data with the slaves. If an error occurs during this process and ‘Auto Clear Mode’ was selected while configuring, the coupler changes back to CLEAR state and tries again to establish the connections. If ‘Auto Clear Mode’ was not selected, the coupler remains (in case of an error) in OPERATE state. If the user program is stopped, the coupler also changes back to STOP state.

DNM_STATE is only valid if EN = TRUE and ERR = 0.

**COM_ERR**

COM_ERR outputs possible communication errors. COM_ERR is only valid if EN = TRUE and ERR = 0.

**BUS_ERR**

BUS_ERR outputs the number of occurred bus failures. A bus failure occurs if the internal error frame counter exceeds a specific value. BUS_ERR is only valid if EN = TRUE and ERR = 0.

**BUS_OFF**

BUS_OFF outputs how often the coupler was excluded from the bus activities. An exclusion from the bus activities will occur when the internal error frame counter is overflowed. The coupler is automatically re-initialized after each overflow. BUS_OFF is only valid if EN = TRUE and ERR = 0.

Note: The inputs and outputs can neither be duplicated nor inverted.
Function call in IL

CAL STAT
(EN := STAT_EN,
CONO := STAT_CONO)
LD STAT.ERR
ST STAT.ERR
LD STAT.STATE_BITS
ST STAT.STATE_BITS
LD STAT.DNM_STATE
ST STAT.DNM_STATE
LD STAT.COM_ERR
ST STAT.COM_ERR
LD STAT.BUS_ERR
ST STAT.BUS_ERR
LD STAT.BUS_OFF
ST STAT.BUS_OFF

Note:
In IL, the function call has to be performed in one line.

Function call in ST

STAT
(EN := STAT.EN,
CONO := STAT_CONO);
STAT.ERR := STAT.ERR;
STAT.STATE_BITS := STAT.STATE_BITS;
STAT.DNM_STATE := STAT.DNM_STATE;
STAT.COM_ERR := STAT.COM_ERR;
STAT.BUS_ERR := STAT.BUS_ERR;
STAT.BUS_OFF := STAT.BUS_OFF;

STATE_BITS DNM_STATE_BITS_TYPE

The structure STATE_BITS consists of six boolean variables which display different communication states. Within the DeviceNet library the data type DNM_STATE_BITS_TYPE is declared as follows:

TYPE DNM_STATE_BITS_TYPE:
STRUCT
  CTRL: BOOL;
  AUTO_CLR: BOOL;
  NO_EXCH: BOOL;
  FATAL: BOOL;
  DUP_MAC_CHK: BOOL;
  DUP_MAC_DET: BOOL;
END_STRUCT
END_TYPE

CTRL BOOL
If this bit is set to TRUE, a parameterization error occurred. In normal operation CTRL should be FALSE. If not, the parameter and configuration data have to be checked.

AUTO CLR BOOL
If AUTO CLR is set to TRUE, the coupler has stopped the data exchange with all slaves due to communication errors and has changed back to CLEAR state (see DNM_STATE).

NO_EXCH BOOL
This bit is set to TRUE, if exchanging process data with one or several slaves is not possible. The error cause can be found in the configuration data as well as in the slaves.

FATAL BOOL
If FATAL is set to TRUE, no communication via DeviceNet is possible due to a fatal bus error (e.g. bus line short circuit). In this case, all bus lines have to be checked.

DUP_MAC_CHK BOOL
If DUP_MAC_CHK = TRUE, the coupler checks the bus for duplicate MAC IDs. The check is completed when the coupler detects other DeviceNet devices on the bus whose MAC IDs can be checked and no duplicate MAC ID was found. The completion of the check is indicated by DUP_MAC_CHK = FALSE.

DUP_MAC_DET BOOL
If DUP_MAC_DET = TRUE, the coupler has detected another DeviceNet device with the same MAC ID on the bus. In this case, the MAC IDs of all devices on the bus have to be checked.
COM_ERR DNM_COM_ERR_TYPE

Communication errors can be located more precisely using COM_ERR. The output COM_ERR is represented as a structure of the type DNM_COM_ERR_TYPE. Within the DeviceNet library this data type is declared as follows:

```
TYPE COM_ERR_TYPE:
  STRUCT
    ADDRESS: BYTE;
    EVENT:  BYTE;
  END_STRUCT
END_TYPE
```

ADDRESS BYTE

If an error occurs, ADDRESS contains the bus address of the faulty device (0 to 63). If ADDRESS has the value 255, the coupler itself causes the error.

EVENT BYTE

EVENT displays the error cause. The following tables show the encoding of the various errors.

ADDRESS = 255 Coupler error

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>Unknown process data handshake mode</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>53</td>
<td>Invalid data transmission rate</td>
<td>Configuration</td>
<td>Check data transmission rate specified in configuration data</td>
</tr>
<tr>
<td>54</td>
<td>Invalid MAC ID</td>
<td>Configuration</td>
<td>Check MAC ID specified in configuration data</td>
</tr>
<tr>
<td>57</td>
<td>Duplicate MAC ID found on the bus</td>
<td>Configuration / another device</td>
<td>Check MAC ID of all devices specified in configuration data</td>
</tr>
<tr>
<td>58</td>
<td>No device entry in current configuration data</td>
<td>Configuration</td>
<td>Error during loading the configuration data; contact Support</td>
</tr>
<tr>
<td>210</td>
<td>No configuration data</td>
<td>Configuration / Coupler</td>
<td>Load configuration data into coupler</td>
</tr>
<tr>
<td>212</td>
<td>Error during reading database</td>
<td>Configuration / Coupler</td>
<td>Contact Support</td>
</tr>
<tr>
<td>220</td>
<td>Watchdog error</td>
<td>Control system</td>
<td>Contact Support</td>
</tr>
<tr>
<td>221</td>
<td>No process data handshake</td>
<td>Control system</td>
<td>Contact Support</td>
</tr>
</tbody>
</table>

ADDRESS = 0..63 Error in device with bus address ADDRESS

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Slave monitoring failed after slave was already operational</td>
<td>Slave</td>
<td>Check that the slave is switched on and runs</td>
</tr>
<tr>
<td>30</td>
<td>Slave access timeout</td>
<td>Slave</td>
<td>Slave is not responding; check baud rate and MAC ID of the slave</td>
</tr>
<tr>
<td>32</td>
<td>Slave refuses access with error ID</td>
<td>Slave</td>
<td>Get slave diagnostics to evaluate the error ID</td>
</tr>
<tr>
<td>35</td>
<td>Slave responds with connection error during allocation phase</td>
<td>Slave</td>
<td>Get slave diagnostics to evaluate the extended error ID</td>
</tr>
<tr>
<td>EVENT</td>
<td>Meaning</td>
<td>Error source</td>
<td>Cause / Remedy</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>36</td>
<td>Actual length of produced connection (input data from the master's point of view) differs from the configured length</td>
<td>Slave / Configuration</td>
<td>Get slave diagnostics to determine the actual length</td>
</tr>
<tr>
<td>37</td>
<td>Actual length of consumed connection (output data from the master's point of view) differs from the configured length</td>
<td>Slave / Configuration</td>
<td>Get slave diagnostics to determine the actual length</td>
</tr>
<tr>
<td>38</td>
<td>Unknown and unprocessed slave response to service telegram</td>
<td>Slave / Coupler</td>
<td>Get slave diagnostics to determine the actual length</td>
</tr>
<tr>
<td>39</td>
<td>Connection already requested</td>
<td>Slave</td>
<td>Connection is terminated automatically</td>
</tr>
<tr>
<td>40</td>
<td>Number of CAN message data bytes is not 4 when reading the length of the produced or consumed connection</td>
<td>Slave</td>
<td>Slave behavior does not conform to the standard, communication with slave not possible</td>
</tr>
<tr>
<td>41</td>
<td>Predefined master slave connection already exists</td>
<td>Slave / Coupler</td>
<td>Connection is terminated automatically</td>
</tr>
<tr>
<td>42</td>
<td>Slave polling request length differs from the produced connection length</td>
<td>Slave</td>
<td>-</td>
</tr>
<tr>
<td>43</td>
<td>Sequence error in slave polling response</td>
<td>Slave</td>
<td>The first two segments in multiplex transfer were received</td>
</tr>
<tr>
<td>44</td>
<td>Fragmentation error in slave polling response</td>
<td>Slave</td>
<td>Fragmentation counter differs from the expected value in multiplex transfer</td>
</tr>
<tr>
<td>45</td>
<td>Sequence error in slave polling response</td>
<td>Slave</td>
<td>Middle or last segment was received prior to first segment</td>
</tr>
<tr>
<td>46</td>
<td>Length of slave bit strobe response differs from the produced connection length</td>
<td>Slave</td>
<td>-</td>
</tr>
<tr>
<td>47</td>
<td>Sequence error in state change response or cyclical response of slave</td>
<td>Slave</td>
<td>The first two segments in multiplex transfer were received</td>
</tr>
<tr>
<td>48</td>
<td>Fragmentation error in state change response or cyclical response of slave</td>
<td>Slave</td>
<td>Fragmentation counter differs from the expected value in multiplex transfer</td>
</tr>
<tr>
<td>49</td>
<td>Sequence error in state change response or cyclical response of slave</td>
<td>Slave</td>
<td>Middle or last segment was received prior to first segment</td>
</tr>
<tr>
<td>51</td>
<td>Length of state change response or cyclical response of the slave differs from the produced connection length</td>
<td>Slave</td>
<td>-</td>
</tr>
<tr>
<td>52</td>
<td>UCMM group not supported</td>
<td>Slave</td>
<td>Change UCMM group</td>
</tr>
<tr>
<td>59</td>
<td>Actual configuration data contain duplicate device addresses</td>
<td>Configuration</td>
<td>Each device must have an unique MAC ID; edit configuration data</td>
</tr>
<tr>
<td>60</td>
<td>Invalid length indicator of a slave data set</td>
<td>Configuration</td>
<td>Error loading the configuration data; contact Support</td>
</tr>
<tr>
<td>61</td>
<td>Additional table of predefined master slave connections has invalid length</td>
<td>Configuration</td>
<td>Error loading the configuration data; contact Support</td>
</tr>
<tr>
<td>62</td>
<td>Invalid length indicator of the master slave I/O configuration table</td>
<td>Configuration</td>
<td>Error loading the configuration data; contact Support</td>
</tr>
<tr>
<td>63</td>
<td>Predefined master slave I/O configuration differs from the additional table</td>
<td>Configuration</td>
<td>Different number of I/O modules and configured offset addresses; contact Support</td>
</tr>
<tr>
<td>64</td>
<td>Invalid length indicator of the parameter data set</td>
<td>Configuration</td>
<td>Too low length indicator; contact Support</td>
</tr>
<tr>
<td>65</td>
<td>Number of declared inputs in the additional table differs from the number defined in the I/O configuration</td>
<td>Configuration</td>
<td>Each I/O configuration entry must have exactly one assigned entry in the additional table; contact Support</td>
</tr>
<tr>
<td>EVENT</td>
<td>Meaning</td>
<td>Error source</td>
<td>Cause / Remedy</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------------------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>66</td>
<td>Number of declared outputs in the additional table differs from the number defined in the I/O configuration</td>
<td>Configuration</td>
<td>Each I/O configuration entry must have exactly one assigned entry in the additional table; contact Support</td>
</tr>
<tr>
<td>67</td>
<td>I/O configuration contains unknown data type</td>
<td>Configuration</td>
<td>Only the data types BOOL, BYTE, WORD, DWORD and STRING are supported</td>
</tr>
<tr>
<td>68</td>
<td>Data type of a configured I/O module in a connection differs from the defined data length</td>
<td>Configuration</td>
<td>Valid data types and data lengths are: BOOL= 1 Byte UINT8=BYTE = 1 Byte UINT16=WORD = 2 Byte UINT32=DWORD= 4 Byte</td>
</tr>
<tr>
<td>69</td>
<td>Configured output addresses of a module (offset + length) exceed the maximum value 3584</td>
<td>Configuration</td>
<td>Process data image limited to 3584 bytes</td>
</tr>
<tr>
<td>70</td>
<td>Configured input addresses of a module (offset + length) exceed the maximum value 3584</td>
<td>Configuration</td>
<td>Process data image limited to 3584 bytes</td>
</tr>
<tr>
<td>71</td>
<td>Unknown predefined connection type</td>
<td>Configuration</td>
<td>Only the connection types cyclic, poll, state change and bit strobe are supported</td>
</tr>
<tr>
<td>72</td>
<td>Multiple parallel connections defined</td>
<td>Configuration</td>
<td>Only one connection type is supported per slave</td>
</tr>
<tr>
<td>73</td>
<td>Configured expected packet rate of a connection is lower than the production inhibit time</td>
<td>Configuration</td>
<td>Expected packet rate must be greater than production inhibit time</td>
</tr>
</tbody>
</table>
The block DNM_SYSDIAG outputs a bit field as state survey about all slaves (clients) at output SLV. Three different surveys can be selected via input TYP.

### Block type

Function block without historical values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>DNM_SYSDIAG</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE Slot (module number) of the coupler</td>
</tr>
<tr>
<td>TYP</td>
<td>BYTE Selection of the survey type</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD Error message</td>
</tr>
<tr>
<td>DEV</td>
<td>ARRAY [0..63] OF BOOL Status survey of the slaves</td>
</tr>
</tbody>
</table>

### Description

The block DNM_SYSDIAG outputs different status surveys about all slaves. Three survey types can be selected:

- configuration survey
- I/O data exchange survey
- diagnosis survey

**EN**

The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are available at the outputs.

**CONO**

The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

07 SL 97 R015X  \( \text{CONO} = 1 \)
07 SL 97 R01X5  \( \text{CONO} = 2 \)

**TYP**

The input TYP is used to select which status survey should be displayed at output DEV.

- **TYP = 1** configuration survey
- **TYP = 2** data exchange survey
- **TYP = 3** diagnosis survey

Output DEV displays the slaves which were successfully connected to the master. Please note that the master establishes only a connection to those slaves which were assigned to the master when defining the configuration data.

Error message: ERR outputs an error ID, if an invalid value is applied to an input. ERR has always to be considered in conjunction with the input EN. The ERR output value is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The encoding of the error messages available at ERR applies to all DeviceNet blocks and is explained in section "ERR – Error messages of the DeviceNet blocks".
DEV ARRAY [0..63] OF BOOL

DEV outputs the status survey as a bit field. Every bit within this field represents a slave. The index corresponds to the slave's bus address (MAC ID). A bit which is set to TRUE means that the state selected using TYP is applicable for the corresponding slave.

If e.g. TYP = 1 is selected and SLV[2] = TRUE, the slave with the MAC ID 2 was successfully configured by the master.
If SLV[2] = FALSE, the configuration of the specific slave has not yet been completed or the slave is not part of the master's configuration data.

If TYP = 2 and SLV[2] = TRUE, the master exchanges I/O data with the slave containing the MAC ID 2. However, if SLV[2] = FALSE, the master currently does not exchange I/O data with the slave. The master can only exchange data with such slaves which were previously brought into operation by the master.

If TYP = 3 and e.g. SLV[2] = TRUE, the slave with the MAC ID 2 reports a diagnosis. The detailed diagnosis can then be requested using the block DNM_DEVDIAG.

The bit field obtained at DEV is only valid if EN = TRUE and ERR = 0.

Note:
The inputs and outputs can neither be duplicated nor inverted.

Function call in IL
CAL SYSDIAG
(EN := SYSDIAG_EN,
CONO := SYSDIAG_CONO,
TYP := SYSDIAG_TYP)
LD SYSDIAG.ERR
ST SYSDIAG.ERR
LD SYSDIAG.DEV
ST SYSDIAG.DEV

Note:
In IL, the function call has to be performed in one line.

Function call in ST
SYSDIAG
(EN := SYSDIAG_EN,
CONO := SYSDIAG_CONO,
TYP := SYSDIAG_TYP);
SYSDIAG_ERR := SYSDIAG.ERR;
SYSDIAG_DEV := SYSDIAG.DEV;
READING AN ATTRIBUTE FROM A SLAVE OBJECT

The block DNM_GET_ATTR can be used to read individual attributes from object instances of a slave.

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>DNM_GET_ATTR</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>MAC_ID</td>
<td>BYTE</td>
<td>MAC_ID of the relevant slave</td>
</tr>
<tr>
<td>CLASS_ID</td>
<td>BYTE</td>
<td>Object class ID of the object</td>
</tr>
<tr>
<td>INST_ID</td>
<td>BYTE</td>
<td>Object instance number</td>
</tr>
<tr>
<td>ATTR_ID</td>
<td>BYTE</td>
<td>Attribute number within the object</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address from which on the read data are stored (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of read data (byte value)</td>
</tr>
</tbody>
</table>

Description

The block DNM_GET_ATTR can be used to read individual attributes from slave objects.

Every time a FALSE → TRUE edge is applied to the input EN, DNM_GET_ATTR reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

The DeviceNet standard describes an abstract object model (see also system technology of internal couplers – DeviceNet coupler). Here different standard classes are defined which represent a kind of type definitions for the objects. In addition further manufacturer-specific classes can be defined for a DeviceNet module.

Objects are instances of the relevant class. These objects contain attributes which can be read using the block DNM_GET_ATTR.

Some objects must be available in every DeviceNet device by default. Normally, these standard objects are not accessed by the user program during running operation, even though this is possible in principle. The accesses are usually limited to optional additional objects or their attributes. These additional slave attributes can be displayed in 907 FB 1131 by selecting Device configuration | Parameter data (see 907 FB 1131 documentation). This view displays a list of available attributes for the specific slave. Each entry contains a sequence number, the object class ID (Class ID), the instance number (Instance ID), the number of the individual attribute (Attribute ID), the data type of the attribute, the valid access types (read/write), an attribute description and the valid limits. For the function block DNM_GET_ATTR, this information can be used to select the required CLASS_ID, INST_ID and ATTR_ID. In addition, it can be determined which type the variable at the block input DATA must have (via ADR operator) in order to be able to store and interpret the received data.
EN

BOOL

For a FALSE → TRUE edge at input EN, all further inputs are applied.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

As the block execution requires a bus access, the data are available in the next cycle at the earliest after activating the block.

CONO

BYTE

The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

07 SL 97 R015X  CONO = 1
07 SL 97 R01X5  CONO = 2

MAC_ID

BYTE

At input MAC_ID, the MAC ID of the slave is specified of which an attribute is to be requested.

CLASS_ID

BYTE

At input CLASS_ID, the class ID of the object is specified which contains the attribute to be read (compare to Class entry in Parameter data list in 907 FB 1131).

INST_ID

BYTE

At input INST_ID, the instance number of the object is specified which contains the attribute to be read (compare to Inst. entry in Parameter data list in 907 FB 1131).

ATTR_ID

BYTE

At input ATTR_ID, the number of the attribute is specified, which should be read within the object containing the attribute (compare to Attr. entry in Parameter data list in 907 FB 1131).

DATA

DWORD

At input DATA, the address of the variable is specified, to which the received attribute data are to be written (via the ADR operator). For the data format refer to the Type entry in the Parameter data list in 907 FB 1131. The received data are only valid if DONE = TRUE, ERR = 0 and DATA_LEN > 0.

DONE

BOOL

The output DONE reflects the processing state. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE for one cycle period.

The values at the block outputs are only valid if DONE = TRUE.

ERR

WORD

ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all DeviceNet blocks and is explained in section "ERR – Error messages of the DeviceNet blocks".

DATA_LEN

BYTE

The length of the received attribute data (byte value) are output at DATA_LEN after the procedure was successfully completed. As the block execution requires a bus access, the data are available in the next cycle at the earliest after activating the block. The value of this input is only valid for DONE = TRUE and ERR = 0.

Note:
The inputs and outputs can neither be duplicated nor inverted.
READING AN ATTRIBUTE FROM A SLAVE OBJECT

**DNM_GET_ATTR**

### Function call in IL

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD GET_DATA</td>
<td></td>
</tr>
<tr>
<td>ADR</td>
<td></td>
</tr>
<tr>
<td>ST ADR_GET_DATA</td>
<td></td>
</tr>
<tr>
<td>CAL GET</td>
<td></td>
</tr>
<tr>
<td>(EN := GET_EN,</td>
<td></td>
</tr>
<tr>
<td>CONO := GET_CONO,</td>
<td></td>
</tr>
<tr>
<td>MAC_ID := GET_MAC_ID,</td>
<td></td>
</tr>
<tr>
<td>CLASS_ID := GET_CLASS_ID,</td>
<td></td>
</tr>
<tr>
<td>INST_ID := GET_INST_ID,</td>
<td></td>
</tr>
<tr>
<td>ATTR_ID := GET_ATTR_ID,</td>
<td></td>
</tr>
<tr>
<td>DATA := ADR_GET_DATA)</td>
<td></td>
</tr>
<tr>
<td>LD GET_DONE</td>
<td></td>
</tr>
<tr>
<td>ST GET_DONE</td>
<td></td>
</tr>
<tr>
<td>LD GET_ERR</td>
<td></td>
</tr>
<tr>
<td>ST GET_ERR</td>
<td></td>
</tr>
<tr>
<td>LD GET_DATA_LEN</td>
<td></td>
</tr>
<tr>
<td>ST GET_DATA_LEN</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
In IL, the function call has to be performed in one line.

### Function call in ST

```
GET
(EN := GET_EN,
CONO := GET_CONO,
MAC_ID := GET_MAC_ID,
CLASS_ID := GET_CLASS_ID,
INST_ID := GET_INST_ID,
ATTR_ID := GET_ATTR_ID,
DATA := ADR(GET_DATA));

GET_DONE := GET_DONE;
GET_ERR := GET_ERR;
GET_DATA_LEN := GET_DATA_LEN;
```
The block DNM_RESET_OBJ can be used to reset a slave object.

### Block type

Function block with historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DNM_RESET_OBJ</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>MAC_ID</td>
<td>BYTE</td>
<td>MAC_ID of the relevant slave</td>
</tr>
<tr>
<td>CLASS_ID</td>
<td>BYTE</td>
<td>Object class ID of the object</td>
</tr>
<tr>
<td>INST_ID</td>
<td>BYTE</td>
<td>Object instance number</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

### Description

The block DNM_RESET_OBJ can be used to reset an instance of a class (object) in a slave.

Every time a FALSE $\rightarrow$ TRUE edge is applied to the input EN, DNM_RESET_OBJ reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE $\rightarrow$ TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

The DeviceNet standard describes an abstract object model (see also system technology of internal couplers – DeviceNet coupler). Here different standard classes are defined which represent a kind of type definitions for the objects. In addition further manufacturer-specific classes can be defined for a DeviceNet module. Objects are instances of the relevant class. These objects can then be reset using the block DNM_RESET.

Some objects must be available in every DeviceNet device by default. Normally, these standard objects are not accessed by the user program during running operation, even though this is possible in principle. The accesses are usually limited to optional additional objects or their attributes. These additional slave objects/attributes can be displayed in 907 FB 1131 by selecting Device configuration | Parameter data (see 907 FB 1131 documentation). This view displays a list of available attributes for the specific slave. Each entry contains a sequence number, the object class ID (Class ID), the instance number (Instance ID), the number of the individual attribute (Attribute ID), the data type of the attribute, the valid access types (read/write), an attribute description and the valid limits. For the function block DNM_RESET_OBJ, this information can be used to select the required CLASS_ID and INST_ID.
**EN**

For a FALSE $\rightarrow$ TRUE edge at input EN, all further inputs are applied.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

**CONO**

The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

07 SL 97 R015X  CONO = 1
07 SL 97 R01X5  CONO = 2

**MAC_ID**

At input MAC_ID, the MAC ID of the slave is specified of which an object is to be reset.

**CLASS_ID**

At input CLASS_ID, the class ID of the object to be reset is specified (compare to Class entry in Parameter data list in 907 FB 1131).

**INST_ID**

At input INST_ID, the instance number of the object to be reset is specified (compare to Inst. entry in Parameter data list in 907 FB 1131).

**DONE**

The output DONE reflects the processing state. If the block is triggered by a FALSE $\rightarrow$ TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE for one cycle period.

The values at the block outputs are only valid if DONE = TRUE.

**ERR**

ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all DeviceNet blocks and is explained in section "ERR – Error messages of the DeviceNet blocks".

Note:
The inputs and outputs can neither be duplicated nor inverted.

**Function call in IL**

```
CAL RESET
(EN := RESET_EN,
 CONO := RESET_CONO,
 MAC_ID := RESET_MAC_ID,
 CLASS_ID := RESET_CLASS_ID),
LD RESET_DONE
ST RESET_DONE
LD RESET_ERR
ST RESET_ERR
```

Note:
In IL, the function call has to be performed in one line.

**Function call in ST**

```
RESET
(EN := RESET_EN,
 CONO := RESET_CONO,
 MAC_ID := RESET_MAC_ID,
 CLASS_ID := RESET_CLASS_ID,
 INST_ID := RESET_INST_ID),
RESET_DONE := RESET_DONE;
RESET_ERR := RESET_ERR;
```
WRITING AN ATTRIBUTE TO A SLAVE OBJECT

The block DNM_SET_ATTR can be used to write individual attributes of object instances in a slave.

Block type
- Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>DNM_SET_ATTR</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>MAC_ID</td>
<td>BYTE</td>
<td>MAC_ID of the relevant slave</td>
</tr>
<tr>
<td>CLASS_ID</td>
<td>BYTE</td>
<td>Object class ID of the object</td>
</tr>
<tr>
<td>INST_ID</td>
<td>BYTE</td>
<td>Object instance number</td>
</tr>
<tr>
<td>ATTR_ID</td>
<td>BYTE</td>
<td>Attribute number within the object</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address from which on the data to be written are stored (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

Description

The block DNM_SET_ATTR can be used to write individual attributes in slave objects.

Every time a FALSE → TRUE edge is applied to the input EN, DNM_SET_ATTR reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

The DeviceNet standard describes an abstract object model (see also system technology of internal couplers – DeviceNet coupler). Here different standard classes are defined which represent a kind of type definitions for the objects. In addition further manufacturer-specific classes can be defined for a DeviceNet module.

Objects are instances of the relevant class. These objects contain attributes which can be written using the block DNM_SET_ATTR.

Some objects must be available in every DeviceNet device by default. Normally, these standard objects are not accessed by the user program during running operation, even though this is possible in principle. The accesses are usually limited to optional additional objects or their attributes. These additional slave attributes can be displayed in 907 FB 1131 by selecting Device configuration | Parameter data (see 907 FB 1131 documentation). This view displays a list of available attributes for the specific slave. Each entry contains a sequence number, the object class ID (Class ID), the instance number (Instance ID), the number of the individual attribute (Attribute ID), the data type of the attribute, the valid access types (read/write), an attribute description and the valid limits. For the function block DNM_SET_ATTR, this information can be used to select the required CLASS_ID, INST_ID and ATTR_ID. In addition, it can be determined which type the variable at the block input DATA must have (via ADR operator) and which value must be assigned to this variable in order to get the required result.
EN BOOL
For a FALSE \rightarrow TRUE edge at input EN, all further inputs are applied.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

CONO BYTE
The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

07 SL 97 R015X CONO = 1
07 SL 97 R01X5 CONO = 2

MAC_ID BYTE
At input MAC_ID, the MAC ID of the slave is specified of which an object is to be written.

CLASS_ID BYTE
At input CLASS_ID, the class ID of the object, which contains the attribute to be written, is specified (compare to Class entry in Parameter data list in 907 FB 1131).

INST_ID BYTE
At input INST_ID, the instance number of the object, which contains the attribute to be written, is specified (compare to Inst. entry in Parameter data list in 907 FB 1131).

ATTR_ID BYTE
At input ATTR_ID, the number of the attribute to be written within the object, which contains the attribute, is specified (compare to Attr. entry in Parameter data list in 907 FB 1131).

DATA DWORD
At input DATA, the address of the variable is specified which contains the attribute data to be transmitted (via ADR operator. For the data format refer to the Type entry in the Parameter data list in 907 FB 1131.

DATA_LEN BYTE
At input DATA_LEN, the length of the data to be transmitted stored in the variable at address DATA is specified as a byte value.

DONE BOOL
The output DONE reflects the processing state. If the block is trigged by a FALSE \rightarrow TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE for one cycle period.

The values at the block outputs are only valid if DONE = TRUE.

ERR WORD
ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all DeviceNet blocks and is explained in section "ERR – Error messages of the DeviceNet blocks".

Note:
The inputs and outputs can neither be duplicated nor inverted.
Function call in IL

LD SET_DATA
ADR
ST ADR_SET_DATA
CAL SET
(EN := SET_EN,
CONO := SET_CONO,
MAC_ID := SET_MAC_ID,
CLASS_ID := SET_CLASS_ID,
INST_ID := SET_INST_ID,
ATTR_ID := SET_ATTR_ID,
DATA := SET_DATA,
DATA_LEN := SET_DATA_LEN)
LD SET_DONE
ST SET_DONE
LD SET_ERR
ST SET_ERR

Note:
In IL, the function call has to be performed in one line.

Function call in ST

SET
(EN := SET_EN,
CONO := SET_CONO,
MAC_ID := SET_MAC_ID,
CLASS_ID := SET_CLASS_ID,
INST_ID := SET_INST_ID,
ATTR_ID := SET_ATTR_ID,
DATA := ADR(SET_DATA),
DATA_LEN := SET_DATA_LEN);
SET_DONE := SET_DONE;
SET_ERR := SET_ERR;
Glossary

BOOL
Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT
DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 Bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

DWORD
DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

INT
INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

WORD
WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

STRING
Variables of the type STRING can include any character string. The specified value for the memory space allocation in the declaration corresponds to characters and can be defined using parenthesis or brackets. If no value (1 to 255) is specified, 80 characters are applied by default.

**Functions**

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures etc.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can be derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

**Function blocks**

Function blocks are subroutines which can have as many inputs, outputs and internal variables as required. They are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. This allows for example to realize counters which may not forget their counter value. i.e. function blocks can have an internal memory.

Functions and function blocks differ in two main points:

- A function block has multiple output parameters and a function has a maximum of one output parameter, where the output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

   For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

   For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

   The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
Components of the DeviceNet library 2

D
DeviceNet S90
  DEVNET_INFO 5
  DNM_DEVDIAG 7
  DNM_STAT 12
  DNM_SYSDIAG 18
  DEVICENET S90
    DEVICESTATUS_1_TYPE 9
    DNM_COM_ERR_TYPE 15
    DNM_GET_ATTR 20
    DNM_RESET_OBJ 23
    DNM_SET_ATTR 25
    DNM_STATE_BITS_TYPE 14
  DEVICESTATUS_1_TYPE 9
  DEVNET_INFO S90 5
  DNM_COM_ERR_TYPE 15
  DNM_DEVDIAG S90 7
  DNM_GET_ATTR S90 20
  DNM_RESET_OBJ S90 23
  DNM_SET_ATTR S90 25
  DNM_STAT S90 12
  DNM_STATE_BITS_TYPE 14
  DNM_SYSDIAG S90 18

E
Error messages of the DeviceNet blocks 3
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

Interbus
Function Block Library
90 Series
# Contents

<table>
<thead>
<tr>
<th>Interbus library</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL CHARACTERISTICS IN THE INTERBUS LIBRARY</td>
<td>2</td>
</tr>
<tr>
<td>COMPONENTS OF THE INTERBUS LIBRARY</td>
<td>3</td>
</tr>
<tr>
<td>Function blocks</td>
<td>3</td>
</tr>
<tr>
<td>Data types</td>
<td>3</td>
</tr>
<tr>
<td>ERR - ERROR MESSAGES OF THE INTERBUS BLOCKS</td>
<td>4</td>
</tr>
<tr>
<td>Errors at block inputs</td>
<td>4</td>
</tr>
<tr>
<td>Execution errors, coupler or slave responses</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overview of blocks arranged according to their call names</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>READING INFORMATION ABOUT THE INSTALLED INTERBUS COUPLERS</td>
<td>INTERBUS_INFO .... 6</td>
</tr>
<tr>
<td>POLLING DIAGNOSTIC DATA FROM A SLAVE</td>
<td>IBM_SLVDIAG .... 8</td>
</tr>
<tr>
<td>READING THE INTERBUS COUPLER STATUS</td>
<td>IBM_STAT .... 13</td>
</tr>
<tr>
<td>READING A STATUS SURVEY OF ALL SLAVES</td>
<td>IBM_SYSDIAG .... 22</td>
</tr>
<tr>
<td>READING AN OBJECT DESCRIPTION FROM A SLAVE SUPPORTING PCP</td>
<td>IBM_GET_OD .... 24</td>
</tr>
<tr>
<td>READING AN OBJECT FROM A SLAVE SUPPORTING PCP</td>
<td>IBM_READ .... 28</td>
</tr>
<tr>
<td>ENABLING THE READING OF AN OBJECT FROM THE CONTROL SYSTEM VIA A SLAVE SUPPORTING PCP</td>
<td>IBM_READ_EN .... 31</td>
</tr>
<tr>
<td>WRITING AN OBJECT TO A SLAVE SUPPORTING PCP</td>
<td>IBM_WRITE .... 34</td>
</tr>
<tr>
<td>ENABLING THE WRITING OF AN OBJECT BY A SLAVE SUPPORTING PCP</td>
<td>IBM_WRITE_EN .... 37</td>
</tr>
</tbody>
</table>

| Glossary | 40 |

| Index | 42 |
SPECIAL CHARACTERISTICS IN THE INTERBUS LIBRARY

Note:
Interbus communication is only performed in RUN mode of the PLC and not in simulation mode.

Prerequisites for use of the library:
The function blocks contained in the Interbus library access to both, the PLC run time system and directly to the coupler. The required definitions and functions are stored in the internal library Coupler_S90_V41.LIB.

When using the Interbus library Interbus_Master_S90_V43.LIB, it is absolutely essential to insert also the library Coupler_S90_V41.LIB into the project.
COMPONENTS OF THE INTERBUS LIBRARY

The library contains function blocks which allow an easy handling of the Interbus coupler. Additionally various data types are defined in this library. These structures enable a clear presentation of data sets.

Function blocks

The Interbus library contains the following function blocks:

<table>
<thead>
<tr>
<th>Group</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERBUS_INFO</td>
<td>- Reading information about installed couplers</td>
</tr>
</tbody>
</table>

Group | Interbus master

Diagnosis:

- IBM_SLVDIAG - Polling diagnostic data from a slave
- IBM_STAT - Reading the Interbus coupler status
- IBM_SYSDIAG - Displaying status information of all slaves

PCP:

- IBM_GET_OD - Reading object descriptions from a slave supporting PCP
- IBM_READ - Writing parameters to a slave supporting PCP
- IBM_READ_EN - Enabling the reading of parameters from the control system via a slave supporting PCP
- IBM_WRITE - Writing parameters to a slave supporting PCP
- IBM_WRITE_EN - Enabling the writing of parameters to the control system via a slave supporting PCP

Detailed information about the various blocks can be found in the following sections.

Data types

The following data types (structures) are defined in the Interbus library:

- IBM_COM_ERR_TYPE - Communication error
- IBM_STATE_BITS_TYPE - Bits for coupler state description
- IBM_DEVICESTATUS_1_TYPE - Slave diagnostics
- IBM_OBJ_DESC_TYPE - Object description

For detailed information about the different data types please refer to the respective description of the corresponding block.
ERR - ERROR MESSAGES OF THE INTERBUS BLOCKS

If the output ERR of an Interbus block is unequal to 0, either one of the values at the block inputs is invalid or an internal error occurred when executing the block. The error cause can be determined with the help of the following tables.

Errors at block inputs

<table>
<thead>
<tr>
<th>ERR</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>HEX</td>
</tr>
<tr>
<td>8193</td>
<td>2001</td>
</tr>
<tr>
<td>8194</td>
<td>2002</td>
</tr>
<tr>
<td>8195</td>
<td>2003</td>
</tr>
<tr>
<td>8196</td>
<td>2004</td>
</tr>
<tr>
<td>8197</td>
<td>2005</td>
</tr>
<tr>
<td>8198</td>
<td>2006</td>
</tr>
<tr>
<td>8201</td>
<td>2009</td>
</tr>
<tr>
<td>8202</td>
<td>200A</td>
</tr>
</tbody>
</table>

Execution errors, coupler or slave responses

<table>
<thead>
<tr>
<th>ERR</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>HEX</td>
</tr>
<tr>
<td>9281</td>
<td>2441</td>
</tr>
<tr>
<td>9283</td>
<td>2443</td>
</tr>
<tr>
<td>9345</td>
<td>2481</td>
</tr>
<tr>
<td>9346</td>
<td>2482</td>
</tr>
<tr>
<td>9347</td>
<td>2483</td>
</tr>
<tr>
<td>9351</td>
<td>2487</td>
</tr>
<tr>
<td>9367</td>
<td>2497</td>
</tr>
<tr>
<td>9371</td>
<td>249B</td>
</tr>
<tr>
<td>9377</td>
<td>24A1</td>
</tr>
</tbody>
</table>
Overview of blocks arranged according to their call names

Character description:

FBmV  ... Function block with historical values
FBoV  ... Function block without historical values
  F  ... Function

<table>
<thead>
<tr>
<th>CE name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM_GET_OD</td>
<td>FBmV</td>
<td>Reading object descriptions from a slave supporting PCP</td>
<td>24</td>
</tr>
<tr>
<td>IBM_READ</td>
<td>FBmV</td>
<td>Reading parameters from a slave supporting PCP</td>
<td>28</td>
</tr>
<tr>
<td>IBM_READ_EN</td>
<td>FBmV</td>
<td>Enabling the reading of parameters from the control system via a slave supporting PCP</td>
<td>31</td>
</tr>
<tr>
<td>IBM_SLVDIAG</td>
<td>FBmV</td>
<td>Polling diagnostic data from a slave</td>
<td>8</td>
</tr>
<tr>
<td>IBM_STAT</td>
<td>FBoV</td>
<td>Reading the Interbus coupler status</td>
<td>13</td>
</tr>
<tr>
<td>IBM_SYSDIAG</td>
<td>FBoV</td>
<td>Displaying status information of all slaves</td>
<td>22</td>
</tr>
<tr>
<td>IBM_WRITE</td>
<td>FBmV</td>
<td>Writing parameters to a slave supporting PCP</td>
<td>34</td>
</tr>
<tr>
<td>IBM_WRITE_EN</td>
<td>FBmV</td>
<td>Enabling the writing of parameters to the control system via a slave supporting PCP</td>
<td>37</td>
</tr>
<tr>
<td>INTERBUS_INFO</td>
<td>FBoV</td>
<td>Reading information about the installed couplers</td>
<td>6</td>
</tr>
</tbody>
</table>
READING INFORMATION ABOUT THE INSTALLED INTERBUS COUPLERS

The block INTERBUS_INFO outputs coupler related information. The following items are displayed: coupler type and model, operating mode, manufacturing date, device number and serial number as well as firmware designation and firmware version.

Block type

Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>INTERBUS_INFO</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>TYP</td>
<td>STRING(17)</td>
<td>Coupler type</td>
</tr>
<tr>
<td>MODEL</td>
<td>STRING(22)</td>
<td>Coupler model</td>
</tr>
<tr>
<td>OP_MODE</td>
<td>STRING(22)</td>
<td>Operating mode of the coupler</td>
</tr>
<tr>
<td>MAN_DATE</td>
<td>DATE</td>
<td>Manufacturing date of the coupler</td>
</tr>
<tr>
<td>DEV_NO</td>
<td>DWORD</td>
<td>Device number of the coupler</td>
</tr>
<tr>
<td>SER_NO</td>
<td>DWORD</td>
<td>Serial number of the coupler</td>
</tr>
<tr>
<td>FW_NAME</td>
<td>STRING(17)</td>
<td>Designation of the coupler firmware</td>
</tr>
<tr>
<td>FW_VER</td>
<td>STRING(17)</td>
<td>Version of the coupler firmware</td>
</tr>
</tbody>
</table>

Description

The block INTERBUS_INFO is always active. It reads the slot number at the block input CONO and outputs the corresponding information about the selected coupler at the outputs.

INTERBUS_INFO recognizes different coupler types. For that reason, the block outputs do not only relate to the Interbus coupler.

The block is not intended for usage in normal user programs. It should be mainly used to support error diagnostics and maintenance operations.

CONO BYTE

The slot of the coupler (module number) whose information are to be requested is specified at input CONO. The module number depends on the PLC category.

R019X CONO = 1
R01X9 CONO = 2

ERR WORD

ERR outputs an error ID, if an invalid value is applied to input CONO. The encoding of the error messages available at ERR applies to all Interbus blocks and is explained in section "ERR – Error messages of the Interbus blocks".

TYP STRING(17)

TYP outputs the coupler type (i.e. the coupler design). Different types are recognized by the block.

The Interbus coupler has the type TYP = 'single mode'. If a coupler type cannot be determined, the output TYP is set to 'unknown'.
READING INFORMATION ABOUT THE INSTALLED
INTERBUS COUPLERS

MODEL STRING(22)
MODEL outputs the transfer protocol supported by the
coupler as well as the subscriber type.
The model ID of the Interbus coupler is
MODEL = 'Interbus Master'.
If the block detects an unknown model, MODEL
outputs the value 'unknown'.

OP_MODE STRING(22)
OP_MODE outputs the current coupler operating mode
setting. This output is only applicable for couplers with
switchable operating modes.
Therefore, this output is not used for the Interbus
coupler.

MAN_DATE DATE
MAN_DATE outputs the manufacturing date of the
coupler. The date is a variable of the data type DATE
and has the format D#YYYY-MM-DD. The initial value
is D#2000-01-01.

DEV_NO DWORD
DEV_NO outputs the device number of the coupler.

SER_NO DWORD
SER_NO outputs the serial number of the coupler.

FW_NAME STRING(17)
FW_NAME outputs the name of the coupler firmware.

FW_VER STRING(17)
FW_VER outputs the release number (version) and
issue date of the coupler firmware. These data are
represented as a string (e.g., 'V1.003 15.07.00').
Note:
The inputs and outputs can neither be duplicated nor
inverted.

Function call in IL
CAL INFO(CONO := INFO_CONO)
LD INFO.ERR
ST INFO.ERR
LD INFO.TYP
ST INFO.TYP
LD INFO.MODEL
ST INFO.MODEL
LD INFO.OP_MODE
ST INFO.OP_MODE
LD INFO.MAN_DATE
ST INFO.MAN_DATE
LD INFO.DEV_NO
ST INFO.DEV_NO
LD INFO.SER_NO
ST INFO.SER_NO
LD INFO.FW_NAME
ST INFO.FW_NAME
LD INFO.FW_VER
ST INFO.FW_VER

Note:
In IL, the function call has to be performed in one
line.

Function call in ST
INFO(CONO := INFO_CONO);
INFO.ERR := INFO.ERR;
INFO.TYP := INFO.TYP;
INFO.MODEL := INFO.MODEL;
INFO.OP_MODE := INFO.OP_MODE;
INFO.MAN_DATE := INFO.MAN_DATE;
INFO.DEV_NO := INFO.DEV_NO;
INFO.SER_NO := INFO.SER_NO;
INFO.FW_NAME := INFO.FW_NAME;
INFO.FW_VER := INFO.FW_VER;
The block IBM_SLVDIAG reads the diagnostic data of a slave.

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>IBM_SLVDIAG</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>SLV</td>
<td>BYTE</td>
<td>Physical position of the slave on the bus</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>STAT_1</td>
<td>IBM_DEVICESTATUS_1_TYPE</td>
<td>Device status_1, device diagnostics</td>
</tr>
<tr>
<td>LEN_CODE</td>
<td>BYTE</td>
<td>Length code of the device</td>
</tr>
<tr>
<td>IDENT_CODE</td>
<td>BYTE</td>
<td>ID code of the device</td>
</tr>
<tr>
<td>NUM_CRC_ERR</td>
<td>WORD</td>
<td>Number of CRC errors of the device</td>
</tr>
<tr>
<td>ONL_ERR</td>
<td>BYTE</td>
<td>Online error of the device</td>
</tr>
<tr>
<td>EXT_DIAG_LEN</td>
<td>BYTE</td>
<td>Valid length of the extended diagnostic data</td>
</tr>
<tr>
<td>EXT_DIAG_DAT</td>
<td>ARRAY[1..50] OF BYTE</td>
<td>Extended diagnostic data</td>
</tr>
</tbody>
</table>

Description

Using the block IBM_SLVDIAG, diagnostic data of individual devices on the bus can be requested.

Every time a FALSE \(\rightarrow\) TRUE edge is applied to the input EN, IBM_SLVDIAG reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE \(\rightarrow\) TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

EN

If a FALSE \(\rightarrow\) TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

CONO

The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

R019X CONO = 1
R01X9  CONO = 2
SLV BYTE
The physical position of the device in the Interbus network for which diagnostic data are to be requested is specified at input SLV. Valid values are 1 to 128.

Numbering of the devices starts with the device immediately connected to the master. This device gets the number 1. All further devices connected to the remote bus are numbered continuously. When using bus terminals the numbering is first continued in the lower-level segment. The following figure shows the counting mode when determining the physical position of a device in an Interbus network.

 ERR WORD
ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all Interbus blocks and is explained in section "ERR – Error messages of the Interbus blocks".

STAT_1 IBM_DEVICESTATUS_1_TYPE
STAT_1 outputs different bits as a structure of the type IBM_DEVICESTATUS_1_TYPE which display error states of the slave. STAT_1 is only valid if DONE = TRUE and ERR = 0.

The format of the structure of the type IBM_DEVICESTATUS_1_TYPE is defined in the Interbus library (see description below).

LEN_CODE BYTE
LEN_CODE outputs the actual length code of the device. This value is directly read from the concerning device. In case of configuration errors, the comparison of this actual length code with the corresponding set length code specified in the configuration data can be used to determine the error source. LEN_CODE is only valid if DONE = TRUE and ERR = 0.

IDENT_CODE BYTE
IDENT_CODE outputs the actual ID code of the device. This value is directly read from the concerning device. In case of configuration errors, the comparison of this actual ID code with the corresponding set ID code specified in the configuration data can be used to determine the error source. IDENT_CODE is only valid if DONE = TRUE and ERR = 0.

NUM_CRC_ERR WORD
NUM_CRC_ERR outputs the number of occurred checksum errors. An ascending counting value indicates electrical or electromagnetic interference in the environment of the concerning device. NUM_CRC_ERR is only valid if DONE = TRUE and ERR = 0.

DONE BOOL
The output DONE reflects the processing state. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE.

The values at the block outputs are only valid if DONE = TRUE. The diagnostic data are only valid if also ERR = 0. If ERR is not equal to 0, an error occurred.
ONL_ERR BYTE
ONL_ERR outputs a value which describes possible existing communication errors between master coupler and slave. ONL_ERR is only valid if DONE = TRUE and ERR = 0.

The error IDs of ONL_ERR correspond to the IDs of the IBM_STAT block output COM_ERR.EVENT. They are described in the table provided in the IBM_STAT block description.

EXT_DIAG_LEN BYTE
EXT_DIAG_LEN outputs the length or the number of valid entries of the extended diagnostic data in EXT_DIAG_DAT. If EXT_DIAG_LEN = 0, no extended diagnostic data for the corresponding device are available. Otherwise, EXT_DIAG_LEN contains the index of the last valid BYTE of the ARRAY EXT_DIAG_DAT. EXT_DIAG_LEN is only valid if DONE = TRUE and ERR = 0.

Function call in IL

| CAL  | DIAG
| (EN  | := DIAG_EN,
| CONO | := DIAG_CONO,
| SLV  | := DIAG_SLV) |

| LD   | DIAG.Done | ST  | DIAG_DONE
| ST   | DIAG.ERR  | LD  | DIAG.ERR
| ST   | DIAG.STAT_1 | LD  | DIAG.STAT_1
| ST   | DIAG.LEN_CODE | LD  | DIAG.LEN_CODE
| ST   | DIAG.IDENT_CODE | LD  | DIAG.IDENT_CODE
| ST   | DIAG_NUM_CRC | LD  | DIAG_NUM_CRC
| ST   | DIAG_ONL_ERR | LD  | DIAG_ONL_ERR
| ST   | DIAG_EXT_DIAG_LEN | LD  | DIAG_EXT_DIAG_LEN
| ST   | DIAG_EXT_DIAG_LEN | ST  | DIAG_EXT_DIAG_DAT
| ST   | DIAG_EXT_DIAG_DAT |

Note: In IL, the function call has to be performed in one line.

Function call in ST

| DIAG | (EN  | := DIAG_EN,
| CONO | := DIAG_CONO,
| SLV  | := DIAG_SLV); |

| DIAG_DONE   | := DIAG_DONE;
| DIAG.ERR    | := DIAG.ERR;
| DIAG_STAT_1 | := DIAG_STAT_1;
| DIAG.LEN_CODE | := DIAG.LEN_CODE;
| DIAG.IDENT_CODE | := DIAG.IDENT_CODE;
| DIAG_NUM_CRC | := DIAG_NUM_CRC;
| DIAG.ONL_ERR | := DIAG.ONL_ERR;
| DIAG_EXT_DIAG_LEN | := DIAG_EXT_DIAG_LEN;
| DIAG_EXT_DIAG_DAT | := DIAG_EXT_DIAG_DAT; |
The output STAT_1 of IBM_DEVDIAG displays different diagnostic bits as a structure of the type IBM_DEVICESTATUS_1_TYPE. Within the Interbus library the structure IBM_DEVICESTATUS_1_TYPE is declared as follows:

```
TYPE IBM_DEVICESTATUS_1_TYPE:
  STRUCT
    NO_RESPONSE: BOOL;
    BUFFER_OVFL: BOOL;
    PERIPHERAL: BOOL;
    CFG_FAULT: BOOL;
    RECONFIG: BOOL;
    INTERFACE_2: BOOL;
    INTERFACE_1: BOOL;
    DEACTIVATED: BOOL;
  END_STRUCT
END_TYPE
```

---

**NO_RESPONSE**

If this bit is set, the slave with the position specified for the block input SLV does not respond to the requests of the master or is not available. Normally NO_RESPONSE should be set to FALSE.

**BUFFER_OVFL**

If this bit is set, more diagnostic data concerning the relevant device are available than data can be buffered.

**PERIPHERAL**

If this bit is set, the slave has detected a power supply failure for the process inputs or a short circuit in at least one process input.

---

**CFG_FAULT**

This bit is set, if the actual length and/or ID code (actual configuration) of the device differs from the configured length and/or ID code (nominal configuration). In this case a configuration error occurred.

**RECONFIG**

If this bit is set, the slave reports a reconfiguration request. Bus terminals often provide a reconfiguration button which is accessible from the outside. Pressing this button on the site (e.g. because an additional slave module was installed), this is reported by RECONFIG = TRUE. As a result the master should be reconfigured in order to enable it to detect the planned system expansion.

**INTERFACE_2**

If this bit is set, the concerning device has switched off its outgoing bus interface 2 due to a failure.

**INTERFACE_1**

If this bit is set, the concerning device has switched off its outgoing bus interface 1 due to a failure. This interface is only available for devices with bus terminal property. It describes the interface to the lower-level segment.

**DEACTIVATED**

This bit is set to TRUE, if the device is defined in the configuration data of the master, but it is not activated. Therefore the concerning device is not processed. To include the device into processing, it has to be activated in the configuration data and the data has to be reloaded into the master.

---

The output EXT_DIAG_DAT consists of an ARRAY with a minimum size of 50 BYTE. This ARRAY provides, if available, additional diagnostic information concerning the relevant device.

The maximum number of extended diagnostic data is 50 BYTE. The actual amount of currently available data may change. As a result, the current valid length is output at EXT_DIAG_LEN. If for example EXT_DIAG_DAT is an ARRAY [1..50] OF BYTE and EXT_DIAG_LEN = 7, the entries with the indexes 1 to 7 contain the currently available diagnostic information. If EXT_DIAG_LEN = 0, no extended information are available.
Each BYTE of the ARRAY EXT_DIAG_DAT contains independent information. If more information are available at the same time, these are listed in sequence. Therefore, EXT_DIAG_DAT has to be evaluated byte wise. The following table shows the possible error descriptions for EXT_DIAG_DAT and provides appropriate actions for their remedy.

<table>
<thead>
<tr>
<th>EXT_DIAG_DAT[i]</th>
<th>Meaning</th>
<th>Error source</th>
<th>Action/Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>Device was not available during last network scan cycle</td>
<td>Device / Configuration</td>
<td>Check if configured device actually exists and Interbus is properly cabled</td>
</tr>
<tr>
<td>31</td>
<td>Device contains other ID code than configured</td>
<td>Device / Configuration</td>
<td>Check if configured ID code (EDS file) or ID code reported by the device is valid</td>
</tr>
<tr>
<td>32</td>
<td>Device contains other length code than configured</td>
<td>Device / Configuration</td>
<td>Check if configured length code (EDS file) or length code reported by the device is valid</td>
</tr>
<tr>
<td>33</td>
<td>Device has detected an unconfigured device at the outgoing bus interface 1</td>
<td>Device / Configuration</td>
<td>Determine the actual configuration of the device(s) and insert the device(s) into configuration</td>
</tr>
<tr>
<td>34</td>
<td>Device has detected an unconfigured device at the outgoing bus interface 2</td>
<td>Device / Configuration</td>
<td>Determine the actual configuration of the device(s) and insert the device(s) into configuration</td>
</tr>
<tr>
<td>35</td>
<td>Device was not available during last network scan cycle</td>
<td>Device / Configuration</td>
<td>Check the complete segment, where the device is located, for configuration errors</td>
</tr>
<tr>
<td>36</td>
<td>Device reports error at process input</td>
<td>Device</td>
<td>Check if supply voltage of external peripheral devices is properly connected or a short circuit in a process input exists</td>
</tr>
<tr>
<td>37</td>
<td>Device reports reconfiguration request</td>
<td>Device</td>
<td>Review configuration data and retransmit the data to the coupler</td>
</tr>
<tr>
<td>38</td>
<td>Device detected CRC error during data transmission</td>
<td>Device</td>
<td>Check the environment of the device for electrical or electromagnetic interference</td>
</tr>
<tr>
<td>39</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>Device has damaged outgoing bus interface 1 (bus terminal only)</td>
<td>Device</td>
<td>Check the interface for correct cabling</td>
</tr>
<tr>
<td>41</td>
<td>Device has damaged outgoing bus interface 2</td>
<td>Device</td>
<td>Check the interface for correct cabling</td>
</tr>
<tr>
<td>42</td>
<td>Device has not properly reported ID and length code during last network scan cycle</td>
<td>Network</td>
<td>Check the environment of the device for electrical or electromagnetic interference</td>
</tr>
<tr>
<td>43</td>
<td>Device not available during running operation due to disconnection of the Interbus connection</td>
<td>Network</td>
<td>Check the Interbus cabling between the concerning device and the devices located physically before</td>
</tr>
<tr>
<td>44</td>
<td>Connection to this device due to a disconnection in a local bus terminal lost</td>
<td>Local bus terminal</td>
<td>Check the Interbus cabling between the concerning device and the devices located physically before</td>
</tr>
<tr>
<td>45</td>
<td>During the last network scan cycle in running operation this was the last physical device to which a connection could be made</td>
<td>Network</td>
<td>Check the Interbus cabling between the concerning device and the device located physically behind</td>
</tr>
<tr>
<td>46</td>
<td>As the control system was temporarily in the STOP state, also the communication to this device was stopped temporarily</td>
<td>Operation</td>
<td>No error. This is only a note. This status message is generated only once after a RUN -&gt; STOP -&gt; RUN procedure of the control system</td>
</tr>
</tbody>
</table>
IBM_STAT outputs the status of an Interbus coupler. The outputs provide information about the communication state and error events.

Block type
- Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>IBM_STAT</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>STATE_BITS</td>
<td>IBM_STATE_BITS_TYPE</td>
<td>Atypical communication states</td>
</tr>
<tr>
<td>IBM_STATE</td>
<td>BYTE</td>
<td>General state of the Interbus master</td>
</tr>
<tr>
<td>COM_ERR</td>
<td>IBM_COM_ERR_TYPE</td>
<td>Communication error</td>
</tr>
<tr>
<td>DEF_DATA_CYC</td>
<td>WORD</td>
<td>Number of bad data cycles</td>
</tr>
<tr>
<td>NETWORK_SCANS</td>
<td>WORD</td>
<td>Number of required network scans</td>
</tr>
<tr>
<td>PD_CYC</td>
<td>DWORD</td>
<td>Number of performed process data cycles</td>
</tr>
<tr>
<td>DEF_PD_CYC</td>
<td>DWORD</td>
<td>Number of bad process data cycles</td>
</tr>
<tr>
<td>DIAG_CYC</td>
<td>DWORD</td>
<td>Number of performed diagnostic cycles</td>
</tr>
<tr>
<td>DEV_DIAGS</td>
<td>DWORD</td>
<td>Number of reported device diagnostics</td>
</tr>
</tbody>
</table>

Description
- The block IBM_STAT outputs the current Interbus coupler status.

IBM_STAT is active if input EN = TRUE. If the block is active, always the current values are displayed at the outputs.

EN
- BOOL
  - The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are available at the outputs.

CONO
- BYTE
  - The slot of the coupler (module number) whose state is to be read is specified at input CONO. The module number depends on the PLC category.

R019X CONO = 1
R01X9 CONO = 2

ERR
- INT
  - ERR outputs an error ID, if an invalid value was applied to a block input. ERR has always to be considered in conjunction with the input EN. The ERR output value is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The encoding of the error messages available at ERR applies to all Interbus blocks and is explained in section "ERR – Error messages of the Interbus blocks".

STATE_BITS
- IBM_STATE_BITS_TYPE
  - STATE_BITS outputs atypical communication states of the Interbus coupler. STATE_BITS is only valid if EN = TRUE and ERR = 0.
  - The format of the structure of the type IBM_STATE_BITS_TYPE is defined in the Interbus library (see description below).
**IBM_STATE**

IBM_STAT outputs the general communication state of the Interbus master. The following states are defined:

- **OFFLINE** 00 \text{HEX} / 00 \text{DEC}
- **STOP** 40 \text{HEX} / 64 \text{DEC}
- **CLEAR** 80 \text{HEX} / 128 \text{DEC}
- **OPERATE** C0 \text{HEX} / 192 \text{DEC}

**IBM_STATE** = OFFLINE

If **IBM_STATE** is set to OFFLINE, the Interbus coupler performs an initialization. After the initialization phase is completed the coupler changes to STOP state.

**IBM_STATE** = STOP

If **IBM_STATE** has the value STOP, the coupler is completely initialized. In this state the coupler is ready to receive configuration data. There is no data exchange with the slaves. The coupler is set to this state if no user program is running.

**IBM_STATE** = CLEAR

If the user program is started, the coupler changes from STOP into CLEAR and starts to establish the connections defined during configuration. After the setup has been successfully completed, the coupler changes into OPERATE state. If an error occurs during parameterization, the coupler changes back to STOP state.

**IBM_STATE** = OPERATE

Normally the coupler is in OPERATE state while an user program is running. In this state the master exchanges I/O data with the slaves. If an error occurs during this process, the coupler behaves as specified for the configured bus parameters, i.e. it remains in the OPERATE state or changes back to CLEAR state and retries, if necessary, to run the system. If the user program is stopped, the coupler also changes back to STOP state.

**IBM_STATE** is only valid if EN = TRUE and ERR = 0.

**DEF_DATA_CYC**

DEF_DATA_CYC outputs the total number of bus malfunctions occurred since system configuration or startup. Bad process data cycles and failures (diagnostic) reported by the slaves are counted. If the value of DEF_DATA_CYC increases, this indicates EMC interference in the environment of the Interbus network, an incorrect bus cabling or a bad power supply of the devices.

DEF_DATA_CYC is only valid if EN = TRUE and ERR = 0.

**NETWORK_SCANS**

NETWORK_SCANS outputs the total number of required re-initializations of the Interbus network since system configuration or startup. Such re-initializations result from bus errors. If a fatal bus error occurs, the master resets the Interbus network and searches the network for the fault location. This procedure is called ‘scan’. Depending on the configuration, the master either stops the complete Interbus communication or restarts the communication. If communication is restarted, only those devices are included which were detected during the last scan and which are error free.

NETWORK_SCANS is only valid if EN = TRUE and ERR = 0.

**PD_CYC**

PD_CYC outputs the total number of process data cycles performed since system configuration or startup.

PD_CYC is only valid if EN = TRUE and ERR = 0.

**DEF_PD_CYC**

DEF_PD_CYC outputs the total number of failed process data cycles detected since system configuration or startup.

DEF_PD_CYC is only valid if EN = TRUE and ERR = 0.

**DIAG_CYC**

DIAG_CYC outputs the total number of diagnostic cycles performed since system configuration or startup.

DIAG_CYC is only valid if EN = TRUE and ERR = 0.
DEV_DIAGS.

DEV_DIAGS outputs the total number of diagnostic events reported by all slaves.

DEV_DIAGS is only valid if EN = TRUE and ERR = 0.

**Note:**
The inputs and outputs can neither be duplicated nor inverted.

### Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>STAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>:= STAT_EN,</td>
</tr>
<tr>
<td>CONO</td>
<td>:= STAT_CONO</td>
</tr>
</tbody>
</table>

LD STAT.ERR
ST STAT.ERR
LD STAT.STATE_BITS
ST STAT.STATE_BITS
LD STAT.IBM_STATE
ST STAT.IBM_STATE
LD STAT.COM_ERR
ST STAT.COM_ERR
LD STAT.DEF_DATA_CYC
ST STAT.DEF_DATA_CYC
LD STAT.NETWORK_SCANS
ST STAT.NETWORK_SCANS
LD STAT.PD_CYC
ST STAT.PD_CYC
LD STAT.DEF_PD_CYC
ST STAT.DEF_PD_CYC
LD STAT.DIAG_CYC
ST STAT.DIAG_CYC
LD STAT.DEV_DIAGS
ST STAT.DEV_DIAGS

### Function call in ST

<table>
<thead>
<tr>
<th>STAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
</tr>
<tr>
<td>CONO</td>
</tr>
</tbody>
</table>

STAT.ERR := STAT.ERR;
STAT.STATE_BITS := STAT.STATE_BITS;
STAT.IBM_STATE := STAT.IBM_STATE;
STAT.COM_ERR := STAT.COM_ERR;
STAT.DEF_DATA_CYC := STAT.DEF_DATA_CYC;
STAT.NETWORK_SCANS := STAT.NETWORK_SCANS;
STAT.PD_CYC := STAT.PD_CYC;
STAT.DEF_PD_CYC := STAT.DEF_PD_CYC;
STAT.DIAG_CYC := STAT.DIAG_CYC;
STAT.DEV_DIAGS := STAT.DEV_DIAGS;

**Note:**
In IL, the function call has to be performed in one line.
STATE_BITS IBM_STATE_BITS_TYPE

The structure STATE_BITS consists of seven boolean variables which display different communication states. Within the Interbus library the data type IBM_STATE_BITS_TYPE is declared as follows:

TYPE IBM_STATE_BITS_TYPE:
  STRUCT
    CTRL: BOOL;
    AUTO_CLR: BOOL;
    NO_EXCH: BOOL;
    PERIPHERAL: BOOL;
    EVENT: BOOL;
    INTERFACE_1: BOOL;
    INTERFACE_2: BOOL;
  END_STRUCT
END_TYPE

CTRL BOOL
If this bit is set to TRUE, a parameterization error or a fatal runtime error occurred. In normal operation CTRL should be FALSE. If not, the parameter and configuration data have to be checked.

AUTO_CLR BOOL
If AUTO_CLR is set to TRUE, the coupler has stopped the data exchange with all slaves due to communication errors and has changed back to CLEAR state (see IBM_STATE). This will cause all outputs to change to the secure zero state. The behavior of the master is determined during configuration (bus parameters). Either the communication is reactivated by stopping and restarting the control system or a cold or warm start is required.

NO_EXCH BOOL
This bit is set to TRUE, if exchanging process data with at least one slave is not possible. The error cause can be found in the configuration data as well as in the slave itself. A more detailed diagnostic can be obtained by evaluating COM_ERR.

PERIPHERAL BOOL
If at least one device reports a peripheral error, the bit PERIPHERAL is set to TRUE. A peripheral error can be a short circuit in a local process output of a slave or a bad or missing power supply for the process inputs. The faulty device can be located using COM_ERR.ADDRESS. If the error occurs simultaneously in several devices, COM_ERR.ADDRESS contains the position of the device which is physically located next to the master.

EVENT BOOL
If EVENT is set to TRUE, at least one bad process data cycle occurred or the Interbus network was reset, scanned and reinitialized due to a modification of the network configuration (see DEF_DATA_CYC and NETWORK_SCANS). This bit can only be reset by a warm or cold start of the control system.

INTERFACE_1 BOOL
If INTERFACE_1 = TRUE, the outgoing bus interface 1 of a device (bus terminal) was switched off due to an error. The faulty device can be located using COM_ERR.ADDRESS. If the error occurs simultaneously in several devices, COM_ERR.ADDRESS contains the position of the device which is physically located next to the master.

INTERFACE_2 BOOL
If INTERFACE_2 = TRUE, the outgoing bus interface 2 of a device (bus terminal or slave) was switched off due to an error. The faulty device can be located using COM_ERR.ADDRESS. If the error occurs simultaneously in several devices, COM_ERR.ADDRESS contains the position of the device which is physically located next to the master.
**COM_ERR** **IBM_COM_ERR_TYPE**

Communication errors can be located more precisely using COM_ERR. The output COM_ERR is represented as a structure of the type IBM_COM_ERR_TYPE. Within the Interbus library this data type is declared as follows:

```
TYPE COM_ERR_TYPE:
    STRUCT
        ADDRESS: BYTE;
        EVENT: BYTE;
    END_STRUCT
END_TYPE
```

In case of an error, ADDRESS contains the bus address of the faulty device (0 to 128). If ADDRESS has the value 255, the coupler itself causes the error.

**ADDRESS** **BYTE**

**EVENT** **BYTE**

EVENT displays the error cause. The following tables show the encoding of the various errors.

### ADDRESS = 255  Coupler error

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>Unknown process data handshake mode</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>56</td>
<td>No configuration data</td>
<td>Configuration / Coupler</td>
<td>Load configuration data into coupler</td>
</tr>
<tr>
<td>57</td>
<td>Defective Interbus Controller</td>
<td>Coupler</td>
<td>Contact Support; coupler replacement</td>
</tr>
<tr>
<td>101</td>
<td>A configured length or ID code differs from the real network configuration</td>
<td>Configuration / Network</td>
<td>Check the ID and length codes of all devices</td>
</tr>
<tr>
<td>102</td>
<td>Too many devices connected</td>
<td>Network</td>
<td>Reduce the number of devices on the bus</td>
</tr>
<tr>
<td>103</td>
<td>Network configuration has changed during ID cycle</td>
<td>Network</td>
<td>Wait until the coupler automatically performs the next ID cycle</td>
</tr>
<tr>
<td>104</td>
<td>Network startup failed</td>
<td>Network</td>
<td>Contact Support</td>
</tr>
<tr>
<td>105</td>
<td>ID cycle interrupt due to non detectable network errors which are caused by the installation of devices or a faulty slave</td>
<td>Network</td>
<td>Wait until the coupler automatically performs the next ID cycle</td>
</tr>
<tr>
<td>106</td>
<td>Device, which was already detected during the previous ID cycle, is missing in the following cycle</td>
<td>Network</td>
<td>Wait until the coupler automatically performs the next ID cycle</td>
</tr>
<tr>
<td>107</td>
<td>Network configuration was changed in running operation, an already operating device does no longer respond</td>
<td>Network</td>
<td>Check the network; wait until the coupler automatically performs the next ID cycle</td>
</tr>
<tr>
<td>108</td>
<td>No connection to Interbus network</td>
<td>Network</td>
<td>Check the connection between master and first slave</td>
</tr>
<tr>
<td>220</td>
<td>Watchdog error</td>
<td>Control system</td>
<td>Contact Support</td>
</tr>
<tr>
<td>221</td>
<td>No process data handshake</td>
<td>Control system</td>
<td>Contact Support</td>
</tr>
<tr>
<td>224</td>
<td>Error communicating with Interbus Controller</td>
<td>Coupler</td>
<td>Contact Support</td>
</tr>
</tbody>
</table>
### ADDRESS = 1..128  Error in device with physical position ADDRESS

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>Device was not available during last network scan cycle</td>
<td>Slave / Configuration</td>
<td>Check the configuration data and Interbus cabling</td>
</tr>
</tbody>
</table>

A configured device could not be found in the Interbus network. But no fundamental error could be determined in the bus cabling. Therefore, this can be an error in the master configuration or the concerning module is not located at the configured position within the Interbus ring.

Compare the configured system description to the real configuration of the Interbus network.

If no differences can be found, check the connection between the faulty module and the module located before in the Interbus ring.

If the connection is also OK, monitor the module's LED 'RC' or 'CC'. If this LED does not light statically, a cabling error occurred. Please note that an Interbus cable must have a bridge between two pins of the male connector at the outgoing bus interface of the module. The interface is only activated, if this bridge is available. Verify that the bridge at the outgoing interface of the preceding module is properly wired.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Device reports another ID code than specified in the configuration data</td>
<td>Slave / Configuration</td>
<td>Compare the configured ID code to the real ID code of the device</td>
</tr>
</tbody>
</table>

The concerning device was found during the last network cycle but has reported another ID code than defined in the configuration data of the master. The ID code determines the device functionality and defines the device class and the I/O data supported by the device. If the retransmitted ID code differs from the configured ID, the master does not include the device into process data exchange.

Compare both ID codes and refer to the manufacturer's documentation or the label on the device for the valid code. If necessary, edit the EDS file of this device and reload the configuration into the master.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Device reports another length code than specified in the configuration data</td>
<td>Slave / Configuration</td>
<td>Compare the configured length code to the real length code of the device</td>
</tr>
</tbody>
</table>

The relevant device was found during the last network cycle but has reported another length code than contained in the configuration data of the master. The length code describes the number and length of I/O data of a device. If the retransmitted length code differs from the configured length, the master does not include the device into process data exchange.

Compare both length codes and refer to the manufacturer's documentation or the label on the device for the valid code. If necessary, edit the EDS file of this device and reload the configuration into the master.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>Unconfigured device found at outgoing bus interface 1</td>
<td>Slave / Configuration</td>
<td>Check the actual configuration of the unconfigured device; change configuration data</td>
</tr>
</tbody>
</table>

A communication error occurred. The following configured device is located in another bus segment level than the device configured at this position. This can have two reasons.

Either the bus segment level of the following device is configured improperly or the master has really detected at least one unconfigured device at the outgoing bus interface 1 of this device. Only devices with bus terminal properties are equipped with this outgoing bus interface 1.

Compare the configuration data to the real Interbus system configuration. Especially pay attention to missing devices and segment levels. Reedit the configuration and reload it into the master, if needed.
<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>Unconfigured device found at outgoing bus interface 2</td>
<td>Slave / Configuration</td>
<td>Check the actual configuration of the unconfigured device; change configuration data</td>
</tr>
</tbody>
</table>

A communication error occurred. The following configured device is located in another bus segment level than the device configured at this position. This can have two reasons. Either the bus segment level of the following device is configured improperly or the master has really detected at least one unconfigured device at the outgoing bus interface 2 of this device. Both (installation) remote bus and local bus devices contain this outgoing bus interface 2. If a local bus segment causes the error, the complete segment is switched off. If the error occurs in the remote bus, all devices located before the faulty device remain running and all following devices will be deactivated.

Compare the configuration data to the real Interbus system configuration. Especially pay attention to missing devices and segment levels. Rededit the configuration and reload it into the master, if needed.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>Peripheral error</td>
<td>Slave</td>
<td>Check the power supply of the module’s process inputs; check the process outputs for short circuit</td>
</tr>
</tbody>
</table>

If a slave reports a peripheral error, it has detected either an error in the power supply for the process inputs or a short circuit at the local inputs or outputs. The module can be operated in principle and it is still processed.

As this error message is generated vendor specific, no detailed statement can be made concerning the error source.

First, you should check the module’s LEDs. The LED ‘Us’ indicates if power supply is available. To be able to determine if an error occurred at the local I/Os, the input and output LEDs or a specific I/O error LED should be monitored depending on the module’s functionality.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Reconfiguration request</td>
<td>Slave</td>
<td>Master reset; update configuration data accordingly</td>
</tr>
</tbody>
</table>

Devices with bus terminal functionality usually have a reconfiguration button accessible from the outside. If this button is pressed, the relevant module sends a corresponding message to the master. This message causes the master to reset the Interbus system and to reconfigure the system. As this procedure cannot be performed automatically, a warm or cold start has to be performed manually. The configuration data have to be edited and reloaded, if needed.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Outgoing bus interface 1 (bus terminal) failed</td>
<td>Device</td>
<td>Check the cabling of the relevant bus interface</td>
</tr>
</tbody>
</table>

The outgoing (installation) remote bus or local bus interface of the bus terminal has caused a timeout error during a network scan cycle used to search for devices which are actually available in the segment.

Typically, this error occurs when connecting an Interbus cable to the interface to which no further devices are connected. Check the connection between the interface and the following module.

If no error can be found, check the LEDs of the connected modules.

In case of a local bus terminal, the LEDs ‘RC’ or ‘CC’ of all modules should light continuously. These LEDs indicate a generally working connection to the master. If the LED of one module does not light continuously, the Interbus connection between the relevant device and the preceding device has to be checked.

If all LEDs are on, the connections in transmitting direction (from the master’s point of view) are error-free. In this case, probably one of the modules itself is faulty and retransmits no data or the reception interface (towards the master) of one module is electrically defective.

In case of an (installation) remote bus terminal, the ‘RC’ or ‘CC’ LED of the directly following module of this terminal should light continuously. If not, the Interbus connection between the two devices is interrupted. The error can be caused by both the outgoing bus interface of the terminal and the incoming interface of the directly following module. First, the Interbus connection between these two devices has to be checked.

If the LED lights, the error can be caused by the module’s return connection to the master or by the incoming interface of the terminal. In this case, the devices should be replaced individually to locate the error.
<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>Outgoing bus interface 2 (bus terminal or slave) failed</td>
<td>Device</td>
<td>Check the cabling of the relevant bus interface</td>
</tr>
<tr>
<td></td>
<td>The outgoing interface of the device has caused a timeout error during</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a network scan cycle used to search for actually available devices.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Typically, this error occurs when connecting an Interbus cable to the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>interface to which no further devices are connected. Check the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>connection between the interface and the following module. If no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>error can be detected, check the „RC“ or „CC“ LED of the directly</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>following module. If the LED does not light continuously, the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interbus transmitting line between the two devices is interrupted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Check the Interbus connection between the two devices. If the LED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lights, either the following module is faulty and does not transmit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data or the incoming interface of the device itself is electrically</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>defective.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Device did not properly respond to the request for an ID code or</td>
<td>Network</td>
<td>Check the environment of the device for possible EMC interference sources</td>
</tr>
<tr>
<td></td>
<td>length code during the last network scan</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If errors are reported for process data cycles in running operation,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a network scan cycle is automatically performed. The master then</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>compares the ID and length codes retransmitted by the devices with</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>the corresponding codes defined in the configuration data. If the IDs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>differ, this error is generated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Typically, this error can only occur, if one device is first switched</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>off and then switched on again and the ID code „µP not ready“ = 0x0038</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>is reported during this time. As a result, this error event indicates</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>that a slave was switched off in the mean time. Check the bus cable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>shielding and the slave grounding and ensure that no temporary</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>outages or overvoltages occur for the power supply.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>Device is no longer operated by the coupler</td>
<td>Coupler</td>
<td>Coupler has stopped communicating with this device</td>
</tr>
</tbody>
</table>
## Configuration errors (70 – 92)

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>Duplicate address configured</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>71</td>
<td>Incorrect length of device data set</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>72</td>
<td>Incorrect length of process data configuration</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>73</td>
<td>Incorrect length of extended table</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>74</td>
<td>Incorrect length of PCP data</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>75</td>
<td>Inconsistency of the complete data set</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>76</td>
<td>Inconsistency of extended table</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>77</td>
<td>Maximum allowed offset for the process output data exceeded</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>78</td>
<td>Maximum allowed offset for the process input data exceeded</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>79</td>
<td>Maximum allowed offset address &gt; 255 exceeded</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>80</td>
<td>Number of modules and offsets inconsistent</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>81</td>
<td>Number of output modules and offsets differs</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>82</td>
<td>Number of input modules and offsets differs</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>83</td>
<td>Actual output data length differs from configured length</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>84</td>
<td>Actual input data length differs from configured length</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>85</td>
<td>Output data overlapping</td>
<td>Control system</td>
<td>Contact Support</td>
</tr>
<tr>
<td>86</td>
<td>Input data overlapping</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>87</td>
<td>Output device has also inputs</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>88</td>
<td>Input device has also outputs</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>89</td>
<td>Output device has inputs</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>90</td>
<td>Input device has outputs</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>91</td>
<td>Device with incorrect segment level configured</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>92</td>
<td>Configured ID code not supported by the coupler</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
</tbody>
</table>
READING A STATUS SURVEY OF ALL SLAVES

The block IBM_SYSDIAG outputs a bit field as status survey about all slaves at output SLV.
Three different surveys can be selected via input TYP.

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>IBM_SYSDIAG</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>TYP</td>
<td>BYTE</td>
<td>Selection of the survey type</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>SLV</td>
<td>ARRAY [0..128] OF BOOL</td>
<td>Status survey of the slaves</td>
</tr>
</tbody>
</table>

Description
The block IBM_SYSDIAG outputs different status surveys about all slaves. Three survey types can be selected:

configuration survey
I/O data exchange survey
diagnosis survey

EN BOOL
The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are available at the outputs.

CONO BYTE
The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.
R019X CONO = 1
R01X9 CONO = 2

TYP BYTE
The input TYP is used to select which status survey is required at output SLV.

TYP = 1 configuration survey
The output SLV displays which slaves have been successfully configured in the master.

TYP = 2 data exchange survey
Output SLV displays with which slaves the master exchanges I/O data. The data exchange can only be performed with slaves which were brought into service by the master. The data exchange survey can only be requested, if the coupler is in OPERATE state.

TYP = 3 diagnosis survey
The output SLV indicates which slaves report a diagnostics. The diagnosis survey can only be requested, if the coupler is in OPERATE state.

ERR WORD
ERR outputs an error ID, if an invalid value is applied to an input. ERR has always to be considered in conjunction with the input EN. The ERR output value is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The encoding of the error messages available at ERR apply to all Interbus blocks and is explained following the block descriptions.

SLV ARRAY [0..128] OF BOOL
SLV outputs the status survey as a bit field. Every bit within this field represents a slave. The index corresponds to the physical position of the slave in the Interbus network. A bit, which is set to TRUE, means that the state selected using TYP applies to the corresponding slave.

If e.g. TYP = 1 and SLV[2] = TRUE, the slave with bus address 2 (from the master's point of view) was configured successfully by the master. If SLV[2] = FALSE, the configuration of the specific slave has not yet been completed or the slave is not part of the master's configuration data.

If TYP = 2 and SLV[2] = TRUE, the master exchanges I/O data with the slave located at position 2. However, if SLV[2] = FALSE, the master currently does not exchange I/O data with the slave.
The master can only exchange data with such slaves which were previously brought into operation by the master.

If TYP = 3 and e.g. SLV[2] = TRUE, the slave at position 2 reports a diagnosis. The detailed diagnosis can then be requested using the block IBM_SLVDIAG.

The bit field obtained at SLV is only valid if EN = TRUE and ERR = 0.

Note:
The inputs and outputs can neither be duplicated nor inverted.

**Function call in IL**

```il
CAL   SYSDIAG
(EN   := SYSDIAG_EN,
CONO  := SYSDIAG_CONO,
TYP   := SYSDIAG_TYP)
LD    SYSDIAG.ERR
ST    SYSDIAG.ERR
LD    SYSDIAG.SLV
ST    SYSDIAG.SLV
```

Note:
In IL, the function call has to be performed in one line.

**Function call in ST**

```st
SYSDIAG
(EN   := SYSDIAG_EN,
CONO  := SYSDIAG_CONO,
TYP   := SYSDIAG_TYP);

SYSDIAG.ERR   := SYSDIAG.ERR;
SYSDIAG.SLV   := SYSDIAG.SLV;
```
READING AN OBJECT DESCRIPTION FROM A
SLAVE SUPPORTING PCP

The block IBM_GET_OD can be used
to get a description of an object from
a slave supporting PCP. The object of
interest can be accessed either using
the object index or the symbolic
name. IBM_GET_OD does not
provide the value of an object.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>IBM_GET_OD</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>COM_REF</td>
<td>BYTE</td>
<td>Communication reference</td>
</tr>
<tr>
<td>BY_NAME</td>
<td>BOOL</td>
<td>Request selection via object name or index name</td>
</tr>
<tr>
<td>OBJ_IDX</td>
<td>WORD</td>
<td>Object index</td>
</tr>
<tr>
<td>OBJ_NAME</td>
<td>STRING(22)</td>
<td>Object name</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>OBJ_DESC</td>
<td>IBM_OBJ_DESC_TYPE</td>
<td>Object information</td>
</tr>
</tbody>
</table>

**Description**

The block IBM_GET_OD can be used to get
information about a PCP object from a slave. The
object of interest can be accessed either using
the object index or the symbolic name. IBM_GET_OD does
not provide the value of an object (see block IBM_READ). The corresponding Communication
Reference (CR) as well as an Object Directory (OD) of
the slave must have been previously defined when
configuring (see 907 FB 1131 documentation).

Every time a FALSE → TRUE edge is applied to the
input EN, IBM_GET_OD reads the data at its inputs
and sends a corresponding request message to the
coupler. Further FALSE → TRUE edges at input EN
are ignored until processing of the active request is
finished. The completion of the request processing is
indicated by DONE = TRUE.

For a FALSE → TRUE edge at input EN, all further
inputs are applied.

If the input values are valid, a corresponding request
message is sent to the coupler. While this request
message is processed, the output DONE is set to
FALSE. If at least one input value is invalid, an error is
indicated at output ERR. Additionally the termination of
the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input
EN are recognized but not evaluated.

The slot of the coupler (module number) which should
process the request is specified at input CONO. The
module number depends on the PLC category.

R019X  CONO = 1
R01X9  CONO = 2
READING AN OBJECT DESCRIPTION FROM A
SLAVE SUPPORTING PCP

COM_REF BYTE
The Communication Reference (CR) number is specified at input COM_REF. Valid values are 2 to 63. Using the Communication Reference, the device is indirectly selected which data are to be read. For the corresponding number refer to the Communication Relationship List (CRL) in the 907 FB 1131 configuration data.

BY_NAME BYTE
The input BY_NAME is used to select how the object is accessed whose description is required.

OBJ_IDX WORD
The index of the object, whose description is to be requested, is specified at input OBJ_IDX. Generally, all values (0 – 65535) are valid for OBJ_IDX, but the access to unconfigured objects causes an error. For the configured objects and the corresponding indices (Obj.Idx.) refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

OBJ_NAME STRING(22)
The symbolic name of the object, whose description is to be requested, is specified at input OBJ_NAME. Generally, every STRING with up to 10 characters is allowed for OBJ_NAME (last character is string end character), but the access to unconfigured objects causes an error. For the configured objects and the corresponding names (description) refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

DONE BOOL
The output DONE reflects the processing state. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE for one cycle period.

The values at the block outputs are only valid if DONE = TRUE. The object description OBJ_DESC is only valid if also ERR = 0. If ERR is not equal to 0, an error occurred.

ERR WORD
ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to Interbus blocks and is explained after the block descriptions.
OBJ_DESC IBM_OBJ_DESC_TYPE
OBJ_DESC outputs the description of the requested object. OBJ_DESC is only valid if DONE = TRUE and ERR = 0.

The format of the structure of the type IBM_OBJ_DESC_TYPE is defined in the Interbus library as follows.

TYPE IBM_OBJ_DESC_TYPE:
  STRUCT
    OBJ_INDEX: WORD;
    OBJ_CODE: BYTE;
    NUM_OBJ_ELEM: BYTE;
    INDEX_OF_TYPE: WORD;
    OBJ_LENGTH: BYTE;
  END_STRUCT
  END_TYPE

OBJ_INDEX WORD
OBJ_INDEX provides the index of the requested object. In case of an object description request using the object index (BY_NAME = FALSE), this value provides no further information. In this case, it contains the value of the input OBJ_IDX after an error-free execution. If the object description request is done using the symbolic name (BY_NAME = TRUE), OBJ_INDEX outputs the index of the object with the symbolic name OBJ_NAME.

OBJ_CODE BYTE
OBJ_CODE provides the type of the requested object. There are 3 object types:

<table>
<thead>
<tr>
<th>OBJ_CODE</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Single variable</td>
</tr>
<tr>
<td>8</td>
<td>ARRAY</td>
</tr>
<tr>
<td>9</td>
<td>STRUCT</td>
</tr>
</tbody>
</table>

If OBJ_CODE = 7, the object is a single variable of the data type specified for INDEX_OF_TYPE.
OBJ_CODE = 8 specifies an object organized as ARRAY with NUM_OBJ_ELEM elements of the data type specified for INDEX_OF_TYPE. OBJ_CODE = 9 describes an object which represents a structure (STRUCT) combined using several variables of different data types. Different standardized structure types are defined in the Interbus profiles (see INDEX_OF_TYPE).

NUM_OBJ_ELEM BYTE
NUM_OBJ_ELEM contains the number of elements of an object of the type ARRAY (OBJ_CODE = 8).

INDEX_OF_TYPE BYTE
INDEX_OF_TYPE provides the type of data contained in the requested object. The following list shows the IDs of the defined standard data types. Additionally, the data types 'Process data description' and 'Ramp', described in the Interbus profile 22 (DRIVECOM), are listed. For the structure of these data types as well as the IDs and structures of the additional types refer to the corresponding profiles or the manufacturer descriptions for the particular device.

<table>
<thead>
<tr>
<th>INDEX_OF_TYPE</th>
<th>Data type</th>
<th>Length in Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOL</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>SINT</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>INT</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>DINT</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>USINT, BYTE</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>UINT, WORD</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>UDINT, DWORD</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>REAL</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>STRING</td>
<td>n</td>
</tr>
<tr>
<td>10</td>
<td>ARRAY [] OF BYTE</td>
<td>n</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>Date</td>
</tr>
<tr>
<td>12</td>
<td>TIME_OF_DAY</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>(TIME)</td>
<td>Time-Difference</td>
</tr>
<tr>
<td>14</td>
<td>(ARRAY [] OF BYTE)</td>
<td>Bit string</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>Process data description</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>Ramp</td>
</tr>
</tbody>
</table>

If the relevant object is a single variable (OBJ_CODE = 7), the complete object consists of one single variable of a listed data type. If an object consists of an ARRAY of variables (OBJ_CODE = 8), INDEX_OF_TYPE specifies the data type of each individual element of the ARRAY.

Here it has to be observed that for combined data types also ARRAYS of structures (e.g. INDEX_OF_TYPE = 32) or two-dimensional ARRAYS (e.g. INDEX_OF_TYPE = 10) can exist.

Additionally, it has to be considered that for some Interbus data types no corresponding data types (really) exist in 907 AC 1131.
READ AN OBJECT DESCRIPTION FROM A SLAVE SUPPORTING PCP IBM_GET_OD

OBJ_LENGTH BYTE
If the relevant object is a single variable (OBJ_CODE = 7) or a structure (OBJ_CODE = 9), OBJ_LENGTH provides the length or structure of the variables and therefore the length of the complete object as a BYTE value. If the object is organized as ARRAY (OBJ_CODE = 8), OBJ_LENGTH specifies the length of each individual element of the ARRAY as a BYTE value.

Note: The inputs and outputs can neither be duplicated nor inverted.

Function call in IL

<table>
<thead>
<tr>
<th>CAL</th>
<th>GET_OD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>:= GET_OD_EN,</td>
</tr>
<tr>
<td>CONO</td>
<td>:= GET_OD_CONO,</td>
</tr>
<tr>
<td>COM_REF</td>
<td>:= GET_OD_MAC_ID,</td>
</tr>
<tr>
<td>BY_NAME</td>
<td>:= GET_OD_CLASS_ID,</td>
</tr>
<tr>
<td>OBJ_IDX</td>
<td>:= GET_OD_INST_ID,</td>
</tr>
<tr>
<td>OBJ_NAME</td>
<td>:= GET_OD_ATTR_ID</td>
</tr>
</tbody>
</table>

LD GET_OD_DONE
ST GET_OD_DONE
LD GET_OD_ERR
ST GET_OD_ERR
LD GET_OD_OBJ_DESC
ST GET_OD_OBJ_DESC

Note: In IL, the function call has to be performed in one line.

Function call in ST

<table>
<thead>
<tr>
<th>GET_OD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN : = GET_OD_EN,</td>
</tr>
<tr>
<td>CONO : = GET_OD_CONO,</td>
</tr>
<tr>
<td>COM_REF : = GET_OD_COM_REF,</td>
</tr>
<tr>
<td>BY_NAME : = GET_OD_BY_NAME,</td>
</tr>
<tr>
<td>OBJ_IDX : = GET_OD_OBJ_IDX,</td>
</tr>
<tr>
<td>OBJ_NAME : = GET_OD_OBJ_NAME</td>
</tr>
</tbody>
</table>

GET_OD_DONE := GET_OD_DONE;
GET_OD_ERR := GET_OD_ERR;
GET_OD_OBJ_DESC := GET_OD_OBJ_DESC;
READING AN OBJECT FROM A SLAVE SUPPORTING PCP

The block IBM_READ can be used to read data from a slave supporting PCP via the parameter data channel.

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>COM_REF</td>
<td>BYTE</td>
<td>Communication reference</td>
</tr>
<tr>
<td>OBJ_IDX</td>
<td>WORD</td>
<td>Object index</td>
</tr>
<tr>
<td>OBJ_SUB_IDX</td>
<td>BYTE</td>
<td>Object subindex</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address of the variable (via ADR operator) from which on the read object data are stored</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of the object data to be read</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

Description

The block IBM_READ can be used to read an object from a slave via the PCP channel. The corresponding Communication Reference (CR) as well as an Object Directory (OD) of the slave must have been previously defined when configuring (see 907 FB 1131 documentation).

Every time a FALSE → TRUE edge is applied to the input EN, IBM_READ reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

EN BOOL
For a FALSE → TRUE edge at input EN, all further inputs are applied.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

CONO BYTE
The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

R019X CONO = 1
R01X9 CONO = 2
COM_REF BYTE
The Communication Reference (CR) number is specified at input COM_REF. Valid values are 2 to 63. Using the Communication Reference, the device is indirectly selected which data are to be read. For the corresponding number refer to the Communication Relationship List (CRL) in the 907 FB 1131 configuration data.

OBJ_IDX WORD
The index of the object to be read from the slave is specified at input OBJ_IDX. Generally, all values (0 – 65535) are valid for OBJ_IDX, but the access to unconfigured objects causes an error. For the configured objects and the corresponding indices (Obj.Idx.) refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

OBJ_SUB_IDX BYTE
The subindex of the object to be read from the slave is specified at input OBJ_SUB_IDX. Generally, all values (0 – 255) are valid for OBJ_SUB_IDX, but the access to unconfigured objects causes an error. For the access to a single variable only OBJ_SUB_IDX = 0 is allowed. If an object of the type ARRAY is accessed using OBJ_SUB_IDX = 0, the complete ARRAY is read. The access to a single element of the ARRAY is done with OBJ_SUB_IDX unequal to 0. Here it has to be observed that the first element of an ARRAY always has the subindex 1 (ARRAY [1..X]). For the formats of the configured objects and the allowed subindices refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

DATA DWORD
The address of the variable, to which the read object data are to be written, is specified at the input DATA via the ADR operator. It must be absolutely observed that enough memory area is available after this address in order to store the expected data volume. To be able to interpret the received data, also the format of the variables has to correspond to the format of the object specified using OBJ_IDX and OBJ_SUB_IDX.

If the object to be read is for example an INT-ARRAY with 32 elements (OBJ_SUB_IDX = 0), a variable of the type ARRAY [1..32] OF INT has to be generated and its start address has to be applied to the input DATA using the ADR operator. Using the block IBM_READ objects with a length of up to 240 bytes can be read. If the object size exceeds this maximum size, reading must be performed sequentially during several successive procedures.

For the formats of the configured objects refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data. The received object data stored beginning at address DATA are only valid if DONE = TRUE and ERR = 0.

DATA_LEN BYTE
At input DATA_LEN, the length of the object data to be read and stored beginning at address DATA is specified as a byte value. DATA_LEN must correspond to the object specified using COM_REF, OBJ_IDX and OBJ_SUB_IDX. If the data length received by the slave differs from DATA_LEN, an error is reported and the data are not transmitted to the control system. For each procedure a maximum of 240 bytes of data can be written. If the object size exceeds this maximum size, writing must be performed sequentially during several successive procedures.

DONE BOOL
The output DONE reflects the processing state. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE for one cycle period. The values at the block outputs are only valid if DONE = TRUE. The object data are only valid if also ERR = 0. If ERR is not equal to 0, an error occurred.
ERR word

ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all Interbus blocks and is explained in section "ERR – Error messages of the Interbus blocks".

Note:
The inputs and outputs can neither be duplicated nor inverted.

Function call in IL

CAL READ_DATA
(EN := READ_EN,
CONO := READ_CONO,
COM_REF := READ_COM_REF,
OBJ_IDX := READ_OBJ_IDX,
OBJ_SUBIDX := READ_OBJ_SUBIDX,
DATA := READ_DATA,
DATA_LEN := READ_DATA_LEN),

LD READ_DONE
ST READ_DONE
LD READ_ERR
ST READ_ERR

Note:
In IL, the function call has to be performed in one line.

Function call in ST

READ
(EN := READ_EN,
CONO := READ_CONO,
COM_REF := READ_COM_REF,
OBJ_IDX := READ_OBJ_IDX,
OBJ_SUBIDX := READ_OBJ_SUBIDX,
DATA := ADR(READ_DATA),
DATA_LEN := READ_DATA_LEN),

READ_DONE := READ_DONE;
READ_ERR := READ_ERR;
ENABLING THE READING OF AN OBJECT FROM THE CONTROL SYSTEM VIA A SLAVE SUPPORTING PCP

The block IBM_READ_EN allows a slave to independently read data from the control system via the parameter data channel.

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>Instance name</th>
<th>EN</th>
<th>BOOL</th>
<th>Enabling the block processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONO</td>
<td>SLOT (module number) of the coupler</td>
<td>COM_REF</td>
<td>BYTE</td>
<td>Communication reference</td>
</tr>
<tr>
<td>OBJ_IDX</td>
<td>OBJECT index</td>
<td>OBJ_SUB_IDX</td>
<td>BYTE</td>
<td>Object subindex</td>
</tr>
<tr>
<td>DATA</td>
<td>ADDRESS from which on the data to be written are stored (via ADR operator)</td>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of data enabled for writing</td>
</tr>
<tr>
<td>DONE</td>
<td>READY message of the block</td>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

Description

The block IBM_READ_EN enables a slave to independently read an object from the control system via the PCP channel. The corresponding Communication Reference (CR) as well as an Object Directory (OD) of the slave must have been previously defined when configuring.

The block IBM_READ_EN is activated by a FALSE → TRUE edge at input EN. The values applied to the block inputs at this time are taken. The block is active while EN = TRUE. In this state the block automatically processes all read requests of a slave whose parameters COM_REF, OBJ_IDX and OBJ_SUB_IDX match to the parameters which were applied to the block inputs on activation (FALSE → TRUE edge at input EN). After each processing of a slave request, the block sets the output DONE to TRUE for one cycle and displays an error at the output ERR, if occurred.

To change the parameters COM_REF, OBJ_IDX and OBJ_SUB_IDX, the block has to be deactivated first (EN = FALSE) and after the modification it has to be reactivated using the block inputs.

EN | BOOL
For a FALSE → TRUE edge at input EN, all further inputs are applied.

If valid values are applied to the inputs, it is cyclically checked whether a read request of a slave according to the inputs COM_REF, OBJ_IDX and OBJ_SUB_IDX exists. If so, the coupler automatically sends the data with the length DATA_LEN available at the input DATA to the slave and signalizes this by setting the output DONE to TRUE for one cycle period. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

The block can be deactivated with EN = FALSE.
ENABLING THE READING OF AN OBJECT FROM THE
CONTROL SYSTEM VIA A SLAVE SUPPORTING PCP

CONO BYTE
The slot of the coupler (module number) which should
process the request is specified at input CONO. The
module number depends on the PLC category.
R019X CONO = 1
R01X9 CONO = 2

COM_REF BYTE
The Communication Reference (CR) number is
specified at input COM_REF. Valid values are 2 to 63.
Using the Communication Reference, the device is
indirectly selected which may read the data. For the
(corresponding number refer to the Communication
Relationship List (CRL) in the 907 FB 1131
configuration data.

OBJ_IDX WORD
The index of the object, which should be enabled for
read requests by the slave, is specified at input
OBJ_IDX. Generally, all values (0 – 65535) are valid for
OBJ_IDX, but the access to unconfigured objects
causes an error. For the configured objects and the
(corresponding indices refer to the Object Directory
(OD) of the specific device in the 907 FB 1131
configuration data.

OBJ_SUB_IDX BYTE
The subindex of the object, which should be enabled
for read requests by the slave, is specified at input
OBJ_SUB_IDX. Generally, all values (0 – 255) are
valid for OBJ_SUB_IDX, but the access to
unconfigured objects causes an error. For the access
to a single variable only OBJ_SUB_IDX = 0 is allowed.
If an object of the type ARRAY is accessed using
OBJ_SUB_IDX = 0, the complete ARRAY is read. The
access to a single element of the ARRAY is done with
OBJ_SUB_IDX unequal to 0. Here it has to be
observed that the first element of an ARRAY always
has the subindex 1 (ARRAY [1..X]).
For the formats of the configured objects and the
allowed subindices refer to the Object Directory (OD) of
the specific device in the 907 FB 1131 configuration
data.

DATA DWORD
The address of the variable, which contains the object
data to be read, is specified at the input DATA via the
ADR operator.
To be able to interpret the stored data, the format of the
variables has to correspond to the format of the object
specified using OBJ_IDX and OBJ_SUB_IDX.

If the object to be read is for example an INT-ARRAY
with 32 elements, a variable of the type
ARRAY [1..32] OF INT has to be generated and its
start address has to be applied to the input DATA using
the ADR operator (OBJ_SUB_IDX = 0).

Using the block IBM_READ_EN, objects with a length
of up to 238 bytes can be read. If the object size
exceeds this maximum size, reading must be
performed sequentially during several successive
procedures.

For the formats of the configured objects refer to the
Object Directory (OD) of the specific device in the
907 FB 1131 configuration data.

DATA_LEN BYTE
At input DATA_LEN, the length of the object data
stored beginning at address DATA is specified as a
byte value. These data should be transmitted during a
read access via the appropriate communication
relationship. DATA_LEN must correspond to the object
specified using COM_REF, OBJ_IDX and
OBJ_SUB_IDX. If the length of the data read by the
slave differs from DATA_LEN, an error is reported and
the data will not be transmitted to the slave. For each
procedure a maximum of 238 bytes of data can be
read.

DONE BOOL
The output DONE reflects the processing state. After
each processing of a task, DONE is set to TRUE for
one cycle period.
The values at the block outputs are only valid if
DONE = TRUE. If ERR is not equal to 0 at this time, an
error occurred.

ERR WORD
ERR outputs an error ID if an invalid value was applied
to an input or if an error occurred during request
processing. ERR has always to be considered in
conjunction with output DONE. The ERR output value
is only valid if DONE = TRUE. The encoding of the
error messages available at ERR applies to all Interbus
blocks and is explained in section "ERR – Error
messages of the Interbus blocks".

Note:
The inputs and outputs can neither be duplicated nor
inverted.
### Function call in IL

<table>
<thead>
<tr>
<th>Operator</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>READ_EN_DATA</td>
</tr>
<tr>
<td>ADR</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>ADR_RAED_EN_DATA</td>
</tr>
<tr>
<td>CAL</td>
<td>READ_EN</td>
</tr>
<tr>
<td>(EN := READ_EN_EN,</td>
<td></td>
</tr>
<tr>
<td>CONO := READ_EN_CONO,</td>
<td></td>
</tr>
<tr>
<td>COM_REF := READ_EN_COM_REF,</td>
<td></td>
</tr>
<tr>
<td>OBJ_IDX := READ_EN_OBJ_IDX,</td>
<td></td>
</tr>
<tr>
<td>OBJ_SUB_IDX := READ_EN_OBJ_SUB_IDX,</td>
<td></td>
</tr>
<tr>
<td>DATA := ADR_READ_EN_DATA,</td>
<td></td>
</tr>
<tr>
<td>DATA_LEN := READ_EN_DATA_LEN)</td>
<td></td>
</tr>
</tbody>
</table>

### Function call in ST

```plaintext
READ_EN
(EN := READ_EN_EN, |
CONO := READ_EN_CONO, |
COM_REF := READ_EN_COM_REF, |
OBJ_IDX := READ_EN_OBJ_IDX, |
OBJ_SUB_IDX := READ_EN_OBJ_SUB_IDX, |
DATA := ADR(READ_EN_DATA), |
DATA_LEN := READ_EN_DATA_LEN);
```

```plaintext
READ_ENDONE := READ_EN.DONE;
READ_EN_ERR := READ_EN.ERR;
```

Note:
- In IL, the function call has to be performed in one line.
WRITING AN OBJECT TO A SLAVE SUPPORTING PCP

The block IBM_WRITE can be used to write an object to a slave supporting PCP via the parameter data channel.

![Diagram of IBM_WRITE block]

**Block type**

Function block with historical values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL Enabling the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE Slot (module number) of the coupler</td>
</tr>
<tr>
<td>COM_REF</td>
<td>BYTE Communication reference</td>
</tr>
<tr>
<td>OBJ_IDX</td>
<td>WORD Object index</td>
</tr>
<tr>
<td>OBJ_SUB_IDX</td>
<td>BYTE Object subindex</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD Address (via ADR operator) from which on the object data to be written are stored</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE Length of the object data to be written</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD Error message</td>
</tr>
</tbody>
</table>

**Description**

The block IBM_WRITE can be used to write an object to a slave via the PCP channel. The corresponding Communication Reference (CR) as well as an Object Directory (OD) of the slave must have been previously defined when configuring.

Every time a FALSE → TRUE edge is applied to the input EN, IBM_WRITE reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active request is finished. The completion of the request processing is indicated by DONE = TRUE.

EN

The block is activated with a FALSE → TRUE edge at input EN. The values applied to the block inputs at this time are taken.

If valid values are applied to the inputs, it is cyclically checked whether a write request of a slave according to the inputs COM_REF, OBJ_IDX and OBJ_SUB_IDX exists. If so, the received data with the length DATA_LEN are automatically written to the variable address which was specified at the input DATA when the FALSE → TRUE edge at EN was applied. This is signalized by setting the output DONE to TRUE for one cycle period. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

The block can be deactivated with EN = FALSE.
CONO BYTE
The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.
R019X CONO = 1
R01X9 CONO = 2

COM_REF BYTE
The Communication Reference (CR) number is specified at input COM_REF. Valid values are 2 to 63. Using the Communication Reference, the device is indirectly selected to which the data are to be written. For the corresponding number refer to the Communication Relationship List (CRL) in the 907 FB 1131 configuration data.

OBJ_IDX WORD
The index of the object to be written in the slave is specified at input OBJ_IDX. Generally, all values (0 – 65535) are valid for OBJ_IDX, but the access to unconfigured objects causes an error. For the configured objects and the corresponding indices refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

OBJ_SUB_IDX BYTE
The subindex of the object to be written in the slave is specified at input OBJ_SUB_IDX. Generally, all values (0 – 255) are valid for OBJ_SUB_IDX, but the access to unconfigured objects causes an error. For the single variable only OBJ_SUB_IDX = 0 is allowed. If an object of the type ARRAY is accessed using OBJ_SUB_IDX = 0, the complete ARRAY is written. The access to a single element of the ARRAY is done with OBJ_SUB_IDX unequal to 0. Here it has to be observed that the first element of an ARRAY always has the subindex 1 (ARRAY [1..X]).
For the formats of the configured objects and the allowed subindices refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

DATA DWORD
The address of the variable, which contains the object data transmitted by the slave, is specified at the input DATA via the ADR operator.

To ensure the correct data format, the format of the variables has to correspond to the format of the object specified using OBJ_IDX and OBJ_SUB_IDX. If the object to be written is for example an INT-ARRAY with 32 elements, a variable of the type ARRAY [1..32] OF INT has to be generated and its start address has to be applied to the input DATA using the ADR operator (OBJ_SUB_IDX = 0).

If the object is for example an INT-ARRAY with 32 elements, a variable of the type ARRAY [1..32] OF INT has to be generated and its start address has to be applied to the input DATA using the ADR operator (OBJ_SUB_IDX = 0).

For the formats of the configured objects refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data. The length of the data to be written (specified as byte value) has also to be applied to input DATA_LEN.

The write process was successfully completed, if DONE = TRUE and ERR = 0. For DONE = TRUE and ERR unequal to 0, an error occurred and the data were not written. The error cause can be determined by evaluating the output ERR with the help of the table listed below.

The received object data stored beginning at address DATA are only valid if DONE = TRUE and ERR = 0.

DATA_LEN BYTE
At input DATA_LEN, the length of the object data to be written and stored beginning at address DATA is specified as a byte value. DATA_LEN must correspond to the object specified using COM_REF, OBJ_IDX and OBJ_SUB_IDX. If the object length in the slave differs from DATA_LEN, an error is reported and the data will not be written to the slave. For each procedure a maximum of 240 bytes of data can be written. If the object size exceeds this maximum size, writing must be performed sequentially during several successive procedures.

DONE BOOL
The output DONE reflects the processing state. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE. This indicates that the request is in progress. If processing of the task is completed, DONE is set to TRUE for one cycle period.
The values at the block outputs are only valid if DONE = TRUE. If ERR is not equal to 0 at this time, an error occurred.
ERR WORD

ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all Interbus blocks and is explained in section "ERR – Error messages of the Interbus blocks".

Note:
The inputs and outputs can neither be duplicated nor inverted.

Function call in IL

| LD     | WRITE_DATA |
| ADR    |            |
| ST     | ADR_WRITE_DATA |
| CAL    | WRITE_DATA |
| (EN)   | := WRITE_EN, |
| CONO   | := WRITE_CONO, |
| COM_REF| := WRITE_COM_REF, |
| OBJIDX | := WRITE_OBJ_IDX, |
| OBJ_SUB_IDX | := WRITE_OBJ_SUB_IDX, |
| DATA   | := ADR_WRITE_DATA; |
| DATA_LEN| := WRITE_DATA_LEN), |
| LD     | WRITE_DONE |
| ST     | WRITE_DONE |
| LD     | WRITE.ERR |
| ST     | WRITE.ERR |

Note:
In IL, the function call has to be performed in one line.

Function call in ST

WRITE

(EN := WRITE_EN,
CONO := WRITE_CONO,
COM_REF := WRITE_COM_REF,
OBJIDX := WRITE_OBJ_IDX,
OBJ_SUB_IDX := WRITE_OBJ_SUB_IDX,
DATA := ADR(WRITE_DATA),
DATA_LEN := WRITE_DATA_LEN);

WRITE_DONE := WRITE_DONE;
WRITE_ERR := WRITE.ERR;
ENABLING THE WRITING OF AN OBJECT BY A SLAVE SUPPORTING PCP

The block IBM_WRITE_EN allows a slave to independently write data to the control system via the parameter data channel.

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>IBM_WRITE_EN</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN Burk</td>
<td>BOOL</td>
<td>Enabling the block processing</td>
</tr>
<tr>
<td>CONO BYTE</td>
<td>Slot (module number) of the coupler</td>
<td></td>
</tr>
<tr>
<td>COM_REF BYTE</td>
<td>Communication reference</td>
<td></td>
</tr>
<tr>
<td>OBJ_IDX WORD</td>
<td>Object index</td>
<td></td>
</tr>
<tr>
<td>OBJ_SUB_IDX BYTE</td>
<td>Object subindex</td>
<td></td>
</tr>
<tr>
<td>DATA DWORD</td>
<td>Address from which on the data enabled for writing are stored (via ADR operator)</td>
<td></td>
</tr>
<tr>
<td>DATA_LEN BYTE</td>
<td>Length of data enabled for writing</td>
<td></td>
</tr>
<tr>
<td>DONE BOOL</td>
<td>Ready message of the block</td>
<td></td>
</tr>
<tr>
<td>ERR WORD</td>
<td>Error message</td>
<td></td>
</tr>
</tbody>
</table>

Description

The block IBM_WRITE_EN enables a slave to independently write an object to the control system via the PCP channel. The corresponding Communication Reference (CR) as well as an Object Directory (OD) of the slave must have been previously defined when configuring.

The block IBM_WRITE_EN is activated by a FALSE → TRUE edge at input EN. The values applied to the block inputs at this time are taken. The block IBM_WRITE_EN is active while EN = TRUE. In this state the block processes automatically all write requests of a slave whose parameters COM_REF, OBJ_IDX and OBJ_SUB_IDX match to the parameters which were applied to the block inputs on activation (FALSE → TRUE edge at input EN). After each processing of a slave request, the block sets the output DONE to TRUE for one cycle and displays an error at the output ERR, if occurred.

To change the parameters COM_REF, OBJ_IDX and OBJ_SUB_IDX, the block has to be deactivated first (EN = FALSE) and after the modification it has to be reactivated using the block inputs.

EN BOOL
The block is activated with a FALSE → TRUE edge at input EN. The values applied to the block inputs at this time are taken.

If valid values are applied to the inputs, it is cyclically checked whether a write request of a slave according to the inputs COM_REF, OBJ_IDX and OBJ_SUB_IDX exists. If so, the received data with the length DATA_LEN are automatically written to the variable address which was specified at the input DATA when the FALSE → TRUE edge at EN was applied. This is signalized by setting the output DONE to TRUE for one cycle period. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE.

The block can be deactivated with EN = FALSE.
**ENABLING THE WRITING OF AN OBJECT BY A SLAVE**

**SUPPORTING PCP**

**IBM_WRITE_EN**

---

**CONO**

Byte

The slot of the coupler (module number) which should process the request is specified at input CONO. The module number depends on the PLC category.

R019X  CONO = 1
R01X9  CONO = 2

---

**COM_REF**

Byte

The Communication Reference (CR) number is specified at input COM_REF. Valid values are 2 to 63. Using the Communication Reference, the device is indirectly selected which may read the data. For the corresponding number refer to the Communication Relationship List (CRL) in the 907 FB 1131 configuration data.

---

**OBJ_IDX**

Word

The index of the object, which the slave enables for writing, is specified at input OBJ_IDX. Generally, all values (0 – 65535) are valid for OBJ_IDX, but the access to unconfigured objects causes an error. For the configured objects and the corresponding indices refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

---

**OBJ_SUB_IDX**

Byte

The subindex of the object, whose data the slave may independently write to the control system, is specified at input OBJ_SUB_IDX. Generally, all values (0 – 255) are valid for OBJ_SUB_IDX, but the access to unconfigured objects causes an error. For the access to a single variable only OBJ_SUB_IDX = 0 is allowed. If an object of the type ARRAY is accessed using OBJ_SUB_IDX = 0, the complete ARRAY is written. The access to a single element of the ARRAY is done with OBJ_SUB_IDX unequal to 0. Here it has to be observed that the first element of an ARRAY always has the subindex 1 (ARRAY [1..X]).

For the formats of the configured objects and the allowed subindices refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

---

**DATA**

DWORD

The address of the variable, to which the object data transmitted by the slave are to be written, is specified at the input DATA via the ADR operator.

It must be absolutely observed that enough memory area is available after this address in order to store the expected data volume. To be able to interpret the received data, also the format of the variables has to correspond to the format of the object specified using OBJ_IDX and OBJ_SUB_IDX.

For example, if an INT-ARRAY with 32 elements, a variable of the type ARRAY [1..32] OF INT has to be generated and its start address has to be applied to the input DATA using the ADR operator (OBJ_SUB_IDX = 0).

Using the block IBM_WRITE_EN objects with a length of up to 238 bytes can be written to the control system. If the object size exceeds this maximum size, writing must be performed sequentially during several successive procedures.

For the formats of the configured objects refer to the Object Directory (OD) of the specific device in the 907 FB 1131 configuration data.

The received object data stored beginning at address DATA are only valid if DONE = TRUE and ERR = 0.

---

**DATA_LEN**

Byte

At input DATA_LEN, the length of the required memory area (beginning at address DATA) used to store the received object data is specified as a byte value. DATA_LEN must correspond to the object specified using COM_REF, OBJ_IDX and OBJ_SUB_IDX. If the length of the data transmitted by the slave differs from DATA_LEN, an error is reported and the data are not written to the control system. For each procedure a maximum of 238 bytes of data can be written.

---

**DONE**

Bool

The output DONE reflects the processing state. After each processing of a task, DONE is set to TRUE for one cycle period.

The values at the block outputs are only valid if DONE = TRUE. If ERR is not equal to 0 at this time, an error occurred.
ERR WORD
ERR outputs an error ID if an invalid value was applied to an input or if an error occurred during request processing. ERR has always to be considered in conjunction with output DONE. The ERR output value is only valid if DONE = TRUE. The encoding of the error messages available at ERR applies to all Interbus blocks and is explained in section "ERR – Error messages of the Interbus blocks".

Note:
The inputs and outputs can neither be duplicated nor inverted.

Function call in IL

\[
\begin{align*}
\text{LD} & \quad \text{WRITE}_\text{EN\_DATA} \\
\text{ADR} & \\
\text{ST} & \quad \text{ADR\_WRITE}_\text{EN\_DATA} \\
\text{CAL} & \quad \text{WRITE}_\text{EN} \\
\text{EN} & \quad := \text{WRITE}_\text{EN\_EN}, \\
\text{CONO} & \quad := \text{WRITE}_\text{EN\_CONO}, \\
\text{COM\_REF} & \quad := \text{WRITE}_\text{EN\_COM\_REF}, \\
\text{OBJ\_IDX} & \quad := \text{WRITE}_\text{EN\_OBJ\_IDX}, \\
\text{OBJ\_SUB\_IDX} & \quad := \text{WRITE}_\text{EN\_OBJ\_SUB\_IDX}, \\
\text{DATA} & \quad := \text{ADR\_WRITE}_\text{EN\_DATA}, \\
\text{DATA\_LEN} & \quad := \text{WRITE}_\text{EN\_DATA\_LEN}) \\
\text{LD} & \quad \text{WRITE}_\text{EN\_DONE} \\
\text{ST} & \quad \text{WRITE}_\text{EN\_DONE} \\
\text{LD} & \quad \text{WRITE}_\text{EN\_ERR} \\
\text{ST} & \quad \text{WRITE}_\text{EN\_ERR} \\
\end{align*}
\]

Note:
In IL, the function call has to be performed in one line.

Function call in ST

\[
\begin{align*}
\text{WRITE}_\text{EN}\left(\text{EN} := \text{WRITE}_\text{EN\_EN}, \right. \\
\left.\text{CONO} := \text{WRITE}_\text{EN\_CONO}, \right. \\
\left.\text{COM\_REF} := \text{WRITE}_\text{EN\_COM\_REF}, \right. \\
\left.\text{OBJ\_IDX} := \text{WRITE}_\text{EN\_OBJ\_IDX}, \right. \\
\left.\text{OBJ\_SUB\_IDX} := \text{WRITE}_\text{EN\_OBJ\_SUB\_IDX}, \right. \\
\left.\text{DATA} := \text{ADR\_WRITE}_\text{EN\_DATA}, \right. \\
\left.\text{DATA\_LEN} := \text{WRITE}_\text{EN\_DATA\_LEN}\right) \\
\text{WRITE}_\text{EN\_DONE} := \text{WRITE}_\text{EN\_DONE}; \\
\text{WRITE}_\text{EN\_ERR} := \text{WRITE}_\text{EN\_ERR}; \\
\end{align*}
\]
**Glossary**

**BOOL**

Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

**DINT**

DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**DWORD**

DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**INT**

INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**WORD**

WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost during the conversion of greater data types to smaller data types.

**STRING**
Variables of the type STRING can include any character string. The specified value for the memory space allocation in the declaration corresponds to characters and can be defined using parenthesis or brackets. If no value (1 to 255) is specified, 80 characters are applied by default.

Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures etc.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can be derived:

- Within functions, global variables can neither be read nor written.
- Within functions, it is not allowed to read or write absolute operands.
- Within functions, it is not allowed to call function blocks.

Function blocks

Function blocks are subroutines which can have as many inputs, outputs and internal variables as required. They are called by a program or by another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. This allows for example to realize counters which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two main points:

- A function block has multiple output parameters and a function has a maximum of one output parameter, where the output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
Components of the Interbus Library 3

D
DEVICESTATUS_1_TYPE 11

E
Error messages of the Interbus blocks 4

I
IBM_COM_ERR_TYPE 17
IBM_GET_OD S90 24
IBM_READ S90 28
IBM_READ_EN S90 31
IBM_SLVDIAG S90 8
IBM_STAT S90 13
IBM_STATE_BITS_TYPE 16
IBM_SYSDIAG S90 22
IBM_WRITE S90 34
IBM_WRITE_EN S90 37
Interbus S90
IBM_SLVDIAG 8
IBM_STAT 13
IBM_SYSDIAG 22
INTERBUS_INFO 6
INTERBUS S90
DEVICESTATUS_1_TYPE 11
IBM_COM_ERR_TYPE 17
IBM_GET_OD 24
IBM_READ 28
IBM_READ_EN 31
IBM_STATE_BITS_TYPE 16
IBM_WRITE 34
IBM_WRITE_EN 37
INTERBUS_INFO S90 6
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

CANopen
Function Block Library
90 Series

CANopen
90 Series
# Contents

**The CANopen library** 3  
PREREQUISITES FOR USING THE LIBRARY 3  
COMPONENTS OF THE LIBRARY 3  
Function blocks 3  
Data types 4  
GENERAL NOTES REGARDING CAN BLOCKS 5  
CAN 2.0 A - 11 bit identifier 5  
CAN 2.0 B - 29 bit identifier 5  
DESCRIPTION OF CAN DATA TYPES 6  
CAN2A_IDENT_TYPE 6  
CAN2A_MESSAGE_TYPE 7  
CANopen_IDENT_TYPE 8  
CANopen_MESSAGE_TYPE 9  
CAN2B_MESSAGE_TYPE 10  
ERROR MESSAGES OF THE CANOPEN BLOCKS AND CAN BLOCKS 11  
Errors at block inputs 11  
Execution errors, coupler or slave responses 11  

**Overview of blocks arranged according to their call names** 12  

## Block descriptions

<table>
<thead>
<tr>
<th>Block Description</th>
<th>Function Block Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>READING INFORMATION ABOUT THE INSTALLED COUPLERS</td>
<td>CANOPEN_INFO 13</td>
</tr>
<tr>
<td>POLLING DIAGNOSTIC DATA FROM A SLAVE</td>
<td>CANOM_NODEDIAG 15</td>
</tr>
<tr>
<td>RESETTING THE COUPLER’S ERROR INDICATIONS</td>
<td>CANOM_RES_ERR 19</td>
</tr>
<tr>
<td>READING THE CANOPEN COUPLER STATUS</td>
<td>CANOM_STAT 21</td>
</tr>
<tr>
<td>READING A STATUS SURVEY OF ALL SLAVES</td>
<td>CANOM_SYSDIAG 26</td>
</tr>
<tr>
<td>READING AN OBJECT FROM A SLAVE</td>
<td>CANOM_SDO_READ 28</td>
</tr>
<tr>
<td>WRITING AN OBJECT TO A SLAVE</td>
<td>CANOM_SDO_WRITE 31</td>
</tr>
<tr>
<td>CONTROLLING NMT NODE STATES VIA NETWORK MANAGEMENT</td>
<td>CANOM_NMT 34</td>
</tr>
<tr>
<td>GENERATING A GENERAL CAN TELEGRAM HEADER ACCORDING TO CAN 2.0 A</td>
<td>IDENT_CAN2A_TO_WORD 36</td>
</tr>
<tr>
<td>SPLITTING A GENERAL CAN TELEGRAM HEADER ACCORDING TO CAN 2.0 A</td>
<td>WORD_TO_IDENT_CAN2A 37</td>
</tr>
<tr>
<td>GENERATING A CANOPEN-SPECIFIC CAN TELEGRAM HEADER</td>
<td>IDENT_CANOPEN_TO_WORD 38</td>
</tr>
<tr>
<td>SPLITTING A CANOPEN-SPECIFIC CAN TELEGRAM HEADER</td>
<td>WORD_TO_IDENT_CANOPEN 39</td>
</tr>
<tr>
<td>RECEIVING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS ACCORDING TO CAN 2.0 A</td>
<td>CAN_REC_2A 40</td>
</tr>
<tr>
<td>ENABLING OF IDENTIFIERS FOR RECEIVING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS ACCORDING TO CAN 2.0 A</td>
<td>CAN_REC_FILTER_2A 43</td>
</tr>
<tr>
<td>TRANSMITTING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS ACCORDING TO CAN 2.0 A</td>
<td>CAN_SEND_2A 46</td>
</tr>
<tr>
<td>RECEIVING CAN TELEGRAMS WITH 29 BIT IDENTIFIERS ACCORDING TO CAN 2.0 B</td>
<td>CAN_REC_2B 49</td>
</tr>
<tr>
<td>TRANSMITTING CAN TELEGRAMS WITH 29 BIT IDENTIFIERS ACCORDING TO CAN 2.0 B</td>
<td>CAN_SEND_2B 52</td>
</tr>
</tbody>
</table>
Glossary 54
Index 56
The CANopen library

PREREQUISITES FOR USING THE LIBRARY
The function blocks contained in the CANopen library access both, the PLC run time system as well as directly the coupler. The definitions and functions required for this are stored in the internal library Coupler_S90_Vxx.LIB.

When using the CANopen library CANopen_Master_S90_V50.LIB, it is absolutely essential to insert also the library Coupler_S90_V50.LIB into the project.

COMPONENTS OF THE LIBRARY
The library contains function blocks which allow an easy handling of the CANopen coupler. Additionally various data types are defined in this library. These structures enable a clear presentation of data sets.

Function blocks
The CANopen library contains the following function blocks:

| Group: General | CANopen_INFO | Reading information about the installed couplers |
| Group: CANopen master | CANoM_NODEDIAG | Polling diagnostic data from a slave |
| | CANoM_RES_ERR | Resetting internal error indications and counters |
| | CANoM_STAT | Reading the CANopen coupler status |
| | CANoM_SYSDIAG | Displaying status information of all slaves |
| SDO | CANoM_SDO_READ | Reading the value of a slave object |
| | CANoM_SDO_WRITE | Writing the value of a slave object |
| NMT | CANoM_NMT | Changing the operational state of one or all slaves |

Group: CAN 2.0 A

| CAN_REC_2A | Receiving CAN 2.0 A telegrams (11 bit identifier) |
| CAN_REC_FILTER_2A | Setting up receive filters for CAN 2.0 A telegram |
| CAN_SEND_2A | Transmitting CAN 2.0 A telegrams (11 bit identifier) |

Group: CAN 2.0 B

| CAN_REC_2B | Receiving CAN 2.0 B telegrams (29 bit identifier) |
| CAN_SEND_2B | Transmitting CAN 2.0 B telegrams (29 bit identifier) |

Auxiliary functions

| IDENT_CAN2A_TO_WORD | Generating a CAN 2.0 A telegram header |
| WORD_TO_IDENT_CAN2A | Splitting a CAN 2.0 A telegram header into individual components |
| IDENT_CANopen_TO_WORD | Generating a CANopen telegram header according to CAN 2.0 A |
| WORD_TO_IDENT_CANopen | Splitting a CANopen telegram header into individual components |

Detailed information about the various blocks can be found in the following sections.
Data types

The following data types (structures) are defined in the CANopen library:

**CANopen master**

- **CANoM_COM_ERR_TYPE** - Communication error
- **CANoM_EMCY_TYPE** - Emergency telegram
- **CANoM_STATE_BITS_TYPE** - Bits for coupler state description
- **NODESTATUS_1_TYPE** - Node diagnostics

**CAN**

- **CAN2A_IDENT_TYPE** - Components of an 11 bit identifier according to CAN 2.0
- **CAN2A_MESSAGE_TYPE** - Telegram structure according to CAN 2.0 A
- **CAN2B_MESSAGE_TYPE** - Telegram structure according to CAN 2.0 B
- **CANopen_IDENT_TYPE** - Components of a CAN telegram header according to CAN 2.0 A
- **CANopen_MESSAGE_TYPE** - Structure of a CANopen telegram with 11 bit identifier acc. to CAN 2.0 A

The data type descriptions for the CANopen master can be found at the corresponding block descriptions. The CAN data types are described in the following section.
GENERAL NOTES REGARDING CAN BLOCKS

Note:
The CAN blocks represent additional functionality. They are not required for normal CANopen operation.

The data exchange between the controller working as a CANopen master and the connected nodes is normally configured using 907 FB 1131. The configured nodes are taken into operation by the coupler when the program starts. During running operation, data exchange is performed automatically. The I/O data of the nodes can be accessed like variables without any use of additional blocks. If the PLC program is stopped, the master shuts down the bus in a controlled manner. The CAN blocks can be used to implement further functions which are not contained in the configuration. Alternatively it is also possible to do without the configuration and to completely implement the required functions of the CANopen protocol using these blocks.

In this case, the use of the CAN blocks is not restricted to the transmission and reception of CANopen telegrams. They are also able to handle functions of other CAN-based protocols.

The CAN blocks are able to transmit and receive any CAN telegrams. Messages with 11 bit identifiers according to CAN 2.0 A are as well supported as messages with 29 bit identifiers according to CAN 2.0 B.

CAN is the basis for various other protocols (SDS, DeviceNet, CANopen). A common point of all protocols is the telegram structure consisting of 11 bit or 29 bit identifiers and of up to 8 data bytes. Use and meaning of the individual telegram components differ from protocol to protocol. Therefore, the transmitted and received telegrams are forwarded transparent and not interpreted by the blocks to provide universal use of the CAN blocks. For a more comfortable handling of CANopen telegrams, additional auxiliary functions and data structures are available.

**CAN 2.0 A - 11 bit identifier**

**Data types for CAN 2.0 A blocks**

For a more comfortable handling of the CAN blocks, various data types describing CAN telegrams as well as various conversion auxiliary functions are defined in the CAN library. For the description of a CAN 2.0 A telegram the library contains two equivalent data types which can be used alternatively. The data type CANopen_MESSAGE_TYPE considers the special CAN-based telegram format used by CANopen whereas CAN2A_MESSAGE_TYPE describes the generally applicable format. The telegram format used by CANopen can be gathered from the CANopen specification.

**CAN 2.0 B - 29 bit identifier**

**Data types for CAN 2.0 B blocks**

For a more comfortable handling of the CAN blocks, various data types describing CAN telegrams as well as various conversion auxiliary functions are defined in the CAN library. The library contains only one general data type CAN2B_MESSAGE_TYPE to describe a CAN 2.0 B telegram with 29 bit identifier. Since CANopen is based on 11 bit identifiers, no additional CANopen-specific data type exists.
DESCRIPTION OF CAN DATA TYPES

CAN2A_IDENT_TYPE

The data type CAN2A_IDENT_TYPE contains all elements of a general CAN 2.0 A telegram header consisting of an 11 bit identifier, an RTR bit and the data length code. It is defined as follows in the CANopen library. Using the function IDENT_CAN2A_TO_WORD the individual elements can be put together to one CAN telegram header in the WORD format as it is used by the CAN blocks or the message types. When CAN 2.0 A telegrams are received, the header is first in the WORD format, too. Using the function WORD_TO_IDENT_CAN2A it can be converted into the data type CAN2A_IDENT_TYPE and thus be easily decoded.

The special structure of the identifier used with CANopen is defined in the CANopen specification. If CANopen functions shall be implemented using CAN blocks, it is therefore recommended to use the data type CANopen_IDENT_TYPE instead.

TYPE CAN2A_IDENT_TYPE:

STRUCT
  ID: WORD;
  RTR: BOOL;
  DLC: BYTE;
END_STRUCT
END_TYPE

ID WORD
ID contains the 11 bit identifier.

RTR BOOL
RTR contains the 'Remote Transmission Request' bit.

DLC BYTE
DLC contains the 4 bits wide 'Data Length Code' which defines the data length following the header within the CAN telegram. Apart from the telegram header, CAN telegrams can contain 0 to 8 bytes of data.
CAN2A_MESSAGE_TYPE

The data type CAN2A_MESSAGE_TYPE describes the general structure of a CAN 2.0 A telegram with an 11 bit identifier. It is defined as follows in the CANopen library.

The special CAN telegram structure used with CANopen is defined in the CANopen specification. If CANopen functions shall be implemented using CAN blocks, it is therefore recommended to use the data type CANopen_MESSAGE_TYPE instead.

TYPE CAN2A_MESSAGE_TYPE:
  STRUCT
    IDENT: WORD;
    DATA: ARRAY [1..8] OF BYTE;
  END_STRUCT
  END_TYPE

IDENT WORD

IDENT contains the general CAN telegram header consisting of the identifier (11 bit), the data length code (DLC) and the remote transmission request (RTR) in WORD format. Using the function IDENT_CAN2A_TO_WORD, IDENT can be easily composed from the individual components. Conversely IDENT can also be split into its individual components using the function WORD_TO_IDENT_CAN2A, if required. IDENT has the following format:

| DLC.3 | DLC.2 | DLC.1 | DLC.0 | RTR  | ID.10 | ID.9  | ID.8  | ID.7  | ID.6  | ID.5  | ID.4  | ID.3  | ID.2  | ID.1  | ID.0  |
|-------|-------|-------|-------|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 15    | 14    | 13    | 12    | 11   | 10    | 9     | 8     | 7     | 6     | 5     | 4     | 3     | 2     | 1     | 0     |

DATA ARRAY[1..8] OF BYTE

DATA contains the data of a general CAN telegram (if available). A CAN telegram can contain 0 to 8 bytes of data. The actual data length of a telegram is described by the data length code (DLC) in the telegram header (refer to IDENT). In DATA, only the first bytes as specified by the DLC are valid.
CANopen_IDENT_TYPE

The data type CANopen_IDENT_TYPE contains the individual CAN 2.0 A telegram header elements as they are especially used with CANopen. It consists of the function code, the node ID, the RTR bit and the data length code. The meaning of the individual elements is described in the CANopen specification. Using the function IDENT_CANopen_TO_WORD the individual elements can be put together to one CANopen telegram header in the WORD format as it is used by the CAN blocks or the message types. When CAN 2.0 A telegrams are received, the header is first in the WORD format, too. Using the function WORD_TO_IDENT_CANopen it can be converted into the data type CANopen_IDENT_TYPE and thus be easily decoded.

If CANopen functions shall be implemented by means of CAN blocks, it is recommended to use the special data type CANopen_IDENT_TYPE. If functions shall be implemented the telegram structure of which does not match the CANopen structure, the generally applicable data type CAN2A_IDENT_TYPE can be used instead.

```
TYPE CANopen_IDENT_TYPE:
  STRUCT
    FCT: BYTE;
    NODE: BYTE;
    RTR: BOOL;
    DLC: BYTE;
  END_STRUCT
END_TYPE
```

**FCT** BYTE

FCT contains the 4 bits wide 'Function Code'. Together with the node ID, the function code represents the 11 bit identifier.

**NODE** BYTE

NODE contains the node ID (0-127) of a CAN node. Together with the function code, the node ID represents the 11 bit identifier.

**RTR** BOOL

RTR contains the 'Remote Transmission Request' bit.

**DLC** BYTE

DLC contains the 4 bits wide 'Data Length Code' which defines the data length following the header within the CANopen telegram. Apart from the telegram header, a CANopen telegram can contain 0 to 8 bytes of data, where the first 4 bytes are used for special functions and are not available for user data (refer to CANopen_MESSAGE_TYPE).
CANopen MESSAGE_TYPE

The data type CANopen MESSAGE_TYPE describes the special structure of a CANopen telegram with 11 bit identifier acc. to CAN 2.0 A.

The special CAN telegram structure used with CANopen is defined in the CANopen specification. This data type is recommended if CANopen functions shall be implemented using CAN blocks. Implementation of other telegram formats can be done using the data type CAN2A MESSAGE_TYPE.

TYPE CAN2A MESSAGE TYPE:

STRUCT
  IDENT: WORD;
  CMD: BYTE;
  OBJ_IDX: WORD;
  SUB_IDX: BYTE;
  DATA: ARRAY [1..4] OF BYTE;
END_STRUCT
END_TYPE

IDENT WORD

IDENT contains the CANopen-specific CAN telegram header consisting of the node ID, the function code, the data length code (DLC) and the remote transmission request (RTR) in WORD format. Using the function IDENT_CANopen_TO_WORD, IDENT can be easily composed from the individual components. Conversely IDENT can also be split into its individual components using the function WORD_TO_IDENT_CANopen, if required.

IDENT has the following format:

<table>
<thead>
<tr>
<th>DLC.3</th>
<th>DLC.2</th>
<th>DLC.1</th>
<th>DLC.0</th>
<th>RTR</th>
<th>FCT.3</th>
<th>FCT.2</th>
<th>FCT.1</th>
<th>FCT.0</th>
<th>ID.6</th>
<th>ID.5</th>
<th>ID.4</th>
<th>ID.3</th>
<th>ID.2</th>
<th>ID.1</th>
<th>ID.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

CMD BYTE

CMD contains the 'Command' according to the CANopen specification.

OBJ_IDX WORD

OBJ_IDX contains the index of the object to be accessed.

SUB_IDX BYTE

SUB_IDX contains the sub-index of the object to be accessed.

DATA ARRAY[1..4] OF BYTE

DATA contains the data of a CANopen-specific CAN telegram (if available). For CANopen, the first four bytes of a CAN telegram are pre-assigned. Therefore, a maximum of 4 bytes are only available for the user data. The actual data length of a telegram is described by the data length code (DLC) in the telegram header (refer to IDENT). In DATA, only the first bytes are valid as specified by the DLC (minus the 4 pre-assigned bytes).
CAN2B_MESSAGE_TYPE

The data type CAN2B_MESSAGE_TYPE describes the general structure of a CAN 2.0 B telegram with a 29 bit identifier. It is defined as follows in the CANopen library.

TYPE CAN2B_MESSAGE_TYPE:
  STRUCT
    IDENT: DWORD;
    RTR: BOOL;
    DLC: BYTE;
    DATA: ARRAY [1..8] OF BYTE;
  END_STRUCT
END_TYPE

IDENT DWORD

IDENT contains the general 29 bit identifier, its value range reaches from 0 to 536870911 (16#0 to 16#1FFFFFFF).

RTR BOOL

RTR (Remote Transmission Request) contains the RTR bit in the telegram header.

DLC BYTE

DLC contains the data length code in the telegram header and specifies the valid length in bytes for the user data following in DATA. Valid values for DLC are 0 to 8.

DATA ARRAY[1..8] OF BYTE

DATA contains the telegram data (if available). A CAN telegram can contain 0 to 8 bytes of data. The actual data length of a telegram is described by the data length code (DLC) in the telegram header (refer to DLC). In DATA, only the first bytes are valid as specified by the DLC.
ERROR MESSAGES OF THE CANOPEN BLOCKS AND CAN BLOCKS

If the output ERR of a block is unequal to 0, either one of the values at the block inputs is invalid or an internal error occurred while executing the block. The error cause can be determined with the help of the following tables.

Errors at block inputs

<table>
<thead>
<tr>
<th>ERR</th>
<th>DEC</th>
<th>HEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8193</td>
<td>2001</td>
<td>0x01</td>
<td>Invalid value at block input CONO</td>
</tr>
<tr>
<td>8194</td>
<td>2002</td>
<td>0x02</td>
<td>Coupler model in slot CONO does not correspond to the block type</td>
</tr>
<tr>
<td>8195</td>
<td>2003</td>
<td>0x03</td>
<td>No CANopen master coupler in slot CONO</td>
</tr>
<tr>
<td>8196</td>
<td>2004</td>
<td>0x04</td>
<td>CANopen master coupler in slot CONO is not in OPERATE state</td>
</tr>
<tr>
<td>8197</td>
<td>2005</td>
<td>0x05</td>
<td>Invalid status survey type</td>
</tr>
<tr>
<td>8198</td>
<td>2006</td>
<td>0x06</td>
<td>Invalid node address / invalid data length code at block input</td>
</tr>
<tr>
<td>8201</td>
<td>2009</td>
<td>0x09</td>
<td>Invalid command at block input</td>
</tr>
<tr>
<td>8202</td>
<td>200A</td>
<td></td>
<td>Invalid data length at block input</td>
</tr>
</tbody>
</table>

Execution errors, coupler or slave responses

<table>
<thead>
<tr>
<th>ERR</th>
<th>DEC</th>
<th>HEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>9219</td>
<td>2403</td>
<td>0x03</td>
<td>Service denied from slave with Abort SDO Service</td>
</tr>
<tr>
<td>9233</td>
<td>2411</td>
<td>0x11</td>
<td>No slave response, timeout</td>
</tr>
<tr>
<td>9235</td>
<td>2413</td>
<td>0x13</td>
<td>Slave not in operational mode, access denied</td>
</tr>
<tr>
<td>9267</td>
<td>2433</td>
<td>0x33</td>
<td>Data receive buffer overflow</td>
</tr>
<tr>
<td>9279</td>
<td>2435</td>
<td>0x35</td>
<td>Amount of fragmented data exceeds receive buffer size</td>
</tr>
<tr>
<td>9270</td>
<td>2436</td>
<td>0x36</td>
<td>Unknown request or service still in process</td>
</tr>
<tr>
<td>9271</td>
<td>2437</td>
<td>0x37</td>
<td>Node address out of valid range</td>
</tr>
<tr>
<td>9273</td>
<td>2439</td>
<td>0x39</td>
<td>Sequence error in fragmented protocol, request aborted</td>
</tr>
<tr>
<td>9377</td>
<td>24A1</td>
<td></td>
<td>Node address out of valid range</td>
</tr>
<tr>
<td>9416</td>
<td>24C8</td>
<td></td>
<td>Coupler not configured, database not found</td>
</tr>
</tbody>
</table>
Overview of blocks arranged according to their call names

Character description:

FBhv … Function block with historical values
FBnohv … Function block without historical values
F … Function

<table>
<thead>
<tr>
<th>CE name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANopen_INFO</td>
<td>FBnohv</td>
<td>Reading information about the installed couplers</td>
<td>13</td>
</tr>
<tr>
<td>CAN_REC_2A</td>
<td>FBhv</td>
<td>Receiving CAN 2.0 A telegrams (11 bit identifier)</td>
<td>40</td>
</tr>
<tr>
<td>CAN_REC_2B</td>
<td>FBhv</td>
<td>Receiving CAN 2.0 B telegrams (29 bit identifier)</td>
<td>52</td>
</tr>
<tr>
<td>CAN_REC_FILTER_2A</td>
<td>FBhv</td>
<td>Setting up receive filters for CAN 2.0 A telegram</td>
<td>49</td>
</tr>
<tr>
<td>CAN_SEND_2A</td>
<td>FBhv</td>
<td>Transmitting CAN 2.0 A telegrams (11 bit identifier)</td>
<td>46</td>
</tr>
<tr>
<td>CAN_SEND_2B</td>
<td>FBhv</td>
<td>Transmitting CAN 2.0 B telegrams (29 bit identifier)</td>
<td>52</td>
</tr>
<tr>
<td>CANoM_NMT</td>
<td>FBhv</td>
<td>Changing the operational state of one or all slaves</td>
<td>34</td>
</tr>
<tr>
<td>CANoM_NODEDIAG</td>
<td>FBhv</td>
<td>Polling diagnostic data from a slave</td>
<td>15</td>
</tr>
<tr>
<td>CANoM_SDO_READ</td>
<td>FBhv</td>
<td>Reading the value of a slave object</td>
<td>28</td>
</tr>
<tr>
<td>CANoM_SDO_WRITRE</td>
<td>FBhv</td>
<td>Writing the value of a slave object</td>
<td>31</td>
</tr>
<tr>
<td>CANoM_RES_ERR</td>
<td>FBhv</td>
<td>Resetting internal error indications and counters</td>
<td>19</td>
</tr>
<tr>
<td>CANoM_STAT</td>
<td>FBnohv</td>
<td>Reading the CANopen coupler status</td>
<td>21</td>
</tr>
<tr>
<td>CANoM_SYSDIAG</td>
<td>FBnohv</td>
<td>Displaying status information of all slaves</td>
<td>26</td>
</tr>
<tr>
<td>IDENT_CAN2A_TO_WORD</td>
<td>F</td>
<td>Generating a CAN 2.0 A telegram header</td>
<td>36</td>
</tr>
<tr>
<td>IDENT_CANopen_TO_WORD</td>
<td>F</td>
<td>Generating a CANopen telegram header according to CAN 2.0 A</td>
<td>38</td>
</tr>
<tr>
<td>WORD_TO_IDENT_CAN2A</td>
<td>F</td>
<td>Splitting a CAN 2.0 A telegram header into individual components</td>
<td>37</td>
</tr>
<tr>
<td>WORD_TO_IDENT_CANopen</td>
<td>F</td>
<td>Splitting a CANopen telegram header into individual components</td>
<td>39</td>
</tr>
</tbody>
</table>

Block descriptions
The block CANopen_INFO outputs coupler-related information. The following items are displayed: Coupler type and model, operation mode, manufacturing date, device number and serial number as well as firmware designation and firmware version.

### Block data

**Available as of PLC runtime system:** V5.0  
**Included in library:** CANopen_S90_V50.LIB  
**Remark:**

### Block type

Function block without historical values

### Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>CANopen_INFO</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>TYP</td>
<td>STRING(17)</td>
<td>Coupler type</td>
</tr>
<tr>
<td>MODEL</td>
<td>STRING(22)</td>
<td>Coupler model</td>
</tr>
<tr>
<td>OP_MODE</td>
<td>STRING(22)</td>
<td>Operation mode of the coupler</td>
</tr>
<tr>
<td>MAN_DATE</td>
<td>DATE</td>
<td>Manufacturing date of the coupler</td>
</tr>
<tr>
<td>DEV_NO</td>
<td>DWORD</td>
<td>Device number of the coupler</td>
</tr>
<tr>
<td>SER_NO</td>
<td>DWORD</td>
<td>Serial number of the coupler</td>
</tr>
<tr>
<td>FW_NAME</td>
<td>STRING(17)</td>
<td>Designation of the coupler firmware</td>
</tr>
<tr>
<td>FW_VER</td>
<td>STRING(17)</td>
<td>Version of the coupler firmware</td>
</tr>
</tbody>
</table>

### Description

The block CANopen_INFO is always active. It reads the slot number at the block input CONO and outputs the corresponding information about the selected coupler.

CANopen_INFO recognizes different coupler types. For that reason, the block outputs do not only relate to the CANopen coupler.

The block is not intended for use in normal user programs. It should be used to support fault diagnosis and maintenance operations.

### CONO BYTE

At input CONO the slot (module number) of the coupler is applied for which the information should be polled. The module number depends on the PLC category.

R018X CONO = 1  
R01X8 CONO = 2

### ERR WORD

An error identification is applied at output ERR, if an invalid value is applied at input CONO. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.
**TYP** STRING(17)

Output TYP displays the coupler type (i.e. the coupler design). Different types are recognized by the block.

The model ID of the CANopen coupler is MODEL = ‘CANopen Master’.

If the block detects an unknown model, MODEL outputs the value ‘unknown’.

**OP_MODE** STRING(22)

Output OP_MODE displays the current setting for the coupler operation mode. This output is only applicable for couplers with switchable operation modes.

Therefore, this output is not used for the CANopen coupler.

**MAN_DATE** DATE

Output MAN_DATE displays the manufacturing date of the coupler. The date is a variable of the data type DATE and has the format D#YYYY-MM-DD. Its initial value is D#2000-01-01.

**DEV_NO** DWORD

Output DEV_NO outputs the device number of the coupler.

**SER_NO** DWORD

Output SER_NO outputs the serial number of the coupler.

**FW_NAME** STRING(17)

At output FW_NAME the designation of the coupler firmware is applied.

**FW_VER** STRING(17)

Output FW_VER displays the firmware release number (version) and the firmware issue date of the coupler. These data are represented as a string (e.g. ‘V1.003 15.07.00’).

---

**Function call in IL**

CAL INFO(CONO := INFO_CONO)
LD INFO.ERR
ST INFO.ERR
LD INFO.TYP
ST INFO.TYP
LD INFO.MODEL
ST INFO.MODEL
LD INFO.OP_MODE
ST INFO.OP_MODE
LD INFO.MAN_DATE
ST INFO.MAN_DATE
LD INFO.DEV_NO
ST INFO.DEV_NO
LD INFO.SER_NO
ST INFO.SER_NO
LD INFO.FW_NAME
ST INFO.FW_NAME
LD INFO.FW_VER
ST INFO.FW_VER

**Function call in ST**

INFO(CONO := INFO_CONO);
INFO.ERR:=INFO.ERR;
INFO.TYP:=INFO.TYP;
INFO.MODEL:=INFO.MODEL;
INFO.OP_MODE:=INFO.OP_MODE;
INFO.MAN_DATE:=INFO.MAN_DATE;
INFO.DEV_NO:=INFO.DEV_NO;
INFO.SER_NO:=INFO.SER_NO;
INFO.FW_NAME:=INFO.FW_NAME;
INFO.FW_VER:=INFO.FW_VER;

**Note:**
In IL, the function call has to be performed in one line.
The block CANoM_NODEDIAG reads the diagnostic data of a slave.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>CANoM_NODEDIAG Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE Slot (module number) of the coupler</td>
</tr>
<tr>
<td>NODE</td>
<td>BYTE Node number of the concerning slave</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD Error message</td>
</tr>
<tr>
<td>STAT_1</td>
<td>NODESTATUS_1_TYPE Nodestatus_1, slave diagnostics</td>
</tr>
<tr>
<td>STATE</td>
<td>BYTE Operating condition of the slave</td>
</tr>
<tr>
<td>ONL_ERR</td>
<td>BYTE Online error of the slave</td>
</tr>
<tr>
<td>NUM_EMCY</td>
<td>BYTE Number of emergency messages in EMCY_DATA</td>
</tr>
<tr>
<td>EMCY_DATA</td>
<td>ARRAY[1..5] OF CANoM_EMCY_TYPE Content of the emergency messages</td>
</tr>
</tbody>
</table>

Using the block CANoM_NODEDIAG, diagnostic data of the individual slaves can be requested.

Every time a FALSE → TRUE edge is applied to the input EN, CANoM_NODEDIAG reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active requests is finished. The completion of the request processing is indicated by DONE = TRUE.

If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.
CONO BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R018X CONO = 1
R01X8 CONO = 2

NODE BYTE
The node ID of the slave the diagnostic data of which are to be requested, is specified at input NODE.

DONE BOOL
Output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If the processing of the task is completed, DONE is set to TRUE for one cycle period.

The values applied at the block outputs are only valid if DONE = TRUE. Additionally, the diagnostic data are only valid if ERR = 0. If ERR is not 0, an error occurred.

ERR WORD
At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing the job. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

STAT_1 NODESTATUS_1_TYPE
STAT_1 outputs different bits as a structure of the type NODESTATUS_1_TYPE which display error states of the slave. STAT_1 is only valid if DONE = TRUE and ERR = 0.

The format of the structure of the type NODESTATUS_1_TYPE is defined in the CANopen library (see description below).

STATE BYTE
STATE outputs the current operating condition of the relevant slave. STATE is only valid if DONE = TRUE and ERR = 0.

The IDs of the individual states are described on the following pages.

ONL_ERR BYTE
ONL_ERR outputs a value which describes possible existing communication errors between master coupler and slave. ONL_ERR is only valid if DONE = TRUE and ERR = 0.

The error IDs of ONL_ERR correspond to the IDs of the CANoM_STAT block output CANoM_ERR.EVENT. They are described in the table provided in the CANoM_STAT block description.

NUM_EMCY BYTE
NUM_EMCY outputs the number of valid emergency messages of the slave output in EMCY_DATA according to the CANopen specification. Up to 5 emergency messages per slave can be buffered in the coupler.

NUM_EMCY is only valid if DONE = TRUE and ERR = 0.

EMCY_DATA ARRAY[1..5] OF CANoM_EMCY_TYPE
EMCY_DATA outputs up to 5 buffered emergency messages of the slave. The number of valid messages is output by NUM_EMCY. The format of the structure of the type CANoM_EMCY_TYPE is defined in the CANopen library (see description below). EMCY_DATA is only valid if DONE = TRUE and ERR = 0.
Function call in IL

CAL DIAG(EN := DIAG_EN,
CONO := DIAG_CONO,
NODE := DIAG_NODE)

LD DIAG_DONE
ST DIAG_DONE
LD DIAG_ERR
ST DIAG_ERR
LD DIAG_STAT_1
ST DIAG_STAT_1
LD DIAG_STATE
ST DIAG_STATE
LD DIAG_ONL_ERR
ST DIAG_ONL_ERR
LD DIAG_NUM_EMCY
ST DIAG_NUM_EMCY
LD DIAG_EMCY_DATA
ST DIAG_EMCY_DATA

Note:
In IL, the function call has to be performed in one line.

Function call in ST

DIAG
(EN := DIAG_EN,
CONO := DIAG_CONO,
NODE := DIAG_NODE);

DIAG_DONE := DIAG_DONE;
DIAG_ERR := DIAG_ERR;
DIAG_STAT_1 := DIAG_STAT_1;
DIAG_STATE := DIAG_STATE;
DIAG_ONL_ERR := DIAG_ONL_ERR;
DIAG_NUM_EMCY := DIAG_NUM_EMCY;
DIAG_EMCY_DATA := DIAG_EMCY_DATA;

Nodestatus_1 STAT_1 NODESTATUS_1_TYPE

Output STAT_1 of CANoM.NODEDIAG displays different diagnostic bits as a structure of the type NODESTATUS_1_TYPE. Within the CANopen library the structure NODESTATUS_1_TYPE is declared as follows:

TYPE NODESTATUS_1_TYPE:
STRUCT
NO_RESPONSE: BOOL;
EMCY_OVF: BOOL;
PRM_FAULT: BOOL;
GUARD_ACT: BOOL;
reserved1: BOOL;
reserved2: BOOL;
reserved3: BOOL;
DEACTIVATED: BOOL;
END_STRUCT
END_TYPE

NO_RESPONSE BOOL
If this bit is set, the slave with the node number specified at block input NODE does not respond to the master requests. Normally NO_RESPONSE should be set to FALSE.

EMCY_OVF BOOL
This bit is set by the coupler, if more emergency messages were received from the called slave than the buffer can store (refer to block inputs NUM_EMCY and EMCY_DATA).

PRM_FAULT BOOL
This bit is set, if the actual slave configuration differs from the nominal slave configuration in the master.

GUARD_ACT BOOL
This bit is set by the coupler, if the node guarding protocol for this slave is active. This is only a status indication. The active node guarding protocol between master and slave is not synonymous with a node guarding error.

reserved1 BOOL
reserved2 BOOL
reserved3 BOOL
These bits are reserved and are currently not used.
DEACTIVATED BOOL
This bit is set to TRUE, if the slave defined in the configuration data of the master is deactivated and not processed.

STATE
Output STATE indicates the current operating condition of the concerning slave. The following table describes the possible values of STATE and their meanings as specified in the CANopen specification.

<table>
<thead>
<tr>
<th>STATE</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Disconnected</td>
</tr>
<tr>
<td>2</td>
<td>Connecting</td>
</tr>
<tr>
<td>3</td>
<td>Preparing</td>
</tr>
<tr>
<td>4</td>
<td>Prepared</td>
</tr>
<tr>
<td>5</td>
<td>Operational</td>
</tr>
<tr>
<td>127</td>
<td>Pre-Operational</td>
</tr>
</tbody>
</table>

EMCY_DATA
EMCY_DATA outputs the up to 5 (refer to NUM_EMCY) emergency messages received last from the slave. EMCY_DATA consists of one ARRAY [1..5] OF CANoM_EMCY_TYPE. The data type CANoM_EMCY_TYPE corresponds to the format of the emergency telegram described in the CANopen communication profile and is defined as follows in the CANopen library:

```c
TYPE CANoM_EMCY_TYPE:
    STRUCT
        ERROR_CODE: WORD;
        ERROR_REG: BYTE;
        ERROR_DATA: ARRAY [1..5] OF BYTE;
    END_STRUCT
END_TYPE
```

ERROR_CODE WORD
For the emergency object the emergency error codes described in the following table are defined in the CANopen communication profile.

```
Emergency error code | Meaning / error cause
----------------------|---------------------------------------------
00000..00255         | Error on reset or no error                |
04096..04351         | General error                             |
08192..08447         | Current error                             |
08448..08703         | Error at the device input side            |
08704..08959         | Error inside the device                    |
08960..09215         | Error at the device output side           |
12288..12543         | Voltage error                             |
12544..12799         | Supply voltage error                      |
12800..13055         | Error inside the device                    |
13056..13311         | Error at the device output side           |
16384..16639         | Temperature error                         |
16640..16895         | Ambient temperature                       |
16896..17151         | Temperature inside the device             |
20480..20735         | Hardware error in the device              |
22457..24831         | Software error in the device              |
24832..25087         | Device-internal software                  |
25088..25343         | Application software                      |
25344..25999         | Data                                      |
28672..29827         | Error in additional modules               |
32768..33023         | Monitoring                                |
33024..33279         | Communication                            |
36864..37119         | External error                            |
61440..61695         | Error of additional functions             |
62820..65535         | Device-specific errors                    |
```

ERROR_REG BYTE
The variable ERROR_REG displays the error register value (object 1001 hex) of the slave. This value is transmitted by the slave as a part of the emergency message.

ERROR_DATA ARRAY[1..5] OF BYTE
If applicable, ERROR_DATA is used to output manufacturer-specific error information transmitted by the slave as part of the emergency message. For detailed information about the meaning of these data please refer to the particular device documentation.
RESETTING THE COUPLER’S ERROR INDICATIONS

The block CANoM_RES_ERR can be used to reset various internal error indications and counters of the coupler.

Block data
Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

Block type
Function block with historical values

Parameters
<table>
<thead>
<tr>
<th>Instance</th>
<th>CANoM_RES_ERR</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

Description
Using the block CANoM_RES_ERR the following internal error indications and counters of the coupler output via the block CANoM_STAT can be reset:
- STATE_BITS.EVENT
- STATE_BITS.TIMEOUT
- BUS_ERR
- BUS_OFF
- TO_ERR
- RX_OVF

For explanations of the error indications please refer to the description of the block CANoM_STAT.

The reset is initiated by a FALSE -> TRUE edge at input EN.

EN
If a FALSE -> TRUE edge is applied to input EN it is first checked whether the value at input CONO is valid. If the value is valid, a corresponding request message is sent to the coupler. While this request message is processed, output DONE is set to FALSE. If the value at input CONO is not valid, the request is not processed. In this case the error is immediately indicated at output ERR and the completion of the processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

CONO
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.
R018X CONO = 1
R01X8 CONO = 2

DONE
Output DONE reflects the state of the request processing. If the block is triggered by a FALSE -> TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing of the task is completed, DONE is set to TRUE for one cycle period.

If DONE = TRUE and ERR = 0, the error indications could be reset successfully. If ERR is not 0, an error occurred.
RESETTING THE COUPLER’S ERROR INDICATIONS

**ERR**

At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to input CONO or if an error occurred during processing of the job. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

**Function call in IL**

```
CAL  RES_ERR
  (EN := RES_ERR.EN,
   CONO := RES_ERR.CONO)
LD   RES_ERR.DONE
ST   RES_ERR.ERR
LD   RES_ERR_DONE
ST   RES_ERR_ERR
```

**Note:**

In IL, the function call has to be performed in one line.

**Function call in ST**

```
RES_ERR
  (EN := RES_ERR.EN,
   CONO := RES_ERR.CONO);
RES_Err_DONE := RES_Err_DONE;
RES_Err_ERR := RES_Err_ERR;
```
READING THE CANOPEN COUPLER STATUS

CANoM_STAT outputs the CANopen coupler status. The outputs provide information about the communication state and error events.

<table>
<thead>
<tr>
<th>Block data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available as of PLC runtime system: V5.0</td>
</tr>
<tr>
<td>Included in library: CANopen_S90_V50.LIB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function block without historical values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance CANoM_STAT</td>
</tr>
<tr>
<td>EN BOOL</td>
</tr>
<tr>
<td>CONO BYTE</td>
</tr>
<tr>
<td>ERR WORD</td>
</tr>
<tr>
<td>STATE_BITS CANoM_STATE_BITS_TY PE</td>
</tr>
<tr>
<td>CANoM_STATE BYTE</td>
</tr>
<tr>
<td>CANoM_ERR CANoM_COM_ERR_TYP E</td>
</tr>
<tr>
<td>BUS_ERR WORD</td>
</tr>
<tr>
<td>BUS_OFF WORD</td>
</tr>
<tr>
<td>TO_ERR WORD</td>
</tr>
<tr>
<td>RX_OVF WORD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The block CANoM_STAT outputs the current status of the CANopen coupler.</td>
</tr>
</tbody>
</table>

CANoM_STAT is active if input EN = TRUE. If the block is active, always the current values are displayed at the outputs.

<table>
<thead>
<tr>
<th>EN BOOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via the input EN. If the block is active, the current values are applied to the outputs.</td>
</tr>
</tbody>
</table>

CANoM_STAT

<table>
<thead>
<tr>
<th>CONO INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>At input CONO the slot (module number) of the coupler is applied for which the state should be read. The module number depends on the PLC category.</td>
</tr>
<tr>
<td>R018X CONO = 1</td>
</tr>
<tr>
<td>R01X8 CONO = 2</td>
</tr>
</tbody>
</table>
ERR  WORD
An error identification is output at ERR, if an invalid value is applied at a block input. ERR has always to be considered together with input EN. The value applied at ERR is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

STATE_BITS  CANoM_STATE_BITS_TYPE
STATE_BITS outputs atypical communication states of the CANopen coupler. STAT_BITS is only valid if EN = TRUE and ERR = 0.

The format of the structure of the type CANoM_STATE_BITS_TYPE is defined in the CANopen library (see description below).

Some of the error indications in STATE_BITS can be reset via the block CANoM_RES_ERR.

CANoM_STATE  BYTE
CANoM_STATE outputs the general communication state of the CANopen master. The following states are defined:

OFFLINE 00 HEX / 00 DEC
STOP 40 HEX / 64 DEC
CLEAR 80 HEX / 128 DEC
OPERATE C0 HEX / 192 DEC

CANoM_STATE = OFFLINE
If CANoM_STATE is set to OFFLINE, the CANopen coupler performs an initialization. After the initialization phase is completed the coupler changes to STOP state.

CANoM_STATE = STOP
If CANoM_STATE has the value STOP, the coupler is completely initialized. In this state the coupler is ready to receive configuration data. There is no data exchange with the slaves. The coupler has this state if no user program is running.

CANoM_STATE = CLEAR
If the user program is started, the coupler changes from STOP into CLEAR and starts to establish the connections defined during configuration. After the setup has been completed successfully, the coupler changes into OPERATE state. If an error occurs during the parameter setting, the coupler changes back to STOP state.

CANoM_STATE = OPERATE
Normally the coupler is in OPERATE state while a user program is running. In this state the master exchanges I/O data with the slaves. If an error occurs during this process and ‘Auto Clear Mode’ was selected while configuring, the coupler changes back to CLEAR state and tries again to establish the connections. If ‘Auto Clear Mode’ was not selected, the coupler remains (in case of an error) in OPERATE state. If the user program is stopped, the coupler also changes back to STOP state.

CANoM_STATE is only valid if EN = TRUE and ERR = 0.

CANoM_ERR  WORD
At output CANoM_ERR possible communication errors are displayed. CANoM_ERR is only valid if EN = TRUE and ERR = 0.

The format of the structure of the type CANoM_COM_ERR_TYPE is defined in the CANopen library and described below together with the possible errors.

BUS_ERR  WORD
BUS_ERR outputs the number of occurred bus failures. A bus failure occurs if the internal error frame counter exceeds a specific value. BUS_ERR is only valid if EN = TRUE and ERR = 0.

BUS_ERR can be reset using the block CANoM_RES_ERR.

BUS_OFF  WORD
BUS_OFF outputs how often the coupler was excluded from bus activities. An exclusion from bus activities will occur when the internal error frame counter is overflowed. The coupler is automatically re-initialized after each overflow. BUS_OFF is only valid if EN = TRUE and ERR = 0.

BUS_OFF can be reset using the block CANoM_RES_ERR.
TO_ERR outputs how many telegrams could not be transmitted successfully. Transmission of a telegram is considered as failed, if it could not be transmitted within 20 ms, for instance because the communication partner could not be contacted via the bus. TO_ERR is only valid if EN = TRUE and ERR = 0.

TO_ERR can be reset using the block CANoM_RES_ERR.

RX_OVF outputs how many received telegrams were dismissed because they could not be processed successfully due to an overload of the CAN chip. RX_OVF is only valid if EN = TRUE and ERR = 0.

RX_OVF can be reset using the block CANoM_RES_ERR.

Function call in IL

CAL STAT
(EN := STAT_EN,
CONO := STAT_CONO)
LD STAT.ERR
ST STAT_ERR
LD STAT.STATE_BITS
ST STAT_STATE_BITS
LD STAT.CANoM_STATE
ST STAT_CANoM_STATE
LD STAT.CANoM_ERR
ST STAT_CANoM_ERR
LD STAT.BUS_ERR
ST STAT_BUS_ERR
LD STAT.BUS_OFF
ST STAT_BUS_OFF
LD STAT.TO_ERR
ST STAT_TO_ERR
LD STAT.RX_OVF
ST STAT_RX_OVF

Note:
In IL, the function call has to be performed in one line.

Function call in ST

STAT
(EN := STAT_EN,
CONO := STAT_CONO);
STAT.ERR := STAT.ERR;
STAT.STATE_BITS := STAT.STATE_BITS;
STAT.CANoM_STATE := STAT.CANoM_STATE;
STAT.CANoM_ERR := STAT.CANoM_ERR;
STAT.BUS_ERR := STAT.BUS_ERR;
STAT.BUS_OFF := STAT.BUS_OFF;
STAT.TO_ERR := STAT.TO_ERR;
STAT_RX_OVF := STAT_RX_OVF;

STATE_BITS CANoM_STATE_BITS_TYPE

The structure STATE_BITS consists of six boolean variables which display different communication states. Within the CANopen library the data type CANoM_STATE_BITS_TYPE is declared as follows:

TYPE CANoM_STATE_BITS_TYPE:
STRUCT
CTRL: BOOL;
AUTO_CLR: BOOL;
NO_EXCH: BOOL;
FATAL: BOOL;
EVENT: BOOL;
TIMEOUT: BOOL;
END_STRUCT
END_TYPE

CTRL BOOL
If this bit is set to TRUE, a parameter setting error occurred. In normal operation CTRL should be FALSE. If this is not the case, the parameter and configuration data have to be checked.

AUTO_CLR BOOL
If AUTO_CLR is set to TRUE, the coupler has stopped the data exchange with all slaves due to communication errors and has changed back to CLEAR state (see CANoM_STATE).
**NO_EXCH**  **BOOL**

This bit is set to TRUE, if exchanging process data with one or several slaves is not possible. The error cause can be found in the configuration data as well as in the slaves.

**FATAL**  **BOOL**

If FATAL is set to TRUE, no communication via CANopen is possible due to a fatal internal error.

**EVENT**  **BOOL**

EVENT is set to TRUE if transmission errors are detected by the coupler. The number of occurred transmission errors is displayed at the corresponding outputs BUS_ERR and BUS_OFF. If the EVENT bit is set to TRUE, reset is only possible via the block CANoM_RES_ERR.

**TIMEOUT**  **BOOL**

If TIMEOUT is set to TRUE, transmission of at least one telegram failed. Transmission of this telegram was aborted and its content is lost. TIMEOUT = TRUE is an indication that the communication partner could not be contacted via the bus. The number of failed transmissions is displayed at output TO_ERR. If the TIMEOUT bit is set to TRUE, reset is only possible via the block CANoM_RES_ERR.

---

**CANoM_ERR**  **CANoM_COM_ERR_TYPE**

Communication errors can be located precisely via CANoM_ERR. The output CANoM_ERR is represented as a structure of the type CANoM_COM_ERR_TYPE. Within the CANopen library this data type is declared as follows:

```
TYPE CANoM_COM_ERR_TYPE:
  STRUCT
    ADDRESS: BYTE;
    EVENT:  BYTE;
  END_STRUCT
END_TYPE
```

**ADDRESS**  **BYTE**

In case of an error, ADDRESS contains the node address of the faulty subscriber. If ADDRESS has the value 255, the error is located in the coupler itself.

**ADDRESS = 255**  **Coupler error**

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>Unknown process data handshake mode</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>56</td>
<td>Invalid data transmission rate</td>
<td>Configuration</td>
<td>Contact Support</td>
</tr>
<tr>
<td>60</td>
<td>Duplicate node address configured</td>
<td>Configuration</td>
<td>Check node addresses of all devices specified in configuration data</td>
</tr>
<tr>
<td>210</td>
<td>No configuration data</td>
<td>Configuration/ Coupler</td>
<td>Load configuration data into coupler</td>
</tr>
<tr>
<td>212</td>
<td>Error while reading database</td>
<td>Configuration/ Coupler</td>
<td>Load configuration data into coupler once again. Contact Support.</td>
</tr>
<tr>
<td>220</td>
<td>Watchdog error</td>
<td>Basic unit</td>
<td>Contact Support</td>
</tr>
</tbody>
</table>
**ADDRESS <> 255**  
Error at subscriber with node address ADDRESS

<table>
<thead>
<tr>
<th>EVENT</th>
<th>Meaning</th>
<th>Error source</th>
<th>Cause / Remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>Slave monitoring timeout</td>
<td>Slave</td>
<td>Check connection and node address of the slave</td>
</tr>
<tr>
<td>31</td>
<td>Slave aborted operational mode</td>
<td>Slave</td>
<td>Reset slave</td>
</tr>
<tr>
<td>32</td>
<td>Sequence error in node guarding protocol</td>
<td>Slave</td>
<td>Reset slave</td>
</tr>
<tr>
<td>33</td>
<td>No response to configured remote frame PDO</td>
<td>Slave</td>
<td>Check whether the slave supports remote frames</td>
</tr>
<tr>
<td>34</td>
<td>No slave response during configuration</td>
<td>Slave</td>
<td>Check that the slave is connected and ready for operation</td>
</tr>
<tr>
<td>35</td>
<td>Actual device profile of the slave differs from the configured device profile</td>
<td>Configuration</td>
<td>Check the profile supported by the slave</td>
</tr>
<tr>
<td>36</td>
<td>Actual device type of the slave differs from the configured device type</td>
<td>Configuration</td>
<td>Check the services supported by the slave</td>
</tr>
<tr>
<td>37</td>
<td>Unknown SDO response received</td>
<td>Slave</td>
<td>Slave does not meet the specifications of the CiA protocol</td>
</tr>
<tr>
<td>38</td>
<td>Length indicator of a received SDO response is not 8</td>
<td>Slave</td>
<td>Slave does not meet the specifications of the CiA protocol</td>
</tr>
<tr>
<td>39</td>
<td>Slave is not processed. It is in the STOP state.</td>
<td>Configuration/ Coupler</td>
<td>Deactivate the 'Auto Clear Mode'</td>
</tr>
</tbody>
</table>
The block CANoM_SYSDIAG outputs a bit field as a state survey of all slaves (nodes) at output NODE. Three different surveys can be selected via input TYP.

**Block data**

Available as of PLC runtime system: V5.0

Included in library: CANopen_S90_V50.LIB

**Block type**

Function block without historical values

**Parameters**

- **Instance**: CANoM_SYSDIAG
- **EN**: BOOL
- **CONO**: BYTE
- **TYP**: BYTE
- **ERR**: WORD
- **NODE**: ARRAY [0..127] OF BOOL

**Description**

The block CANoM_SYSDIAG outputs different status surveys about all slaves. Three survey types can be selected:

- configuration survey
- operational survey
- diagnosis survey

The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via input EN. If the block is active, the current values are applied to the outputs.

At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R018X CONO = 1
R01X8 CONO = 2
The input TYP is used to select which status survey should be displayed at output NODE.

**TYP** = 1  
**configuration survey**  
Output NODE displays which slaves were successfully connected to the master (TRUE). Please note that the master establishes only a connection to those slaves which were announced to the master when defining the configuration data.

**TYP** = 2  
**operational survey**  
Output NODE displays which slaves are error-free and in operation. A slave can only be announced as operational if it has been configured in the master. The operational survey can only be requested if the coupler is in OPERATE state.

**TYP** = 3  
**diagnosis survey**  
Output NODE indicates which slaves report a diagnosis. The diagnosis survey can only be requested if the coupler is in OPERATE state.

**ERR** = WORD  
An error identification is applied at output ERR if an invalid value is applied at an input. ERR has always to be considered together with input EN. The value applied at ERR is only valid if EN = TRUE. All further outputs are only valid if EN = TRUE and ERR = 0.

The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

**NODE** = ARRAY [0..127] OF BOOL  
At output NODE the status survey is output as a bit field. Every bit within this field represents a slave. With this, the index corresponds to the slave's node address. A bit is set to TRUE, the state selected using TYP applies to the corresponding slave.

If e.g. **TYP** = 1 is selected and **NODE[2]** = TRUE, the slave with this node address was successfully configured by the master and is in operation. If **NODE[2]** = FALSE, the configuration of the specific slave has not yet been completed or the slave is not part of the master's configuration data.

If **TYP** = 3, e.g. **NODE[2]** = TRUE means that the slave with the node address 2 has received an emergency message or that the diagnostic indication of the slave has changed. The detailed diagnosis can then be requested using the block CANoM_NODEDIAG.

The bit field output at NODE is only valid if EN = TRUE and ERR = 0.

### Function call in IL

```
CAL SYSDIAG
(EN := SYSDIAG_EN,
 CONO := SYSDIAG_CONO,
 TYP := SYSDIAG_TYP)
LD SYSDIAG.ERR
ST SYSDIAG_ERR
LD SYSDIAG.NODE
ST SYSDIAG_NODE
```

**Note:**
In IL, the function call has to be performed in one line.
READING AN OBJECT FROM A SLAVE

The block CANoM_SDO_READ can be used to read service data objects (SDOs) individually from a slave.

Block data
Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

Remark:
Included in library: CANopen_S90_V50.LIB

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>CANoM_SDO_READ</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>NODE</td>
<td>BYTE</td>
<td>Node address of the concerning slave</td>
</tr>
<tr>
<td>OBJ_IDX</td>
<td>WORD</td>
<td>Object index of the object to be read</td>
</tr>
<tr>
<td>SUB_IDX</td>
<td>BYTE</td>
<td>Sub-index of the object to be read</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address from which on the read data are stored (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>SDO_ERR</td>
<td>DWORD</td>
<td>SDO abort error message of the slave</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of read data (byte value)</td>
</tr>
</tbody>
</table>

Description

The block CANoM_SDO_READ can be used to read service data objects (SDOs) individually from a slave.

Every time a FALSE → TRUE edge is applied to input EN, CANoM_SDO_READ reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until the processing of the active requests is finished. The completion of the request processing is indicated by DONE = TRUE.

The CANopen object model is defined in the communication profile and device profile specifications (see also system technology of internal couplers - CANopen coupler). Furthermore the device-specific objects are explained in the corresponding device description of the slave.

Some SDOs must be available in every CANopen device by default. Normally, these standard objects are not accessed by the user program during running operation, even though this is possible in principle. The accesses are usually restricted to optional additional objects or their values. These additional slave attributes can be displayed in 907 FB 1131 by selecting Device configuration | Object configuration (see 907 FB 1131 documentation). This view displays a list of predefined objects of the relevant slave, which were retrieved from the EDS file as well as an overview of those standard objects, which were automatically written by the master during the start-up sequence of the node. Each entry contains the current object index, the sub-index, the description of the parameter, its standard value or set value, and the permissible modes of access (read/write).
When using a function block of the CANoM_SDO_READ type, this information can be used to select the desired OBJ_IDX and SUB_IDX.

EN BOOL
If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated. As the block execution requires a bus access, the data are available in the next cycle at the earliest after activating the block.

CONO BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R018X CONO = 1
R01X8 CONO = 2

NODE BYTE
At input NODE the node address of the slave is specified the object value of which is to be requested.

OBJ_IDX WORD
At input OBJ_IDX, the object index of the object to be read in the slave object directory is specified (compare to Obj. Idx. entry in Object configuration list in 907 FB 1131).

SUB_IDX WORD
At input SUB_IDX, the sub-index of the object to be read in the slave object directory is specified (compare to Sub. Idx. entry in Object configuration list in 907 FB 1131).

DATA DWORD
The address of the variable to which the received object data are to be written is specified at input DATA via the ADR operator. According to the object format applied at input DATA, a variable of any data format can be specified via the ADR operator. It is absolutely necessary that the size of the variables (e.g. ARRAY) is sufficient to hold the amount of the read data. The received data are only valid if DONE = TRUE, ERR = 0 and DATA_LEN > 0.

DONE BOOL
The output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing of the task is completed, DONE is set to TRUE for one cycle period.

The values applied at the block outputs are only valid if DONE = TRUE.

ERR WORD
At the output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing of the job. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

If the slave answered the SDO message with an abort message, this is displayed by ERR = 2403 hex (9219 dec). In many cases this is the result of an invalid object index or sub-index or of missing access rights. In this case, the error code transmitted by the slave is additionally output at SDO_ERR.

SDO_ERR DWORD
If the slave answered the SDO message with an abort message, the transmitted error code (refer to CANopen communication profile specification) is output at SDO_ERR. The value applied at SDO_ERR is only valid if DONE = TRUE and ERR = 2403 hex (9219 dec).

DATA_LEN BYTE
The length of the received object data (byte value) is output at DATA_LEN after the procedure has been completed successfully. As the block execution requires a bus access, the data are available in the next cycle at the earliest after activating the block. The value of this input is only valid for DONE = TRUE and ERR = 0.
Function call in IL

LD   SDO_READ_DATA
ADR
ST   ADR_SDO_READ_DATA
CAL   SDO_READ
(EN  := SDO_READ_EN,
CONO := SDO_READ_CONO,
NODE := SDO_READ_NODE,
OBJ_IDX := SDO_READ_OBJ_IDX,
SUB_IDX := SDO_READ_SUB_IDX,
DATA := ADR(SDO_READ_DATA))
LD   SDO_READ_DONE
LD   SDO_READ.ERR
ST   SDO_READ_ERR
LD   SDO_READ.SDO_ERR
ST   SDO_READ_SDO_ERR
LD   SDO_READ.DATA_LEN
ST   SDO_READ_DATA_LEN

Note:
In IL, the function call has to be performed in one line.

Function call in ST

SDO_READ
(EN := SDO_READ_EN,
CONO := SDO_READ_CONO,
NODE := SDO_READ_NODE,
OBJ_IDX := SDO_READ_OBJ_IDX,
SUB_IDX := SDO_READ_SUB_IDX,
DATA := ADR(SDO_READ_DATA));
SDO_READ_DONE := SDO_READ_DONE;
SDO_READ_ERR := SDO_READ.ERR;
SDO_READ_SDO_ERR := SDO_READ.SDO_ERR;
SDO_READ_DATA_LEN := SDO_READ.DATA_LEN;
WRITING AN OBJECT TO A SLAVE

The block CANoM_SDO_WRITE can be used to write service data objects (SDOs) individually to a slave.

Block data
Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>CANoM_SDO_WRITE</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>NODE</td>
<td>BYTE</td>
<td>Node address of the relevant slave</td>
</tr>
<tr>
<td>OBJ_IDX</td>
<td>WORD</td>
<td>Object index of the object to be written</td>
</tr>
<tr>
<td>SUB_IDX</td>
<td>BYTE</td>
<td>Sub-index of the object to be written</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Address from which on the data to be written are stored (via ADR operator)</td>
</tr>
<tr>
<td>DATA_LEN</td>
<td>BYTE</td>
<td>Length of data to be written (byte value)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>SDO_ERR</td>
<td>DWORD</td>
<td>SDO abort error message of the slave</td>
</tr>
</tbody>
</table>

Description

The block CANoM_SDO_WRITE can be used to write service data objects (SDOs) individually to a slave.

Every time a FALSE $\rightarrow$ TRUE edge is applied to input EN, CANoM_SDO_WRITE reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE $\rightarrow$ TRUE edges at input EN are ignored until the processing of the active requests is finished. The completion of the request processing is indicated by DONE $=$ TRUE.

The CANopen object model is defined in the communication profile and device profile specifications (see also system technology of internal couplers - CANopen coupler). Furthermore the device-specific objects are explained in the corresponding device description of the slave.

Some SDOs must be available in every CANopen device by default. Normally, these standard objects are not accessed by the user program during running operation, even though this is possible in principle. The accesses are usually restricted to optional additional objects or their values. These additional slave attributes can be displayed in 907 FB 1131 by selecting Device configuration | Object configuration (see 907 FB 1131 documentation). This view displays a list of predefined objects of the relevant slave, which were retrieved from the EDS file as well as an overview of those standard objects, which were automatically written by the master during the start-up sequence of the node. Each entry contains the current object index, the sub-index, the description of the parameter, its standard value or set value, and the permissible modes of access (read/write).

When using a function block of the CANoM_SDO_WRITE type, this information can be used to select the desired OBJ_IDX and SUB_IDX.

The CANopen object model is defined in the communication profile and device profile specifications (see also system technology of internal couplers - CANopen coupler). Furthermore the device-specific objects are explained in the corresponding device description of the slave.

Some SDOs must be available in every CANopen device by default. Normally, these standard objects are not accessed by the user program during running operation, even though this is possible in principle. The accesses are usually restricted to optional additional objects or their values. These additional slave attributes can be displayed in 907 FB 1131 by selecting Device configuration | Object configuration (see 907 FB 1131 documentation). This view displays a list of predefined objects of the relevant slave, which were retrieved from the EDS file as well as an overview of those standard objects, which were automatically written by the master during the start-up sequence of the node. Each entry contains the current object index, the sub-index, the description of the parameter, its standard value or set value, and the permissible modes of access (read/write). When using a function block of the CANoM_SDO_WRITE type, this information can be used to select the desired OBJ_IDX and SUB_IDX.
EN BOOL
If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

CONO BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R018X CONO = 1
R01X8 CONO = 2

NODE BYTE
At input NODE the node address of the slave is specified to which an object value is to be written.

OBJ_IDX WORD
At input OBJ_IDX, the object index of the object to be written to the slave object directory is specified (compare to Obj. Idx. entry in Object configuration list in 907 FB 1131).

SUB_IDX BYTE
At input SUB_IDX, the sub-index of the object to be written to the slave object directory is specified (compare to Sub. Idx. entry in Object configuration list in 907 FB 1131).

DATA DWORD
The address of the variable, which contains the object data to be written, is specified at the input DATA via the ADR operator. According to the object format applied at input DATA, a variable of any data format can be specified via the ADR operator.

DATA_LEN BYTE
At input DATA_LEN, the length of the data to be transmitted stored in the variable at address DATA is specified as a byte value. This value must match the size of the object to be written. The maximum data length is 247 bytes.

DONE BOOL
The output DONE reflects the state of the request processing. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing of the task is completed, DONE is set to TRUE for one cycle period.

The values applied at the block outputs are only valid if DONE = TRUE.

ERR WORD
At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing of the job. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

If the slave answered the SDO message with an abort message, this is displayed by ERR = 2403 hex (9219 dec). In many cases this is the result of in an invalid object index or sub-index or of missing access rights. In this case, the error code transmitted by the slave is additionally output at SDO_ERR.

SDO_ERR DWORD
If the slave answered the SDO message with an abort message, the transmitted error code (refer to CANopen communication profile specification) is output at SDO_ERR. The value applied at SDO_ERR is only valid if DONE = TRUE and ERR = 2403 hex (9219 dec).
Function call in IL

```
LD   SDO_WRITE_DATA
ADR
ST   ADR_SDO_WRITE_DATA
CAL  SDO_WRITE
(EN   := SDO_WRITE_EN,
CONO := SDO_WRITE_CONO,
NODE := SDO_WRITE_NODE,
OBJ_IDX := SDO_WRITE_OBJ_IDX,
SUB_IDX := SDO_WRITE_SUB_IDX,
DATA := ADR_SDO_WRITE_DATA,
DATA_LEN := SDO_WRITE_DATA_LEN)
LD   SDO_WRITE.DONE
ST   SDO_WRITE_DONE
LD   SDO_WRITE.ERR
ST   SDO_WRITE.ERR
LD   SDO_WRITE.SDO_ERR
ST   SDO_WRITE.SDO_ERR
```

**Note:**
In IL, the function call has to be performed in one line.

Function call in ST

```
SDO_WRITE
(EN   := SDO_WRITE_EN,
CONO := SDO_WRITE_CONO,
NODE := SDO_WRITE_NODE,
OBJ_IDX := SDO_WRITE_OBJ_IDX,
SUB_IDX := SDO_WRITE_SUB_IDX,
DATA := ADR(SDO_WRITE_DATA),
DATA_LEN := SDO_WRITE_DATA_LEN);
SDO_WRITE_DONE :=
SDO_WRITE_DONE;
SDO_WRITE_ERR :=
SDO_WRITE.ERR;
SDO_WRITE_SDO_ERR :=
SDO_WRITE.SDO_ERR;
```
CANoM_NMT can be used to control the operating condition(s) of one specific or all slaves.

### Block data
- Available as of PLC runtime system: V5.0
- Included in library: CANopen_S90_V50.LIB

### Block type
Function block with historical values

### Parameters
- **Instance**: CANoM_NMT
- **EN**: BOOL
- **CONO**: BYTE
- **NODE**: BYTE
- **NMT_CMD**: BYTE
- **DONE**: BOOL
- **ERR**: WORD

### Description
CANoM_NMT can be used to control the operating condition(s) of one specific or all slaves.

Every time a FALSE → TRUE edge is applied to input EN, CANoM_NMT reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until the processing of the active requests is finished. The completion of the request processing is indicated by DONE = TRUE.

Normally, the control of the slave operating states is performed by the automatic control of the CANopen coupler used as NMT master. However, for special applications it can be required to 'manually' change the state of a specific slave. This functionality can be achieved using CANoM_NMT.

- **EN**: BOOL
  - If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.
  - If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.
  - While the request is processed, state changes at input EN are recognized but not evaluated.

- **CONO**: BYTE
  - At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.
    - R018X CONO = 1
    - R01X8 CONO = 2
CONTROLLING NMT NODE STATES VIA NETWORK MANAGEMENT

NODE BYTE
At input NODE the node address of the slave is specified. The operating condition of which shall be changed. By applying a value of 1 to 127 to input NODE, one specific slave can be called with the corresponding node address. If NODE = 0 the command is sent to all slaves.

NMT_CMD BYTE
At input NMT_CMD, the NMT command to be sent is specified. The following NMT commands are defined:

<table>
<thead>
<tr>
<th>NMT command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start remote node (slave)</td>
</tr>
<tr>
<td>2</td>
<td>Stop remote node (slave)</td>
</tr>
<tr>
<td>128</td>
<td>Enter Pre-Operational: Set slave in pre-operational mode</td>
</tr>
<tr>
<td>129</td>
<td>Reset node (slave)</td>
</tr>
<tr>
<td>130</td>
<td>Reset communication</td>
</tr>
</tbody>
</table>

DONE BOOL
The output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If the processing of the task is completed, DONE is set to TRUE for one cycle period.

ERR WORD
At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing of the job. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

Function call in IL

```il
CAL NMT
(EN := NMT_EN,
CONO := NMT_CONO,
NODE := NMT_NODE,
NMT_CMD := NMT_NMT_CMD)
LD SDO_READ_DONE
ST SDO_READ.Done
LD SDO_READ.ERR
ST SDO_READ.ERR
```

Note:
In IL, the function call has to be performed in one line.

Function call in ST

```st
NMT
(EN := NMT_EN,
CONO := NMT_CONO,
NODE := NMT_NODE,
NMT_CMD := NMT_NMT_CMD);
NMT_DONE := NMT_DONE;
NMT_ERR := NMT_ERR;
```
Using the function IDENT_CAN2A_TO_WORD the individual elements of a general CAN telegram header can be put together in the WORD format as it is used by the CAN blocks or the message types.

### Block data
- **Available as of PLC runtime system:** V5.0
- **Included in library:** CANopen_S90_V50.LIB

### Block type
- **Function:**

### Parameters
- **IDENT_ICTW** (*CAN2A_IDENT_TYPE*)
  - Individual components of the general CAN telegram header, put together in a structure
- **WORD_ICTW** (*WORD*)
  - CAN telegram header in WORD format

### Description
Using the function IDENT_CAN2A_TO_WORD the individual elements of a CAN telegram header in the general format, consisting of the 11 bit identifier, the RTR bit and the data length code (DLC), can be easily put together in the WORD format as it is used by the CAN blocks or the message types.

Prior to calling the function, first the desired values have to be assigned to the individual components of the structured input variables. This function has no enable input, it is executed at any time. The output value of this function is structured as follows:

<table>
<thead>
<tr>
<th>DLC.3</th>
<th>DLC.2</th>
<th>DLC.1</th>
<th>DLC.0</th>
<th>RTR</th>
<th>ID.10</th>
<th>ID.9</th>
<th>ID.8</th>
<th>ID.7</th>
<th>ID.6</th>
<th>ID.5</th>
<th>ID.4</th>
<th>ID.3</th>
<th>ID.2</th>
<th>ID.1</th>
<th>ID.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The inverse functionality of IDENT_CAN2A_TO_WORD is provided by the function WORD_TO_IDENT_CAN2A.

### IDENT CAN2A_IDENT_TYPE
At input IDENT a variable of the data type CAN2A_IDENT_TYPE has to be specified. This structure is a summary of the individual components of the general CAN telegram header.

### IDENT_CAN2A_TO_WORD WORD
This function returns the composite general CAN telegram header in the WORD format.

#### Function call in IL
```
IDENT_CAN2A_TO_WORD IDENT_ICTW
ST WORD_ICTW
```

#### Function call in ST
```
WORD_ICTW := IDENT_CAN2A_TO_WORD(IDENT_ICTW);
```
Using the function \texttt{WORD\_TO\_IDENT\_CAN2A}, a compact general CAN telegram header in WORD format (e.g. as supplied by CAN\_REC\_2A) can be split into its individual components for further processing.

\begin{center}
\begin{tabular}{ll}
\textbf{Block data} & \\
Available as of PLC runtime system: & V5.0 \\
Included in library: & CANopen\_S90\_V50.LIB \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ll}
\textbf{Block type} & \\
Function & \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ll}
\textbf{Parameters} & \\
\texttt{WORD\_WTIC} & \texttt{WORD} \\
\texttt{IDENT\_WTIC} & \texttt{CAN2A\_IDENT\_TYPE} \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{lllllllllllllllllllllllllllllll}
\textbf{DLC.3} & \textbf{DLC.2} & \textbf{DLC.1} & \textbf{DLC.0} & \textbf{RTR} & \textbf{ID.10} & \textbf{ID.9} & \textbf{ID.8} & \textbf{ID.7} & \textbf{ID.6} & \textbf{ID.5} & \textbf{ID.4} & \textbf{ID.3} & \textbf{ID.2} & \textbf{ID.1} & \textbf{ID.0} \\
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\end{tabular}
\end{center}

The inverse functionality of \texttt{WORD\_TO\_IDENT\_CAN2A} is provided by the function \texttt{IDENT\_CAN2A\_TO\_WORD}.

\begin{center}
\begin{tabular}{ll}
\textbf{IDENT} & \textbf{WORD} \\
At input IDENT a variable of the type \texttt{WORD} has to be applied which is structured as described above and e.g. supplied by CAN\_REC\_2A. \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ll}
\textbf{IDENT\_CAN2A\_TO\_WORD} & \textbf{CAN2A\_IDENT\_TYPE} \\
This function returns the individual components of the general CAN telegram header, put together in the data type \texttt{CAN2A\_IDENT\_TYPE}. The values of the individual structural elements can then be used for further processing. \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ll}
\textbf{Function call in IL} & \\
\texttt{WORD\_TO\_IDENT\_CAN2A} & \texttt{WORD\_WTIC} \\
\texttt{ST} & \texttt{IDENT\_WTIC} \\
\texttt{Function call in ST} & \\
\texttt{IDENT\_WTIC} := & \\
\texttt{WORD\_TO\_IDENT\_CAN2A(WORD\_WTIC);} \\
\end{tabular}
\end{center}
Using the function IDENT_CANopen_TO_WORD the individual elements of a CANopen-specific CAN telegram header can be put together in the WORD format as it is used by the CAN blocks or the message types.

**Block data**

Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>IDENT_ICTW</th>
<th>CANopen_IDENT_TYPE</th>
<th>Individual components of the CANopen-specific CAN telegram header, put together in a structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD_ICTW</td>
<td>WORD</td>
<td>CAN telegram header in WORD format</td>
</tr>
</tbody>
</table>

**Description**

Using the function IDENT_CANopen_TO_WORD the individual elements of a CAN telegram header in the CANopen-specific format, consisting of the function code, the node ID, the RTR bit and the data length code (DLC), can be easily put together in the WORD format as it is used by the CAN blocks or the message types. Prior to calling the function, first the desired values have to be assigned to the individual components of the structured input variables. This function has no enable input, it is executed at any time. The output value of this function is structured as follows. The 11 bit identifier is composed of the node ID and the function code.

```
<table>
<thead>
<tr>
<th>DLC.3</th>
<th>DLC.2</th>
<th>DLC.1</th>
<th>DLC.0</th>
<th>RTR</th>
<th>FCT.3</th>
<th>FCT.2</th>
<th>FCT.1</th>
<th>FCT.0</th>
<th>ID.6</th>
<th>ID.5</th>
<th>ID.4</th>
<th>ID.3</th>
<th>ID.2</th>
<th>ID.1</th>
<th>ID.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

The inverse functionality of IDENT_CANopen_TO_WORD is provided by the function WORD_TO_IDENT_CANopen.

**Note:**

When generating the header for a CANopen-specific CAN telegram (refer to data type CANopen_MESSAGE_TYPE) it has to be observed that CANopen already uses 4 bytes of user data. For an additional transfer of 4 bytes 'real' user data, a data length code (DLC) of 8 has to be specified.

**IDENT_CANopen_IDENT_TYPE**

At input IDENT a variable of the data type CANopen_IDENT_TYPE has to be specified. This structure is a summary of the individual components of the CANopen-specific CAN telegram header.

**IDENT_CANopen_TO_WORD**

This function returns the composite CANopen-specific CAN telegram header in the WORD format.

**Function call in IL**

```
IDENT_CANopen_TO_WORD IDENT_ICTW
ST WORD_ICTW
```

**Function call in ST**

```
WORD_ICTW :=
IDENT_CANopen_TO_WORD(IDENT_ICTW);
```
Using the function WORD_TO_IDENT_CANopen, a compact CANopen-specific CAN telegram header in WORD format (e.g. as supplied by CAN_REC_2A) can be split into its individual components for further processing.

**Block data**

Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

**Block type**

Function

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD_WTIC</td>
<td>Compact CAN telegram header in WORD format</td>
</tr>
<tr>
<td>IDENT_WTIC</td>
<td>Individual components of the general CAN telegram header, put together in a structure</td>
</tr>
</tbody>
</table>

**Description**

Using the function WORD_TO_IDENT_CANopen a compact CANopen-specific CAN telegram header in WORD format (as used by the CAN blocks or the message types) can be easily split into its individual components function code, node ID, RTR bit and data length code (DLC).

This function has no enable input, it is executed at any time. The input value of this function is structured as follows.

The 11 bit identifier is composed of the node ID and the function code.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>DLC.3</td>
</tr>
<tr>
<td>14</td>
<td>DLC.2</td>
</tr>
<tr>
<td>13</td>
<td>DLC.1</td>
</tr>
<tr>
<td>12</td>
<td>DLC.0</td>
</tr>
<tr>
<td>11</td>
<td>RTR</td>
</tr>
<tr>
<td>10</td>
<td>FCT.3</td>
</tr>
<tr>
<td>9</td>
<td>FCT.2</td>
</tr>
<tr>
<td>8</td>
<td>FCT.1</td>
</tr>
<tr>
<td>7</td>
<td>FCT.0</td>
</tr>
<tr>
<td>6</td>
<td>ID.6</td>
</tr>
<tr>
<td>5</td>
<td>ID.5</td>
</tr>
<tr>
<td>4</td>
<td>ID.4</td>
</tr>
<tr>
<td>3</td>
<td>ID.3</td>
</tr>
<tr>
<td>2</td>
<td>ID.2</td>
</tr>
<tr>
<td>1</td>
<td>ID.1</td>
</tr>
<tr>
<td>0</td>
<td>ID.0</td>
</tr>
</tbody>
</table>

The inverse functionality of WORD_TO_IDENT_CANopen is provided by the function IDENT_CANopen_TO_WORD.

**Note:**

When splitting the header of a CANopen-specific CAN telegram (refer to data type CANopen_MESSAGE_TYPE) it has to be observed that CANopen already uses 4 bytes of user data. If the function outputs a data length code (DLC) of 8, the corresponding telegram contains 4 bytes of ‘real’ user data.

**IDENT**

At input IDENT a variable of the type WORD has to be applied which is structured as described above and e.g. supplied by CAN_REC_2A.

**IDENT_CANopen_TO_WORD**

This function returns the individual components of the CANopen-specific CAN telegram header, put together in the data type CANopen_IDENT_TYPE. The values of the individual structural elements can then be used for further processing.

**Function call in IL**

```
WORD_TO_IDENT_CANopen WORD_WTIC
ST IDENT_WTIC
```

**Function call in ST**

```
IDENT_WTIC := WORD_TO_IDENT_CANopen(WORD_WTIC);
```
RECEIVING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 A

Using CAN_REC_2A any CAN telegrams with 11 bit identifiers according to CAN 2.0 A can be received.

Block data
Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB
Remark:

Block type
Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>CAN_REC_2A</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>MSGS</td>
<td>DWORD</td>
<td>Address from which on the received telegrams are stored (via ADR operator)</td>
</tr>
</tbody>
</table>

| DONE         | BOOL       | Ready message of the block |
| ERR          | WORD       | Error message |
| NUM_MSGS     | BYTE       | Number of received telegrams |

Description
Using CAN_REC_2A, CAN telegrams with 11 bit identifiers according to CAN 2.0 A can be received. Only those telegrams can be received, the identifiers of which have been previously enabled. In contrast to telegrams with 29 bit identifiers which are enabled for reception via 907 FB 1131 (refer to 907 FB 1131 documentation), the reception of telegrams with 11 bit identifiers is enabled via function block (refer to CAN_REC_FILTER_2A). Received telegrams with non-enabled identifiers are automatically dismissed by the coupler.

Every time a FALSE → TRUE edge is applied to input EN, CAN_REC_2A reads the data at its inputs and then waits for a receipt confirmation by the coupler. As long as EN = TRUE, the block is active and waits for the reception of telegrams. The reception of one or several telegrams is displayed via DONE = TRUE and the number of received telegrams is displayed at NUM_MSGS.

Caution:
If 11 bit identifiers are enabled for reception, the program must contain a continuously active CAN_REC_2A block. Otherwise a communication channel overload can occur. Other blocks which are also using this communication channel can possibly be temporarily not executed or only executed in longer time intervals.

If the block is active, it outputs all telegrams received since the last call up to a total length of 254 bytes. According to this, the maximum number of simultaneously output telegrams depends on the sum of the individual telegram lengths. If all received telegrams do not contain any other data than the 2 header bytes (identifier, RTR and DLC; DLC = 0), up to 127 telegrams can be simultaneously output (2 x 127 = 254).

However, if all received telegrams contain the maximum 8 bytes of data, only up to 25 telegrams can be simultaneously output ((2 + 8) x 25 = 250). If more telegrams are received between two calls of the block than the block can simultaneously output, the remaining telegrams are output on the next call of the block. For a high throughput rate of receive telegrams, several CAN_REC_2A blocks can be used.
RECEIVING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 A

EN
If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the values at the inputs are valid, the block is waiting for the reception of CAN 2.0 A telegrams with 11 bit identifiers. DONE is set to FALSE as long as no telegrams are received. If at least one input value is invalid, an error is indicated at output ERR. Additionally the termination of the request processing is indicated by DONE = TRUE.

CONO
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.
R018X CONO = 1
R01X8 CONO = 2

MSGS
The address of the variable to which the received telegrams are to be written is specified at the input MSGS via the ADR operator.
The block evaluates the data length codes (DLC) of the individual received telegrams and structures the telegrams according to the data type CAN2A_MESSAGE_TYPE or CANopen_MESSAGE_TYPE defined in the library.
Because the block is able to output up to 127 telegrams at the same time, the target variable for storing the received telegrams has to be ARRAY [1..127] OF CAN2A_MESSAGE_TYPE (or CANopen_MESSAGE_TYPE). Then, the first element of the ARRAY or directly the ARRAY (without an index) has to be specified at the input of the ADR operator.
The received telegrams stored in the variable are only valid if DONE = TRUE, ERR = 0 and NUM_MSGS <> 0.

DONE
If the block is active (EN = TRUE), output DONE reflects the processing state. In case of a faulty block input, DONE is continuously set to TRUE and the error is displayed at ERR. If error-free, and at least one telegram was received in the current cycle, DONE is set to TRUE for one cycle. Then, the received telegrams are stored in MSGS and their number is displayed in NUM_MSGS.

ERR
At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to a block input or if an error occurred during the processing of the reception. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

NUM_MSGS
NUM_MSGS outputs the number of valid telegrams received during the current cycle and stored in MSGS. Thus, NUM_MSGS indicates the ARRAY index of MSGS up to which the receive telegrams can be evaluated. NUM_MSGS has always to be considered together with the outputs DONE and ERR. The value output at NUM_MSGS is only valid if DONE = TRUE and ERR = 0.
Function call in IL

```
LD REC_2A_MSGS[1]
ADR
ST ADR_REC_2A_MSGS
CAL REC_2A
(EN := REC_2A_EN,
CONO := REC_2A_CONO,
MSGS := ADR_REC_2A_MSGS)
LD REC_2A_DONE
ST REC_2A_DONE
LD REC_2A_ERR
ST REC_2A_ERR
LD REC_2A_NUM_MSGS
ST REC_2A_NUM_MSGS
```

Note:
In IL, the function call has to be performed in one line.

Function call in ST

```
REC_2A
(EN := REC_2A_EN,
CONO := REC_2A_CONO,
MSGS := ADR(REC_2A_MSGS[1]));
REC_2A_DONE := REC_2A_DONE;
REC_2A_ERR := REC_2A_ERR;
REC_2A_NUM_MSGS := REC_2A_NUM_MSGS;
```
ENABLING OF IDENTIFIERS FOR RECEIVING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS ACCORDING TO CAN 2.0 A CAN_REC_FILTER_2A

Using CAN_REC_FILTER_2A
any 11 bit identifiers according to CAN 2.0 A can be enabled for reception.

Block data
Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB
Remark:

Block type
Function block with historical values

Parameters
<table>
<thead>
<tr>
<th>Instance</th>
<th>CAN_REC_FILTER</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>NUM_IDENTS</td>
<td>BYTE</td>
<td>Number of identifiers for which the reception is to be enabled</td>
</tr>
<tr>
<td>IDENTS</td>
<td>DWORD</td>
<td>Address from which on the list of identifiers to be enabled is stored (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

Description
Using CAN_REC_FILTER_2A, 11 bit identifiers according to CAN 2.0 A can be enabled for reception. The actual reception of the telegrams is performed via CAN_REC_2A. Received telegrams containing a not enabled identifier are automatically dismissed by the coupler. Telegrams containing identifiers which were assigned to the integrated CANopen protocol during configuration are as well automatically processed by the coupler and not forwarded to the receiving block, no matter if the concerning identifiers are enabled for reception or not.

Note:
If the PLC program is stopped, all previously set receive filters are automatically reset.
CAN_REC_FILTER_2A has to be executed on each program start to re-enable the reception of the desired identifiers.

Caution:
If 11 bit identifiers are enabled for reception, the program must contain a continuously active CAN_REC_2A block. Otherwise a communication channel overload can occur. Other blocks which are also using this communication channel can possibly be temporarily not executed or only executed in longer time intervals.

Every time a FALSE \( \rightarrow \) TRUE edge is applied to input EN, CAN_REC_FILTER_2A reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE \( \rightarrow \) TRUE edges at input EN are ignored until the processing of the active requests is finished. The completion of the request processing is indicated by DONE = TRUE. A possibly occurring error is indicated at output ERR.

The maximum number of filter identifiers which can be simultaneously set within one operation is limited to 127. A total of 2048 11 bit identifiers (and thus all possible identifiers in the range between 0 and 2047) can be enabled for reception by repeatedly calling one block or by using several blocks.
ENABLING OF IDENTIFIERS FOR RECEIVING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS ACCORDING TO CAN 2.0 A CAN_REC_FILTER_2A

EN  BOOL
If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

CONO  BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.
R018X  CONO = 1
R01X8  CONO = 2

NUM_IDENTS  BYTE
Input NUM_IDENTS is used to specify the number of valid identifiers to be enabled for reception which are stored from the address IDENTS on. Valid values for NUM_IDENTS are 1 to 127.

IDENTS  DWORD
At input IDENTS, the address of the variable which contains the list of identifiers to be enabled for reception is specified via the ADR operator.

The chosen variable should be an ARRAY [1..X] OF WORD, where the value for X must be at least as high as the number specified at input NUM_IDENTS. The individual elements (11 bit identifiers) of this list must either meet the general CAN format:

```
1 5 1 4 1 3 1 2 1 11 09876543210
```

or the CANopen-specific format:

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

(X = not evaluated)

As shown in the figures above, only the 11 bit identifier is relevant for enabling the reception. An additionally specified RTR bit or a DLC (data length code) are not evaluated. This guarantees that all telegrams with an identifier enabled for reception are forwarded to the receiving block CAN_REC2A, independent of the RTR bit and the telegram length. This also allows to use the functions IDENT_CAN2A_TO_WORD or IDENT_CANopen_TO_WORD for generating the individual identifiers.

DONE  BOOL
The output DONE reflects the state of the request processing. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If processing is finished, DONE is set to TRUE. The identifier receive filters have only been read correctly if ERR is also 0. If ERR is not 0, an error occurred.

ERR  WORD
At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to a block input or if an error occurred during processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.
Function call in IL

LD   REC_FILTER_2A_IDENTS
ADR  ADR_REC_FILTER_2A_IDENTS
ST   ADR_REC_FILTER_2A_IDENTS
CAL  REC_FILTER_2A
(EN := REC_FILTER_2A_EN,
CONO := REC_FILTER_2A_CONO,
NUM_IDENTS := REC_FILTER_2A_NUM_IDENTS,
IDENTS := ADR_REC_FILTER_2A_IDENTS)
LD   REC_FILTER_2A_DONE
ST   REC_FILTER_2A_DONE
LD   REC_FILTER_2A_ERR
ST   REC_FILTER_2A_ERR

Note:
In IL, the function call has to be performed in one line.

Function call in ST

REC_FILTER_2A
(EN := REC_FILTER_2A_EN,
CONO := REC_FILTER_2A_CONO,
NUM_IDENTS := REC_FILTER_2A_NUM_IDENTS,
IDENTS := ADR_REC_FILTER_2A_IDENTS);
REC_FILTER_2A_DONE :=
REC_FILTER_2A_DONE;
REC_FILTER_2A_ERR :=
REC_FILTER_2A_ERR;
Using CAN_SEND_2A any CAN telegrams with 11 bit identifiers according to CAN 2.0 A can be transmitted.

**Block data**

Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

**Remark:**

Included in library: CANopen_S90_V50.LIB

**Block type**

Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>CAN_SEND_2A</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>NUM_MSGS</td>
<td>BYTE</td>
<td>Number of telegrams to be transmitted</td>
</tr>
<tr>
<td>MSGS</td>
<td>DWORD</td>
<td>Address from which on the telegrams to be transmitted are stored (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

**Description**

Using CAN_SEND_2A, CAN telegrams with 11 bit identifiers according to CAN 2.0 A can be transmitted.

Every time a FALSE → TRUE edge is applied to input EN, CAN_SEND_2A reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE → TRUE edges at input EN are ignored until processing of the active requests is finished. The completion of the request processing is indicated by DONE = TRUE. A possibly occurring error is indicated at output ERR.

The block is able to simultaneously transmit several telegrams up to a total length of 254 bytes within one event to the coupler which in turn transmits these telegrams sequentially via the bus. According to this, the maximum number of simultaneously transmitted telegrams depends on the sum of the individual telegram lengths. If all telegrams to be transmitted do not contain any other data than the 2 header bytes (identifier, RTR and DLC; DLC = 0), up to 127 telegrams can be simultaneously transmitted (2 x 127 = 254). However, if all telegrams contain the maximum 8 bytes of data, only up to 25 telegrams can be simultaneously transmitted to the coupler ((2 + 8) x 25 = 250).

EN BOOL

If a FALSE → TRUE edge is applied to input EN, all further inputs are read in.

If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.
TRANSMITTING CAN TELEGRAMS WITH 11 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 A

CONO BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.
R018X CONO = 1
R01X8 CONO = 2

NUM_MSGS BYTE
At input NUM_MSGS the number of valid telegrams to be transmitted and stored in MSGS is specified. Basically, the valid values for NUM_MSGS are 1 to 127. However, the upper limit depends on the total length of all telegrams. The total length is calculated by the block from the data length codes (DLC) of the individual telegrams and may not exceed 254 bytes. Otherwise an error message is generated. In such cases the number of telegrams to be transmitted has to be chosen correspondingly that the total length does not exceed 254.

MSGS DWORD
At input MSGS the address of the variable which contains the telegrams to be transmitted is specified via the ADR operator. The telegrams to be transmitted have to be of the data type CAN2A_MESSAGE_TYPE or CANopen_MESSAGE_TYPE defined in the library. If only one telegram has to be transmitted (NUM_MSGS = 1), a single variable is sufficient. If more than one telegram have to be transmitted, the chosen variable has to be a ARRAY [1..X] CAN2A_MESSAGE_TYPE or ARRAY [1..X] OF CANopen_MESSAGE_TYPE, where the value of X must be at least as high as the number specified at input NUM_MSGS. Then, the first element of the ARRAY or directly the ARRAY (without an index) has to be specified at the input of the ADR operator.

The block evaluates only the data length codes (DLC) of the individual telegrams to determine the total length over all telegrams and does not perform any further interpretations of the telegrams. Because the position within the telegram header as well as the encoding of the data length code are identical for all CAN telegrams, the telegrams to be transmitted can be alternatively specified in the general format (CAN2A_MESSAGE_TYPE) or in the CANopen format (CANopen_MESSAGE_TYPE). The functions IDENT_CAN2A_TO_WORD and/or IDENT_CANopen_TO_WORD represent a comfortable option to form the required telegram headers for this block or for the corresponding message type in the WORD format.

DONE BOOL
The output DONE reflects the state of the request processing. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If the processing is finished, DONE is set to TRUE. The transmission of the telegrams to the coupler was only performed correctly if ERR is also 0. If ERR is not 0, an error occurred.

ERR WORD
At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to a block input or if an error occurred during processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.
Function call in IL

<table>
<thead>
<tr>
<th>LD</th>
<th>SEND_2A_MSGS[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>ADR_SEND_2A_MSGS</td>
</tr>
<tr>
<td>CAL</td>
<td>SEND_2A</td>
</tr>
<tr>
<td></td>
<td>(EN := SEND_2A_EN,</td>
</tr>
<tr>
<td></td>
<td>CONO := SEND_2A_CONO,</td>
</tr>
<tr>
<td></td>
<td>NUM_MSGS := SEND_2A_NUM_MSGS,</td>
</tr>
<tr>
<td></td>
<td>MSGS := ADR_SEND_2A_MSGS)</td>
</tr>
<tr>
<td>LD</td>
<td>SEND_2A_DONE</td>
</tr>
<tr>
<td>ST</td>
<td>SEND_2A_DONE</td>
</tr>
<tr>
<td>LD</td>
<td>SEND_2A_ERR</td>
</tr>
<tr>
<td>ST</td>
<td>SEND_2A_ERR</td>
</tr>
</tbody>
</table>

Note:
In IL, the function call has to be performed in one line.

Function call in ST

```plaintext
SEND_2A
(EN := SEND_2A_EN,
CONO := SEND_2A_CONO,
NUM_MSGS := SEND_2A_NUM_MSGS,
MSGS := ADR(SEND_2A_MSGS[1]));
SEND_2A_DONE := SEND_2A_DONE;
SEND_2A_ERR := SEND_2A_ERR;
```
RECEIVING CAN TELEGRAMS WITH 29 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 B

Using CAN_REC_2B any CAN telegrams with 29 bit identifiers according to CAN 2.0 B can be received.

Block data
Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

Block type
Function block with historical values

Parameters
<table>
<thead>
<tr>
<th>Instance</th>
<th>CAN_REC_2B</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>IDENT</td>
<td>DWORD</td>
<td>29 bit identifier which has to be contained in the telegram to be received</td>
</tr>
<tr>
<td>MSG</td>
<td>DWORD</td>
<td>Address from which on the received telegram is stored (via ADR operator)</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Ready message of the block</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
</tbody>
</table>

Description
Using CAN_REC_2B, CAN telegrams with 29 bit identifiers according to CAN 2.0 B can be received.
Only those telegrams can be received, the identifiers of which have been previously enabled. In contrast to telegrams with 11 bit identifiers which are enabled for reception via function block (refer to function block CAN_REC_FILTER_2A), the reception of telegrams with 29 bit identifiers is enabled via 907 FB 1131 (refer to 907 FB 1131 documentation). Received telegrams with non-enabled identifiers are automatically dismissed by the coupler.

Every time a FALSE → TRUE edge is applied to input EN, CAN_REC_2B reads the data at its inputs and then waits for a receipt confirmation of the coupler. As long as EN = TRUE, the block is active and waits for the reception of telegrams with the concerning identifier. The reception of a telegram is indicated via DONE = TRUE and the telegram is stored in the variable address specified at MSG.

Caution:
If 29 bit identifiers are enabled for reception, the program must contain a continuously active CAN_REC_2B block. Otherwise a communication channel overload can occur. Other blocks which are also using this communication channel can possibly be temporarily not executed or only executed in longer time intervals.

Enabling for reception of 29 bit identifiers can be performed in 907 FB 1131 in different ways (refer to documentation). The 29 bit filter includes an accept code and an accept mask. The coupler compares all 29 bit identifiers at the bus with these two entries. If a bit is set within the accept mask, this means that the corresponding bit of the identifier shall not be compared with the corresponding bit of the accept code, but be accepted independent of its value.
RECEIVING CAN TELEGRAMS WITH 29 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 B

If a bit is not set within the accept mask, the value of the concerning bit in the identifier must match the status of the corresponding bit in the accept code in order to be accepted. This evaluation is explained by the following figure.

**Identifier bit0**
**Accept code bit0**

**Accept mask bit0**

**Identifier bit28**
**Accept code bit28**

**Accept mask bit28**

If the block is active, it outputs each received telegram which contains the identifier specified at IDENT. The block is only able to receive one telegram per cycle. To enable the reception of several telegrams with the same identifier within one cycle, several blocks of the type CAN_REC_2B have to be used correspondingly. To enable the reception of telegrams with different identifiers it is recommended to use one continuously active block of the type CAN_REC_2B for each identifier.

**CONO**
**BYTE**

At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

R018X CONO = 1
R01X8 CONO = 2

**IDENT**
**DWORD**

Input IDENT is used to specify the 29 bit identifier which has to be contained in the received telegram. The RTR bit and the data length code additionally contained in the telegram header are not considered on reception. This way it is guaranteed that all telegrams containing the identifier IDENT are received, independent of their RTR bit state and telegram length.

**MSG**
**DWORD**

At input MSG the address of the variable, to which a received telegram has to be written, is specified via the ADR operator.

The block evaluates the data length code (DLC) of the received telegram and structures the telegram according to the data type CAN2B_MESSAGE_TYPE as defined in the library. Therefore a variable of the type CAN2B_MESSAGE_TYPE has to be chosen as the target variable to store the received telegram.

The received telegram stored in the variable is only valid if DONE = TRUE and ERR = 0.

**DONE**
**BOOL**

If the block is active (EN = TRUE), output DONE reflects the processing state. In case of a faulty block input, DONE is continuously set to TRUE and the error is displayed at ERR. If error-free, and a telegram was received in the current cycle, DONE is set to TRUE for one cycle. The received telegram is then stored in MSG.

**ERR**
**WORD**

At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to a block input or if an error occurred during processing of the reception. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.
RECEIVING CAN TELEGRAMS WITH 29 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 B

Function call in IL

LD REC_2B_MSG
ADR
ST ADR_REC_2B_MSG
CAL REC_2B
(EN := REC_2B_EN,
CONO := REC_2B_CONO,
IDENT := REC_2B_IDENT,
MSG := ADR_REC_2B_MSG)
LD REC_2B.DONE
ST REC_2B_DONE
LD REC_2B.ERR
ST REC_2B_ERR

Note:
In IL, the function call has to be performed in one line.

Function call in ST

REC_2B
(EN := REC_2B_EN,
CONO := REC_2B_CONO,
IDENT := REC_2B_IDENT,
MSG := ADR(REC_2B_MSG));
REC_2B_DONE := REC_2B_DONE;
REC_2B_ERR := REC_2B_ERR;
TRANSMITTING CAN TELEGRAMS WITH 29 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 B

Using CAN_SEND_2B any CAN telegrams with 29 bit identifiers according to CAN 2.0 B can be transmitted.

Block data
Available as of PLC runtime system: V5.0
Included in library: CANopen_S90_V50.LIB

Block type
Function block with historical values

Parameters
- **Instance**: CAN_SEND_2B
- **EN**: BOOL
- **CONO**: BYTE
- **MSG**: DWORD
- **DONE**: BOOL
- **ERR**: WORD

Description
Using CAN_SEND_2B, CAN telegrams with 29 bit identifiers according to CAN 2.0 B can be transmitted.

Every time a FALSE $\rightarrow$ TRUE edge is applied to input EN, CAN_SEND_2B reads the data at its inputs and sends a corresponding request message to the coupler. Further FALSE $\rightarrow$ TRUE edges at input EN are ignored until processing of the active requests is finished. The completion of the request processing is indicated by DONE = TRUE. A possibly occurring error is indicated at output ERR.

**EN**
If a FALSE $\rightarrow$ TRUE edge is applied to input EN, all further inputs are read in.
If the input values are valid, a corresponding request message is sent to the coupler. While this request message is processed, the output DONE is set to FALSE. If at least one input value is invalid, an error is indicated at output ERR. Additionally, the termination of the request processing is indicated by DONE = TRUE.

While the request is processed, state changes at input EN are recognized but not evaluated.

**CONO**
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category:
- R018X  CONO = 1
- R01X8  CONO = 2

**MSG**
At input MSG the address of the variable which contains the telegram to be transmitted is specified via the ADR operator. The telegram to be transmitted or the variable have to be of the data type CAN2B_MESSAGE_TYPE as defined in the library.
TRANSMITTING CAN TELEGRAMS WITH 29 BIT IDENTIFIERS
ACCORDING TO CAN 2.0 B

DONE
The output DONE reflects the state of the request processing. If the block is triggered by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If the processing is finished, DONE is set to TRUE. The transmission of the telegrams to the coupler was only performed correctly if ERR is also 0. If ERR is not 0, an error occurred.

ERR
At output ERR an error identifier is applied for a period of one cycle if an invalid value was applied to a block input or if an error occurred during processing. ERR has always to be considered together with the output DONE. The value applied at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all CANopen blocks and is explained at the beginning of the library description.

BOOL

WORD

Function call in IL
LD SEND_2B_MSG
ADR
ST ADR_SEND_2B_MSG
CAL SEND_2B
(EN := SEND_2B_EN,
CONO := SEND_2B_CONO,
MSG := ADR_SEND_2B_MSG)
LD SEND_2B.DONE
ST SEND_2B_DONE
LD SEND_2B.ERR
ST SEND_2B_ERR

Note:
In IL, the function call has to be performed in one line.

Function call in ST
SEND_2B
(EN := SEND_2B_EN,
CONO := SEND_2B_CONO,
MSG := ADR(SEND_2B_MSG));
SEND_2B_DONE := SEND_2B.DONE;
SEND_2B_ERR := SEND_2B.ERR;
Glossary

BOOL

Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

DINT

DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT:</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

DWORD

DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

INT

INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT:</td>
<td>-32768</td>
<td>32767</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

WORD

WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD:</td>
<td>0</td>
<td>65535</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can be derived:

- Within functions, global variables can neither be read nor written.
- Within functions, absolute operands can neither be read nor written.
- Within functions, function blocks must not be called.

Function blocks

Function blocks are subroutines which can have as many inputs, outputs and internal variables as required. They are called from a program or from another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. Therefore e.g. counters can be realized which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters, a function only one. The output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
THE CANOPEN LIBRARY

Index

C
CAN_REC_2A S90 40
CAN_REC_2B S90 49
CAN_REC_FILTER_2A S90 43
CAN_SEND_2A S90 46
CAN_SEND_2B S90 52
CANoM_NMT S90 34
CANoM_NODEDIAG S90 15
CANoM_RES_ERR S90 19
CANoM_SDO_READ S90 28
CANoM_SDO_WRITE S90 31
CANoM_STAT S90 21
CANoM_SYSDIAG S90 26
CANopen S90
   CAN_REC_2A 40
   CAN_REC_2B 49
   CAN_REC_FILTER_2A 43
   CAN_SEND_2A 46
   CAN_SEND_2B 52
   CANoM_NMT 34
   CANoM_NODEDIAG 15
   CANoM_RES_ERR 19
   CANoM_SDO_READ 28
   CANoM_SDO_WRITE 31
   CANoM_STAT 21
   CANoM_SYSDIAG 26
   CANopen_INFO 13
   IDENT_CAN2A_TO_WORD 36
   IDENT_CANopen_TO_WORD 38
   WORD_TO_IDENT_CAN2A 37
   WORD_TO_IDENT_CANopen 39
CANopen_INFO S90 13

components of the library 3

I
IDENT_CAN2A_TO_WORD S90 36
IDENT_CANopen_TO_WORD S90 38

W
WORD_TO_IDENT_CAN2A S90 37
WORD_TO_IDENT_CANopen S90 39
Software Description

Advant Controller 31
Intelligent Decentralized Automation System

Ethernet
Function Block Library
90 Series
Contents

The Ethernet library 2
SPECIAL CHARACTERISTICS OF THE ETHERNET LIBRARY ........................................ 2
COMPONENTS OF THE ETHERNET LIBRARY .................................................. 2
ERROR MESSAGES OF THE ETHERNET LIBRARY BLOCKS .......................... 3

Overview of blocks arranged according to their call names 6
READING INFORMATION ABOUT THE INSTALLED COUPLERS ETH_INFO .............. 7
INITIALIZING THE ETHERNET UDP/IP DATA EXCHANGE ETH_AINIT ............... 9
READING AN ETHERNET UDP/IP DATA PACKAGE ETH_AREC .......................... 14
FROM THE RECEIVE BUFFER ETH_ASEND ........................................ 17
TRANSMITTING AN ETHERNET UDP/IP DATA PACKAGE ETH_ASTO .............. 20
READING ETHERNET UDP/IP TIMEOUT DATA PACKAGES ETH_MODSTAT .......... 23
READING STATUS INFORMATION ABOUT OPENMODBUS PROCESSING ETH_MODMAST .... 26
PROCESSING OPENMODBUS ON TCP/IP CLIENT TELEGRAMS

Glossary 29

Index 31
The Ethernet library

SPECIAL CHARACTERISTICS OF THE ETHERNET LIBRARY

Note:

Ethernet communication is only performed in RUN mode of the PLC, but not in simulation mode.

Operating the controller as OpenModbus on TCP/IP subscriber can be performed simultaneously with other protocols. When operated in this mode, the Ethernet coupler is able to execute the functionality of several servers or several clients at the same time. Mixed operation is also possible. Configuring the coupler accordingly has to be done by means of 907 FB 1131 (refer to 907 FB 1131 documentation).

In order to operate the controller as OpenModbus on TCP/IP server (slave), only the coupler has to be configured accordingly using 907 FB 1131. An additional use of the OpenModbus on TCP/IP blocks in the user program is not necessary.

To operate the controller as OpenModbus on TCP/IP client (master), the coupler also has to be configured using 907 FB 1131. In this case, one or more ETH_MODMAST blocks have to be planned in the user program additionally.

The ETH_MODMAST block can be optionally operated in server mode as well as in client mode or in mixed operation.

The following ports are reserved:

<table>
<thead>
<tr>
<th>Port</th>
<th>Reserved for</th>
</tr>
</thead>
<tbody>
<tr>
<td>32768</td>
<td>Ethernet UDP/IP data exchange (ETH_Axxx-blocks)</td>
</tr>
<tr>
<td>1200</td>
<td>TCP/IP Gateway access</td>
</tr>
<tr>
<td>502</td>
<td>OpenModbus on TCP/IP</td>
</tr>
</tbody>
</table>

COMPONENTS OF THE ETHERNET LIBRARY

The Ethernet library contains the following function blocks:

Group: General
ETH_INFO - Reading information about installed couplers

Group: UDP_IP
ETH_AINIT - Initializing the Ethernet UDP/IP data exchange
ETH_ASEND - Transmitting a data package to a subscriber via Ethernet UDP/IP
ETH_ARC - Reading a data package from the Ethernet UDP/IP receive buffer
ETH_ASTO - Reading Ethernet UDP/IP timeout data packages from the timeout data buffer

Group: MODBUS_TCP (OpenModbus on TCP/IP)
ETH_MODSTAT - Reading status information of the OpenModbus on TCP/IP processing
ETH_MODMAST - Processing OpenModbus on TCP/IP client (master) telegrams

Detailed information about the various blocks can be found in the following sections.
ERROR MESSAGES OF THE ETHERNET LIBRARY BLOCKS

The following errors are signalized at output ERR of the ETH_xxx blocks:

### Errors at block inputs:

<table>
<thead>
<tr>
<th>DEC</th>
<th>HEX</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8193</td>
<td>2001</td>
<td>Invalid value at block input CONO</td>
</tr>
<tr>
<td>8194</td>
<td>2002</td>
<td>Coupler model in slot CONO does not correspond to the block type</td>
</tr>
<tr>
<td>8195</td>
<td>2003</td>
<td>No Ethernet coupler in slot CONO</td>
</tr>
<tr>
<td>8196</td>
<td>2004</td>
<td>Ethernet coupler in slot CONO is not in OPERATE state</td>
</tr>
<tr>
<td>8202</td>
<td>200A</td>
<td>Invalid data length at block input</td>
</tr>
<tr>
<td>8203</td>
<td>200B</td>
<td>Invalid socket number at block input</td>
</tr>
</tbody>
</table>

### Error messages of the ETH library:

<table>
<thead>
<tr>
<th>DEC</th>
<th>HEX</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8449</td>
<td>2101</td>
<td>Invalid socket ID</td>
</tr>
<tr>
<td>8450</td>
<td>2102</td>
<td>Coupler cannot be accessed. System is waiting for free resources</td>
</tr>
<tr>
<td>8451</td>
<td>2103</td>
<td>Invalid socket number</td>
</tr>
<tr>
<td>8452</td>
<td>2104</td>
<td>Socket is already open and active</td>
</tr>
<tr>
<td>8453</td>
<td>2105</td>
<td>No free socket available</td>
</tr>
<tr>
<td>8454</td>
<td>2106</td>
<td>Invalid socket state</td>
</tr>
<tr>
<td>8455</td>
<td>2107</td>
<td>Invalid command</td>
</tr>
<tr>
<td>8456</td>
<td>2108</td>
<td>Invalid data length</td>
</tr>
<tr>
<td>8457</td>
<td>2109</td>
<td>Error while accessing the data buffer</td>
</tr>
<tr>
<td>8458</td>
<td>210A</td>
<td>The coupler performed a reset during processing. Processing was aborted.</td>
</tr>
<tr>
<td>8480</td>
<td>2120</td>
<td>Data buffer full</td>
</tr>
<tr>
<td>8481</td>
<td>2121</td>
<td>Error while creating the data buffer</td>
</tr>
<tr>
<td>8482</td>
<td>2122</td>
<td>Data could not be read from the buffer or written to the buffer successfully // Accessed buffer is empty or full</td>
</tr>
<tr>
<td>DEC</td>
<td>HEX</td>
<td>Error description</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>9266</td>
<td>2432</td>
<td>TCP/UDP task not available</td>
</tr>
<tr>
<td>9267</td>
<td>2433</td>
<td>Internal task with configuration data not available</td>
</tr>
<tr>
<td>9268</td>
<td>2434</td>
<td>No MAC address available</td>
</tr>
<tr>
<td>9269</td>
<td>2435</td>
<td>Waiting for warm start performed by the application</td>
</tr>
<tr>
<td>9270</td>
<td>2436</td>
<td>Unknown flag in start parameters</td>
</tr>
<tr>
<td>9271</td>
<td>2437</td>
<td>Invalid IP address in start parameters</td>
</tr>
<tr>
<td>9272</td>
<td>2438</td>
<td>Invalid subnet mask in start parameters</td>
</tr>
<tr>
<td>9273</td>
<td>2439</td>
<td>Invalid gateway IP in start parameters</td>
</tr>
<tr>
<td>9275</td>
<td>243B</td>
<td>Unknown device type</td>
</tr>
<tr>
<td>9276</td>
<td>243C</td>
<td>Access to IP address in the specified source failed</td>
</tr>
<tr>
<td>9277</td>
<td>243D</td>
<td>Initialization of the driver layer failed</td>
</tr>
<tr>
<td>9278</td>
<td>243E</td>
<td>No source specified for IP address (BOOTP, DHCP, IP address parameter)</td>
</tr>
<tr>
<td>9326</td>
<td>246E</td>
<td>Timeout has occurred</td>
</tr>
<tr>
<td>9327</td>
<td>246F</td>
<td>Invalid timeout parameter</td>
</tr>
<tr>
<td>9328</td>
<td>2470</td>
<td>Invalid socket</td>
</tr>
<tr>
<td>9329</td>
<td>2471</td>
<td>Command cannot be executed. Socket not ready</td>
</tr>
<tr>
<td>9331</td>
<td>2473</td>
<td>No access to target IP address</td>
</tr>
<tr>
<td>9332</td>
<td>2474</td>
<td>Invalid option parameter</td>
</tr>
<tr>
<td>9333</td>
<td>2475</td>
<td>Invalid command parameter</td>
</tr>
<tr>
<td>9334</td>
<td>2476</td>
<td>Invalid IP address or no access to address</td>
</tr>
<tr>
<td>9335</td>
<td>2477</td>
<td>Invalid port number or port not available</td>
</tr>
<tr>
<td>9336</td>
<td>2478</td>
<td>Connection closed</td>
</tr>
<tr>
<td>9337</td>
<td>2479</td>
<td>Connection reset</td>
</tr>
<tr>
<td>9338</td>
<td>247A</td>
<td>Invalid protocol parameter</td>
</tr>
<tr>
<td>9339</td>
<td>247B</td>
<td>No socket available</td>
</tr>
<tr>
<td>9340</td>
<td>247C</td>
<td>Invalid MAC address</td>
</tr>
<tr>
<td>9346</td>
<td>2482</td>
<td>Invalid mode parameter</td>
</tr>
<tr>
<td>9347</td>
<td>2483</td>
<td>Maximum data length exceeded</td>
</tr>
<tr>
<td>9348</td>
<td>2484</td>
<td>Maximum number of messages exceeded</td>
</tr>
<tr>
<td>9349</td>
<td>2485</td>
<td>Maximum number of IP multicast groups exceeded</td>
</tr>
<tr>
<td>9350</td>
<td>2486</td>
<td>ARP not found in ARP cache</td>
</tr>
<tr>
<td>9356</td>
<td>2495</td>
<td>Unexpected response message received</td>
</tr>
<tr>
<td>9357</td>
<td>2496</td>
<td>Invalid message length</td>
</tr>
<tr>
<td>9368</td>
<td>2498</td>
<td>Unknown message command</td>
</tr>
<tr>
<td>9372</td>
<td>249C</td>
<td>Sequence error during transmission in sequence message mode</td>
</tr>
<tr>
<td>9374</td>
<td>249E</td>
<td>Command cannot be executed or command is currently executed</td>
</tr>
<tr>
<td>9416</td>
<td>24C8</td>
<td>Task not initialized</td>
</tr>
<tr>
<td>9426</td>
<td>24D2</td>
<td>No configuration data available</td>
</tr>
<tr>
<td>9428</td>
<td>24D4</td>
<td>Error while reading the configuration data</td>
</tr>
<tr>
<td>9429</td>
<td>24D5</td>
<td>Error while creating the diagnosis structure</td>
</tr>
<tr>
<td>9434</td>
<td>24DA</td>
<td>Not enough memory available</td>
</tr>
</tbody>
</table>
## Errors during execution of OpenModbus on TCP/IP, coupler response

<table>
<thead>
<tr>
<th>DEC</th>
<th>HEX</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9268</td>
<td>2434</td>
<td>Invalid parameter for &quot;ServerConnection&quot;</td>
</tr>
<tr>
<td>9269</td>
<td>2435</td>
<td>Invalid parameter for &quot;Task Timeout&quot;</td>
</tr>
<tr>
<td>9270</td>
<td>2436</td>
<td>Invalid parameter for &quot;OBM Timeout&quot;</td>
</tr>
<tr>
<td>9271</td>
<td>2437</td>
<td>Invalid parameter for &quot;Mode&quot;</td>
</tr>
<tr>
<td>9272</td>
<td>2438</td>
<td>Invalid parameter for &quot;Send Timeout&quot;</td>
</tr>
<tr>
<td>9273</td>
<td>2439</td>
<td>Invalid parameter for &quot;Connect Timeout&quot;</td>
</tr>
<tr>
<td>9274</td>
<td>243A</td>
<td>Invalid parameter for &quot;Close Timeout&quot;</td>
</tr>
<tr>
<td>9275</td>
<td>243B</td>
<td>Invalid parameter for &quot;Swab&quot;</td>
</tr>
<tr>
<td>9276</td>
<td>243C</td>
<td>TCP task not ready</td>
</tr>
<tr>
<td>9277</td>
<td>243D</td>
<td>PLC task not ready</td>
</tr>
<tr>
<td>9278</td>
<td>243E</td>
<td>Error during initialization</td>
</tr>
<tr>
<td>9327</td>
<td>246F</td>
<td>Unknown send or receive telegram</td>
</tr>
<tr>
<td>9328</td>
<td>2470</td>
<td>TCP responds with an error</td>
</tr>
<tr>
<td>9329</td>
<td>2471</td>
<td>No corresponding socket found</td>
</tr>
<tr>
<td>9330</td>
<td>2472</td>
<td>Command with invalid value</td>
</tr>
<tr>
<td>9331</td>
<td>2473</td>
<td>Error in TCP clock status</td>
</tr>
<tr>
<td>9332</td>
<td>2474</td>
<td>Error in Modbus telegram</td>
</tr>
<tr>
<td>9333</td>
<td>2475</td>
<td>No free socket found</td>
</tr>
<tr>
<td>9334</td>
<td>2476</td>
<td>TCP command is directed to an unknown socket</td>
</tr>
<tr>
<td>9335</td>
<td>2477</td>
<td>Time for a client job is over</td>
</tr>
<tr>
<td>9336</td>
<td>2478</td>
<td>Socket has been closed unexpectedly</td>
</tr>
<tr>
<td>9337</td>
<td>2479</td>
<td>Not-Ready flag has been set by the user</td>
</tr>
<tr>
<td>9338</td>
<td>247A</td>
<td>OMB task cannot open socket</td>
</tr>
<tr>
<td>9339</td>
<td>247B</td>
<td>Watchdog event in the PLC task; only in IO mode</td>
</tr>
<tr>
<td>9340</td>
<td>247C</td>
<td>TCP task not in configuration state</td>
</tr>
<tr>
<td>9341</td>
<td>247D</td>
<td>PLC task not initialized</td>
</tr>
<tr>
<td>9342</td>
<td>247E</td>
<td>Server socket was closed without response from the device</td>
</tr>
<tr>
<td>9347</td>
<td>2497</td>
<td>Invalid message length</td>
</tr>
<tr>
<td>9368</td>
<td>2498</td>
<td>Unknown message command</td>
</tr>
<tr>
<td>9376</td>
<td>24A0</td>
<td>Error in telegram header</td>
</tr>
<tr>
<td>9377</td>
<td>24A1</td>
<td>Invalid address detected in the telegram</td>
</tr>
<tr>
<td>9379</td>
<td>24A3</td>
<td>Invalid data address</td>
</tr>
<tr>
<td>9381</td>
<td>24A5</td>
<td>Invalid data counter</td>
</tr>
<tr>
<td>9383</td>
<td>24A7</td>
<td>OMB task received an error in the response of the TCP task</td>
</tr>
<tr>
<td>9418</td>
<td>24CA</td>
<td>OMB task does not have a segment from RCS</td>
</tr>
<tr>
<td>9419</td>
<td>24CB</td>
<td>Unknown or invalid sender contained in the command</td>
</tr>
</tbody>
</table>
Overview of blocks arranged according to their call names

Character description:

FBhv ... Function block with historical values
FBnohv... Function block without historical values
F  ...  Function

<table>
<thead>
<tr>
<th>CE name</th>
<th>Type</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH_AINIT</td>
<td>FBhv</td>
<td>Initializing the Ethernet UDP/IP data exchange</td>
<td>9</td>
</tr>
<tr>
<td>ETH_AREC</td>
<td>FBhv</td>
<td>Reading a data package from the Ethernet UDP/IP receive buffer</td>
<td>14</td>
</tr>
<tr>
<td>ETH_ASEND</td>
<td>FBhv</td>
<td>Transmitting a data package to a subscriber via Ethernet UDP/IP</td>
<td>17</td>
</tr>
<tr>
<td>ETH_ASTO</td>
<td>FBhv</td>
<td>Reading Ethernet UDP/IP timeout data packages from the timeout data buffer</td>
<td>20</td>
</tr>
<tr>
<td>ETH_INFO</td>
<td>FBnohv</td>
<td>Reading information about the installed couplers</td>
<td>7</td>
</tr>
<tr>
<td>ETH_MODSTAT</td>
<td>FBhv</td>
<td>Reading status information of the OpenModbus on TCP/IP processing</td>
<td>23</td>
</tr>
<tr>
<td>ETH_MODMAST</td>
<td>FBhv</td>
<td>Processing OpenModbus on TCP/IP client (master) telegrams</td>
<td>26</td>
</tr>
</tbody>
</table>
READING INFORMATION ABOUT THE INSTALLED COUPLERS

ETH_INFO

The block ETH_INFO outputs coupler related information. The following items are displayed: Coupler type and model, operation mode, manufacturing date, device number and serial number as well as firmware designation and firmware version.

Block data
Available as of PLC runtime system: V5.0
Included in library: Ethernet_S90_V50.LIB

Remark:
Included in library: Ethernet_S90_V50.LIB

Block type
Function block without historical values

Parameters

<table>
<thead>
<tr>
<th>Instance</th>
<th>ETH_INFO</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error message</td>
</tr>
<tr>
<td>TYP</td>
<td>STRING(17)</td>
<td>Coupler type</td>
</tr>
<tr>
<td>MODEL</td>
<td>STRING(22)</td>
<td>Coupler model</td>
</tr>
<tr>
<td>OP_MODE</td>
<td>STRING(22)</td>
<td>Operation mode of the coupler</td>
</tr>
<tr>
<td>MAN_DATE</td>
<td>DATE</td>
<td>Manufacturing date of the coupler</td>
</tr>
<tr>
<td>DEV_NO</td>
<td>DWORD</td>
<td>Device number of the coupler</td>
</tr>
<tr>
<td>SER_NO</td>
<td>DWORD</td>
<td>Serial number of the coupler</td>
</tr>
<tr>
<td>FW_NAME</td>
<td>STRING(17)</td>
<td>Designation of the coupler firmware</td>
</tr>
<tr>
<td>FW_VER</td>
<td>STRING(17)</td>
<td>Version of the coupler firmware</td>
</tr>
</tbody>
</table>

Description

The ETH_INFO block is always active. It reads the slot number at the block input CONO and outputs the corresponding information about the selected coupler.

ETH_INFO recognizes different coupler types. For that reason, the block outputs do not only relate to the Ethernet coupler.

The block is not intended for use in normal user programs. It should be used to support fault diagnosis and maintenance operations.

CONO

BYTE

At input CONO the slot (module number) of the coupler is applied for which the information should be polled. The module number depends on the PLC category.

The following applies:

| 07 KT 9x R027x | CONO = 0, 1 |
| 07 KT 9x R02x7 | CONO = 2 |

ERR

WORD

An error identification is provided at output ERR, if an invalid value is applied at input CONO. The error message encoding at output ERR applies to all Ethernet blocks and is explained at the beginning of the library descriptions.
TYP STRING(17)
The output TYP displays the coupler type (i.e. the coupler design). Different types are recognized by the block.
The Ethernet coupler has the type TYP = 'single mode'.
If a coupler type cannot be determined, the output TYP is set to 'unknown'.

MODEL STRING(22)
MODEL outputs the transfer protocol supported by the coupler as well as the subscriber type.
The model ID of the Ethernet coupler is MODEL = 'Ethernet'.
If the block detects an unknown model, MODEL outputs the value 'unknown'.

OP_MODE STRING(22)
Output OP_MODE displays the current coupler operation mode. This output is only applicable for couplers with switchable operation modes.
Therefore, this output is not used for the Ethernet coupler.

Function call in IL
CAL INFO(CONO := INFO_CONO)
LD INFO.ERR
ST INFO.ERR
LD INFO.TYP
ST INFO.TYP
LD INFO.MODEL
ST INFO.MODEL
LD INFO.OP_MODE
ST INFO.OP_MODE
LD INFO.MAN_DATE
ST INFO.MAN_DATE
LD INFO.DEV_NO
ST INFO.DEV_NO
LD INFO.SER_NO
ST INFO.SER_NO
LD INFO.FW_NAME
ST INFO.FW_NAME
LD INFO.FW_VER
ST INFO.FW_VER

Note:
In IL, the function call has to be performed in one line.

Function call in ST
INFO (CONO := INFO_CONO)
INFO.ERR := INFO.ERR;
INFO.TYP := INFO.TYP;
INFO.MODEL := INFO.MODEL;
INFO.OP_MODE := INFO.OP_MODE;
INFO.MAN_DATE := INFO.MAN_DATE;
INFO.DEV_NO := INFO.DEV_NO;
INFO.SER_NO := INFO.SER_NO;
INFO.FW_NAME := INFO.FW_NAME;
INFO.FW_VER := INIT.FW_VER;

Function call in ST

MAN_DATE DATE
Output MAN_DATE displays the manufacturing date of the coupler. The date is a variable of the data type DATE and has the format D#YYYY-MM-DD. Its initial value is D#2000-01-01.

DEV_NO DWORD
Output DEV_NO outputs the device number of the coupler.

SER_NO DWORD
Output SER_NO outputs the serial number of the coupler.

FW_NAME STRING(17)
Output FW_NAME provides the designation of the coupler firmware.

FW_VER STRING(17)
Output FW_VER displays the firmware release number (version) and the firmware issue date of the coupler. These data are represented as a string (e.g. 'V1.003 15.07.00').
INITIALIZING THE ETHERNET UDP/IP DATA EXCHANGE

ETH_AINIT initializes the Ethernet UDP/IP data exchange for the blocks ETH_ASEND, ETH_AREC and ETH_ASTO.

**ETH_AINIT**

<table>
<thead>
<tr>
<th>INIT</th>
<th>INIT_EN</th>
<th>EN</th>
<th>DONEN</th>
<th>DONEN</th>
<th>INIT_DONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>CONO</td>
<td>CONON</td>
<td>ERR</td>
<td>INIT_ERRN</td>
<td>INIT_STATN</td>
</tr>
<tr>
<td>INIT_CONO</td>
<td>CONO</td>
<td>ERR</td>
<td>INIT_ERR</td>
<td>INIT_STAT</td>
<td>INIT_STAT</td>
</tr>
<tr>
<td>INIT_LRH</td>
<td>LHBN</td>
<td>STAT</td>
<td>INIT.SOCK</td>
<td>INIT.SOCK</td>
<td>INIT.SOCK</td>
</tr>
<tr>
<td>INIT_LSHP</td>
<td>LSHP</td>
<td>IP_ADRN</td>
<td>INIT_IP_ADR</td>
<td>INIT_IP_ADR</td>
<td>INIT_IP_ADR</td>
</tr>
<tr>
<td>INIT_LSLB</td>
<td>LSLBN</td>
<td>SOCK</td>
<td>INIT.SOCK</td>
<td>INIT.SOCK</td>
<td>INIT.SOCK</td>
</tr>
<tr>
<td>INIT_LSTOB</td>
<td>LSTOB</td>
<td>CREBC</td>
<td>INIT_CREBC</td>
<td>INIT_CREBC</td>
<td>INIT_CREBC</td>
</tr>
<tr>
<td>INIT_LHEAD</td>
<td>LHEAD</td>
<td>LEVHP_BY</td>
<td>INIT_LEVHP_BY</td>
<td>INIT_LEVHP_BY</td>
<td>INIT_LEVHP_BY</td>
</tr>
<tr>
<td>INIT_ENEC</td>
<td>ENEC</td>
<td>LEVLP_BY</td>
<td>INIT_LEVLP_BY</td>
<td>INIT_LEVLP_BY</td>
<td>INIT_LEVLP_BY</td>
</tr>
<tr>
<td>INIT_ENRA</td>
<td>ENRA</td>
<td>LEVLP_DS</td>
<td>INIT_LEVLP_DS</td>
<td>INIT_LEVLP_DS</td>
<td>INIT_LEVLP_DS</td>
</tr>
<tr>
<td>INIT_ENRA</td>
<td>ENRA</td>
<td>LEVLTOS</td>
<td>INIT_LEVLTOS</td>
<td>INIT_LEVLTOS</td>
<td>INIT_LEVLTOS</td>
</tr>
</tbody>
</table>

**Block data**

Available as of PLC runtime system: V5.0

Included in library: Ethernet_S90_V50.LIB

**Remark:** Included in library: Ethernet_S90_V50.LIB

**Block type:** Function block with historical values

**Parameters**

<table>
<thead>
<tr>
<th>Instance</th>
<th>ETH_AINIT</th>
<th>Instance name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Initialization of the Ethernet UDP/IP data exchange with a FALSE/TRUE edge. A TRUE/FALSE edge stops the processing.</td>
</tr>
<tr>
<td>LRH</td>
<td>WORD</td>
<td>Receive buffer length in bytes</td>
</tr>
<tr>
<td>LSHP</td>
<td>WORD</td>
<td>Transmit buffer length for high priority in bytes</td>
</tr>
<tr>
<td>LSLB</td>
<td>WORD</td>
<td>Transmit buffer length for low priority in bytes</td>
</tr>
<tr>
<td>LSTOB</td>
<td>WORD</td>
<td>Buffer length for timeout data packages in bytes</td>
</tr>
<tr>
<td>LHEAD</td>
<td>WORD</td>
<td>Number of header data to be copied for timeout packages in bytes</td>
</tr>
<tr>
<td>ENBC</td>
<td>BOOL</td>
<td>Enable/disable broadcast reception (data packages to all stations) (currently not working)</td>
</tr>
<tr>
<td>ENRA</td>
<td>BOOL</td>
<td>Overwrite/disable receive buffer on buffer overflow</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Initialization completed or error occurred</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error number</td>
</tr>
<tr>
<td>STAT</td>
<td>WORD</td>
<td>UDP/IP processing state</td>
</tr>
<tr>
<td>IP_ADR</td>
<td>DWORD</td>
<td>IP address of the Ethernet card in slot CONO</td>
</tr>
<tr>
<td>SOCK</td>
<td>BYTE</td>
<td>Used socket number</td>
</tr>
<tr>
<td>CREBC</td>
<td>WORD</td>
<td>Counter for received broadcasts (data packages to all stations) (currently not working)</td>
</tr>
<tr>
<td>LEVHP BY</td>
<td>WORD</td>
<td>Filling level of the transmit buffer for high priority in bytes</td>
</tr>
<tr>
<td>LEVHP DS</td>
<td>WORD</td>
<td>Filling level of the transmit buffer for high priority in data records</td>
</tr>
<tr>
<td>LEVLP BY</td>
<td>WORD</td>
<td>Filling level of the transmit buffer for low priority in bytes</td>
</tr>
<tr>
<td>LEVLP DS</td>
<td>WORD</td>
<td>Filling level of the transmit buffer for low priority in data records</td>
</tr>
<tr>
<td>LEVR BY</td>
<td>WORD</td>
<td>Filling level of the receive buffer in bytes</td>
</tr>
<tr>
<td>LEVR DS</td>
<td>WORD</td>
<td>Filling level of the receive buffer in data records</td>
</tr>
<tr>
<td>LEVLTOS</td>
<td>WORD</td>
<td>Filling level of the timeout data packages buffer in data records</td>
</tr>
</tbody>
</table>

After a successful initialization, all outputs are updated with each block call if a TRUE signal is applied to input EN.
Description

The block ETH_AINIT initializes the Ethernet UDP/IP processing with variable buffer lengths. There is one transmit buffer for high priority and one for low priority as well as one receive buffer and a buffer for data packages which could not be transmitted successfully (timeout data packages). All buffers can be set variably (LRH, LSHB, LSLB, LSTOB).

If acknowledgement is active (refer to input TOUT of the block ETH_ASEND), the number of timeout data package header data to be copied can be specified via input LHEAD.

The reception of broadcast telegrams is enabled by applying a TRUE signal to input ENBC.

Output DONE indicates that the initialization was completed successfully or that an error occurred. If an error was detected during the processing of the block, the error is additionally indicated at output ERR. Furthermore, the block provides information about the filling levels of the transmit, receive and timeout buffers in bytes (LEVHP_BY, LEVLP_BY, LEVR_BY) and data records (LEVHP_DS, LEVLP_DS, LEVR_DS, LEVLTOS) and outputs the used IP address (IP_ADR) and socket number(SOCK).

The blocks ETH_ASEND, ETH_AREC and ETH_ASTO can only be used after successful initialization. The block ETH_AINIT may only be planned once for each Ethernet coupler.

EN  BOOL  A FALSE/TRUE edge at input EN initiates the initialization of the Ethernet UDP/IP data exchange. If no error is detected, data exchange via Ethernet UDP/IP is enabled. Input EN has to be TRUE as long as the data exchange is going on.

The following applies:
EN = FALSE: Initialization inactive. No data exchange possible.
EN = TRUE: Initialization is initiated. Data exchange is enabled.

CONO  BYTE  At input CONO the coupler slot (module number) is selected which should be used by this block. The module number depends on the PLC category.

The following applies:
07 KT 9x R027x CONO = 0, 1
07 KT 9x R02x7 CONO = 2

LRH  WORD  At input LRH the desired receive buffer length is specified in bytes.

LSHB  WORD  At input LSHB the desired high priority transmit buffer length is specified in bytes.

LSLB  WORD  At input LSLB the desired low priority transmit buffer length is specified in bytes.

LSTOB  WORD  At input LSTOB the desired buffer length for timeout data packages is specified in bytes.

LHEAD  WORD  Input LHEAD determines the number of timeout data package header data to be copied into the readable timeout buffer (refer to block ETH_ASTO).

ENBC  BOOL  Input ENBC is used to enable or disable the reception of broadcasts (data packages to all stations).

The following applies:
ENBC = FALSE  Broadcast reception disabled
ENBC = TRUE  Broadcast reception enabled

(currently not working)

ENRA  BOOL  Input ENRA is used to determine how new incoming data packages are to be handled if the receive buffer is full.

The following applies:
ENRA = FALSE  The oldest data packages stored in the receive buffer are overwritten with the new incoming data packages.
ENRA = TRUE  The new incoming data are dismissed.
DONE BOOL
Output DONE indicates that the initialization process is completed and data exchange is enabled or that an error occurred during the block processing. This is why the output has always to be considered together with output ERR.

Output DONE is set to TRUE for one cycle.
The following applies:
DONE = FALSE
ERR = ---
Initialization is not completed
DONE = TRUE
ERR = 0
Initialization is completed and data exchange is enabled
DONE = TRUE
ERR <> 0
An error occurred during the initialization process. The error can be evaluated at output ERR.

ERR WORD
Output ERR provides an error identifier if an invalid value was applied to an input or if an error occurred during the request processing.

Output ERR is set to TRUE for one cycle.
The error message encoding at output ERR applies to all ETH_AXXX blocks and is explained at the beginning of the library descriptions (refer to Error messages of the Ethernet library blocks).

STAT BYTE
Output STAT displays the current UDP/IP processing state.

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>HEX</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0002</td>
</tr>
<tr>
<td>16</td>
<td>0010</td>
</tr>
<tr>
<td>32</td>
<td>0020</td>
</tr>
<tr>
<td>64</td>
<td>0040</td>
</tr>
<tr>
<td>128</td>
<td>0080</td>
</tr>
<tr>
<td>256</td>
<td>0100</td>
</tr>
<tr>
<td>512</td>
<td>0200</td>
</tr>
<tr>
<td>1024</td>
<td>0400</td>
</tr>
<tr>
<td>2048</td>
<td>0800</td>
</tr>
<tr>
<td>2049</td>
<td>0801</td>
</tr>
<tr>
<td>2050</td>
<td>0802</td>
</tr>
<tr>
<td>4096</td>
<td>1000</td>
</tr>
<tr>
<td>4097</td>
<td>1001</td>
</tr>
<tr>
<td>4098</td>
<td>1002</td>
</tr>
<tr>
<td>6144</td>
<td>1800</td>
</tr>
<tr>
<td>6145</td>
<td>1801</td>
</tr>
<tr>
<td>6146</td>
<td>1802</td>
</tr>
<tr>
<td>8192</td>
<td>2000</td>
</tr>
<tr>
<td>16384</td>
<td>4000</td>
</tr>
<tr>
<td>32768</td>
<td>8000</td>
</tr>
</tbody>
</table>

IP_ADR DWORD
Output IP_ADR displays the block's own IP address. Each byte in IP_ADR represents one octet of the address.

Example:
IP address 192.15.24.2,
IP_ADR (hex) 16#C00F1802,
IP_ADR (dez) 3222214658
SOCK BYTE
Output SOCK displays the socket number used for Ethernet UDP/IP processing.

CREBC WORD
Output CREBC displays the number of broadcasts (data packages to all stations) received by this station since the occurrence of the FALSE/TRUE edge at input EN.

(currently not working)

LEVHP_BY WORD
As long as EN = TRUE, output LEVHP_BY displays the filling level of the high priority transmit buffer in bytes.

LEVHP_DS WORD
As long as EN = TRUE, output LEVHP_DS displays the filling level of the high priority transmit buffer in data records.

LEVLP_BY WORD
As long as EN = TRUE, output LEVLP_BY displays the filling level of the low priority transmit buffer in bytes.

LEVLP_DS WORD
As long as EN = TRUE, output LEVLP_DS displays the filling level of the low priority transmit buffer in data records.

LEVR_BY WORD
As long as EN = TRUE, output LEVR_BY displays the filling level of the receive buffer in bytes.

LEVR_DS WORD
As long as EN = TRUE, output LEVR_DS displays the filling level of the receive buffer in data records.

LEVLTOS WORD
As long as EN = TRUE, output LEVLTOS displays the filling level of the timeout buffer in data records.
Function call in IL

CAL INIT(EN := INIT_EN,
CONO := INIT_CONO,
LRH := INIT_LRH,
LSHB := INIT_LSHB,
LSLB := INIT_LSLB,
LSTOB := INIT_LSTOB,
LHEAD := INIT_LHEAD,
ENBC := INIT_ENBC,
ENRA := INIT_ENRA)

LD INIT.DONE
ST INIT_DONE

LD INIT.ERR
ST INIT_ERR

LD INIT.STAT
ST INIT_STAT

LD INIT.IP_ADR
ST INIT_IP_ADR

LD INIT.SOCK
ST INIT.SOCK

LD INIT.CREBC
ST INIT_CREBC

LD INIT.LEVHP_BY
ST INIT_LEVHP_BY

LD INIT.LEVHP_DS
ST INIT_LEVHP_DS

LD INIT.LEVLP_BY
ST INIT_LEVLP_BY

LD INIT.LEVLP_DS
ST INIT_LEVLP_DS

LD INIT.LEVR_BY
ST INIT_LEVR_BY

LD INIT.LEVR_DS
ST INIT_LEVR_DS

LD INIT.LEVLTOS
ST INIT_LEVLTOS

Note:
In IL, the function call has to be performed in one line.

Function call in ST

INIT (EN := INIT_EN,
CONO := INIT_CONO,
LRH := INIT_LRH,
LSHB := INIT_LSHB,
LSLB := INIT_LSLB,
LSTOB := INIT_LSTOB,
LHEAD := INIT_LHEAD,
ENBC := INIT_ENBC,
ENRA := INIT_ENRA);

INIT_DONE := INIT_DONE;
INIT_ERR := INIT_ERR;
INIT_STAT := INIT_STAT;
INIT_IP_ADR := INIT_IP_ADR;
INIT.SOCK := INIT.SOCK;
INIT_CREBC := INIT_CREBC;
INIT_LEVHP_BY := INIT_LEVHP_BY;
INIT_LEVHP_DS := INIT_LEVHP_DS;
INIT_LEVLP_BY := INIT_LEVLP_BY;
INIT_LEVLP_DS := INIT_LEVLP_DS;
INIT_LEVR_BY := INIT_LEVR_BY;
INIT_LEVR_DS := INIT_LEVR_DS;
INIT_LEVLTOS := INIT_LEVLTOS;
READING AN ETHERNET UDP/IP DATA PACKAGE FROM THE RECEIVE BUFFER

ETH_AREC

The ETH_AREC block reads the next data record from the UDP/IP receive buffer and supplies the user data in the planned memory area.

Block data

Available as of PLC runtime system: V5.0
Included in library: Ethernet_S90_V50.LIB

Block type

Function block with historical values

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>ETH_AREC</td>
<td>Instance name</td>
</tr>
<tr>
<td>EN</td>
<td>BOOL</td>
<td>Enabling of the block processing</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
<td>Slot (module number) of the coupler</td>
</tr>
<tr>
<td>DATA</td>
<td>DWORD</td>
<td>Variable in which the received user data shall be stored. The variable has to be of the type ARRAY or STRUCT.</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
<td>Data package available or error occurred</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
<td>Error number</td>
</tr>
<tr>
<td>IP_ADR</td>
<td>WORD</td>
<td>IP address of the sending device</td>
</tr>
<tr>
<td>LEN</td>
<td>WORD</td>
<td>Data package length in bytes</td>
</tr>
<tr>
<td>LEVR_BY</td>
<td>WORD</td>
<td>Receive buffer filling level in bytes</td>
</tr>
<tr>
<td>LEVR_DS</td>
<td>WORD</td>
<td>Receive buffer filling level in data records</td>
</tr>
</tbody>
</table>

Description

The operating system reads the received UDP/IP data packages from the ETHERNET coupler and stores them in the receive buffer. The buffer size is determined using the ETH_AINIT block. The data packages are stored with variable lengths. For example, a data package consisting of 16 bytes of user data occupies exactly 22 bytes in the receive buffer (4 bytes for the IP address of the sending device, 2 bytes for the packet length and 16 bytes of user data).

Using the ETH_AREC block exactly one data package is read. The user data are stored in the planned memory area (DATA). The address of the sending device and the data package length are supplied at the outputs IP_ADR and LEN. DONE = TRUE and ERR = 0 indicate that the reading process was successful. If an error was detected during the block processing, the error is indicated at output ERR.

Furthermore, the block provides information about the receive buffer filling level displayed in bytes (LEVR_BY) and data records (LEVR_DS).

The ETH_AREC block cannot read data packages from the receive buffer until the ETHERNET UDP/IP processing has been initialized by the ETH_AINIT block.
EN \text{ BOOL}
Reading of the receive buffer is performed depending on the signal applied at input EN.
The following applies:
EN = FALSE: Do not read receive buffer
EN = TRUE: Read receive buffer

CONO \text{ BYTE}
At input CONO the coupler slot (module number) is selected which should be used by this block. The module number depends on the PLC category.
The following applies:
07 KT 9x R027x CONO = 0, 1
07 KT 9x R02x7 CONO = 2

DATA \text{ DWORD}
Input DATA is used to specify the variable address to which the user data shall be copied. The address specified at DATA has to belong to a variable of the type ARRAY or STRUCT.
CAUTION: Adjust the variable size to the maximum expected amount of data in order to avoid overlapping of memory areas.

DONE \text{ BOOL}
Output DONE indicates that the user data of a data package were continuously copied from the receive buffer into the memory area of the variable specified at DATA or that the block processing was aborted due to an occurring error. This is why the output has always to be considered together with output ERR.
Output DONE is set to TRUE for one cycle.
The following applies:
DONE = FALSE
ERR = ---:
The data package was not read from the receive buffer.
DONE = TRUE
ERR = 0:
The data package was successfully read from the receive buffer.
DONE = TRUE
ERR <> 0:
An error occurred while reading the user data from the receive buffer. The user data were not copied to the area specified at DATA. The error can be evaluated at output ERR.

ERR \text{ WORD}
Output ERR provides an error identifier for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing the job. ERR has always to be considered together with output DONE. The value provided at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all ETH_Axxx blocks and is explained at the beginning of the library descriptions (refer to Error messages of the Ethernet library blocks).

IP_ADR \text{ DWORD}
Output IP_ADR displays the IP address of the sending device which transmitted the received data package. Each byte in IP_ADR represents one octet of the address.
Example:
IP address 192.15.24.2,
IP_ADR (hex) 16#C00F1802,
IP_ADR (dez) 3222214658

LEN \text{ WORD}
Output LEN displays the length of the received data package in bytes.

LEVR_BY \text{ WORD}
Output LEVR_BY displays the filling level of the receive buffer in bytes. The displayed value is updated as long as EN is TRUE and applies to the input values read with the rising edge at input EN.
One data package occupies output LEN + 6 bytes in the receive buffer (4 bytes for the IP address of the sending device, 2 bytes for the specification of the length).

LEVR_DS \text{ WORD}
Output LEVR_DS displays the filling level of the receive buffer in data records. The displayed value is updated as long as EN is TRUE and applies to the input values read with the rising edge at input EN.
READING AN ETHERNET UDP/IP DATA PACKAGE
FROM THE RECEIVE BUFFER

Function call in IL

LD REC_DATA
ADR
ST REC_DATA
CAL REC(EN := REC_EN,
CONO := REC_CONO)
LD REC_DONE
ST REC_DONE
LD REC.ERR
ST REC.ERR
LD REC.IP_ADR
ST REC.IP_ADR
LD REC.LEN
ST REC.LEN
LD REC.LEVR_BY
ST REC.LEVR_BY
LD REC.LEVR_DS
ST REC.LEVR_DS

Function call in ST

REC (EN := REC_EN,
CONO := REC_CONO,
DATA := ADR(REC_DATA));
REC_DONE := REC_DONE;
REC.ERR := REC.ERR;
REC.IP_ADR := REC.IP_ADR;
REC.LEN := REC.LEN;
REC.LEVR_BY := REC.LEVR_BY;
REC.LEVR_DS := REC.LEVR_DS;

Note:
In IL, the function call has to be performed in one line.
TRANSMITTING AN ETHERNET UDP/IP DATA PACKAGE

The block ETH_ASEND is used to transmit data packages via the UDP/IP protocol of the ETHERNET coupler.

Block data

- Available as of PLC runtime system: V5.0
- Included in library: Ethernet_S90_V50.LIB

Block type

Function block with historical values

Parameters

- Instance ETH_ASEND Instance name
- EN BOOL Enabling of data package transmission
  - FALSE: do not transmit data package
  - TRUE: transmit data package
- CONO BYTE Slot (module number) of the coupler
- IP_ADR DWORD Target IP address to which the data shall be transmitted
- PRIO BOOL Transmission priority of the data package
  - FALSE = low, TRUE = high
- TOUT WORD Timeout period of the data package in ms
- DATA DWORD Address of the variable from which on the data shall be copied into the transmit buffer. The variable has to be of the type ARRAY or STRUCT.
- LEN WORD Number of user data to be transmitted (in bytes)
- DONE BOOL Data package stored in transmit buffer or error occurred
- ERR WORD Error number
- LEV_BY WORD Filling level (in bytes) of the transmit buffer for low/high priority (depending on input PRIO)
- LEV_DS WORD Filling level (in data records) of the transmit buffer for low/high priority (depending on input PRIO)

Description

The block ETH_ASEND is used to transmit data packages via the UDP/IP protocol of the ETHERNET coupler. The planned packages are stored in the transmit buffer selected via input PRIO. From there, the operating system hands over the data packages to the ETHERNET coupler in order to transmit them to the target address specified at input IP_ADR. The transmit buffer size is determined using the ETH_AINIT block. Using input TOUT the timeout period can be specified. If TOUT <> 0, the UDP/IP data exchange is automatically performed with receive acknowledgement. If TOUT = 0, no acknowledgement is expected. Output DONE indicates that the planned data package has been stored in the transmit buffer or that an error occurred during block processing. If an error was detected during block processing, the error is additionally indicated at output ERR. In case of an error the data package has to be transmitted again.

The ETH_ASEND block cannot store data packages to the transmit buffer until the ETHERNET UDP/IP processing has been initialized by the ETH_AINIT block.
TRANSMITTING AN ETHERNET UDP/IP DATA PACKAGE

**EN**  
BOOL  
Depending on input EN, the planned packet is first stored in the transmit buffer and then transmitted.  
The following applies:  
EN = FALSE: The planned packet is not stored in the transmit buffer and therefore not transmitted.  
EN = TRUE: The planned packet is stored in the transmit buffer and transmitted.  

**CONO**  
BYTE  
At input CONO the coupler slot (module number) is selected which should be used by this block. The module number depends on the PLC category.  
The following applies:  
07 KT 9x R027x CONO = 0, 1  
07 KT 9x R02x7 CONO = 2  

**IP_ADR**  
DWORD  
At this input the IP address of the recipient is specified. Each byte in IP_ADR represents one octet of the address.  
Example:  
IP address 192.15.24.2,  
IP_ADR (hex) 16#C00F1802,  
IP_ADR (dez) 3222214658  

**PRIO**  
BOOL  
Input PRIO is used to specify the transmit priority of the data package.  
The following applies:  
PRIO = FALSE: The planned data package has low priority. Thus it is stored in the low priority transmit buffer. All outputs refer to this buffer.  
PRIO = TRUE: The planned data package has high priority. Thus it is stored in the high priority transmit buffer. All outputs refer to this buffer.  

**TOUT**  
WORD  
Using input TOUT the timeout period can be specified. If TOUT <> 0, the UDP/IP data exchange is automatically performed with receive acknowledgement. If a data package cannot be transmitted within this period (no acknowledge telegram is received), transmission is aborted and the package is lost.  
In this case, some distinctive bytes of the data package (refer to input LHEAD of the ETH_AINIT block) are stored in the timeout buffer and can then be read using the ETH_ASTO block.  

If TOUT = 0, no acknowledgement is expected.  
The following applies:  
TOUT = 0: Data exchange without receive acknowledgement. No data are written to the timeout buffer.  
TOUT <> 0: Data exchange with receive acknowledgement. Each transmitted data record is acknowledged by the recipient. If no acknowledge telegram is received within the set timeout period (in ms), the data are written to the timeout buffer.  

**DATA**  
DWORD  
At input DATA the address of the variable is specified the data of which are transmitted as user data in this package. The address specified at DATA has to belong to a variable of the type ARRAY or STRUCT.  

**LEN**  
WORD  
At input LEN the number of user data bytes is specified for the planned packet.  
The following applies:  
1 ≤ LEN ≤ 1464  

**DONE**  
BOOL  
Output DONE indicates that the planned package has been stored in the transmit buffer or that an error occurred during block processing. This is why the output has always to be considered together with output ERR.  
Output DONE is set to TRUE for one cycle.  
The following applies:  
DONE = FALSE  
ERR = ---: The planned packet has not been stored in the transmit buffer.  
DONE = TRUE  
ERR = 0: The planned packet has been stored in the transmit buffer.  
DONE = TRUE  
ERR <> 0: An error occurred during transmission. The planned data packet was stored in the transmit buffer. The error can be evaluated at output ERR.
ERR  WORD
Output ERR provides an error identifier for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing the job. ERR has always to be considered together with output DONE. The value provided at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all ETH_AXXX blocks and is explained at the beginning of the library descriptions (refer to Error messages of the Ethernet library blocks).

LEV_BY  WORD
Output LEV_BY displays the filling level (in bytes) of the transmit buffer selected at input PRIO. The displayed value is updated as long as EN is TRUE and applies to the input values read with the rising edge at input EN.

One data package occupies output LEN + 8 bytes in the transmit buffer (4 bytes for the IP address of the recipient, 2 bytes for the specification of the length and 2 bytes for the timeout period).

LEV_DS  WORD
Output LEV_DS displays the filling level (in data records) of the transmit buffer selected at input PRIO. The displayed value is updated as long as EN is TRUE and applies to the input values read with the rising edge at input EN.

---

**Function call in IL**

LD SEND_DATA
ADR
ST SEND.DATA
CAL SEND(EN := SEND_EN,
CONO := SEND_CONO,
IP_ADR := SEND_IP_ADR,
PRIO := SEND_PRIO,
TOUT := SEND_TOUT,
LEN := SEND_LEN)
LD SEND_DONE
ST SEND_DONE
LD SEND.ERR
ST SEND_ERR
LD SEND.LEV_BY
ST SEND.LEV_BY
LD SEND.LEV_DS
ST SEND.LEV_DS

Note:
In IL, the function call has to be performed in one line.

**Function call in ST**

SEND (EN := SEND_EN,
CONO := SEND_CONO,
IP_ADR := SEND_IP_ADR,
PRIO := SEND_PRIO,
TOUT := SEND_TOUT,
DATA := ADR(SEND_DATA),
LEN := SEND_LEN);
SEND_DONE := SEND_DONE;
SEND_ERR := SEND.ERR;
SEND_LEV_BY := SEND.LEV_BY;
SEND_LEV_DS := SEND.LEV_DS;
READING ETHERNET UDP/IP TIMEOUT DATA PACKAGES

The ETH_ASTO block reads lost data packages from the timeout data buffer and supplies the user data in the planned memory area.

Block data
Available as of PLC runtime system: V5.0
Included in library: Ethernet_S90_V50.LIB

Block type
Function block with historical values

Parameters
Instance ETH_ASTO Instance name
EN BOOL Enabling of the block processing
CONO BYTE Slot (module number) of the coupler
DATA DWORD Variable in which the data of the timeout package are stored. The variable has to be of the type ARRAY or STRUCT.
DONE BOOL Data package available or error occurred
ERR WORD Error number
LEVDS WORD Timeout buffer filling level in data records

Description
During the transmission of a data package the success of the transmission is monitored by an adjustable timeout period. When this time is exceeded, distinctive information of the data package are stored in the timeout buffer. These are:
- the IP address of the recipient (4 bytes) and
- various header data of the data record (the amount of header data is planned using input LHEAD of the ETH_AINIT block).

The buffer length can be set at input LSTOB of the ETH_AINIT block. The buffer is constructed as a circular buffer (FIFO). If the buffer is full, the oldest entry in the buffer is overwritten. When a rising edge occurs at input EN, the ETH_ASTO block verifies whether a data package is stored in the buffer and makes the information mentioned above from the variable specified at input DATA on available for the user.

The ETH_ASTO block cannot be used until the ETHERNET UDP/IP processing has been initialized by the ETH_AINIT block. Additionally, input TOUT of the transmit block ETH_ASEND must be <> 0.

EN BOOL
Reading of the timeout buffer is performed depending on the signal state applied at input EN.

The following applies:
EN = FALSE:
The timeout buffer is not read.
EN = TRUE:
The timeout buffer is read.

CONO BYTE
At input CONO the coupler slot (module number) is selected which should be used by this block. The module number depends on the PLC category.

The following applies:
07 KT 9x R027x CONO = 0, 1
07 KT 9x R02x7 CONO = 2
Data DWORD
Input DATA is used to specify the variable address to which the user data shall be copied. The address specified at DATA has to belong to a variable of the type ARRAY or STRUCT.

CAUTION: Adjust the variable size to the maximum expected amount of data in order to avoid overlapping of memory areas.

Done BOOL
Output DONE indicates that the information of a data package was continuously copied from the timeout buffer into the memory area of the variable specified at DATA or that the block processing was aborted due to an occurring error. This is why the output has always to be considered together with output ERR.

Output DONE is set to TRUE for one cycle.

The following applies:

DONE = FALSE
ERR = ---:
The information of a data package in the timeout buffer have not been read.

DONE = TRUE
ERR = 0:
The information of a data package in the timeout buffer have been read.

DONE = TRUE
ERR <> 0:
An error occurred while reading the information of a data package from the timeout buffer. The data were not copied to the area specified at DATA. The error can be evaluated at output ERR.

Err Word
Output ERR provides an error identifier for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing the job. ERR has always to be considered together with output DONE. The value provided at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all ETH_Axxx blocks and is explained at the beginning of the library descriptions (refer to Error messages of the Ethernet library blocks).

Levds Word
Output LEVDS displays the filling level of the timeout buffer in data records. The displayed value is updated as long as EN is TRUE and applies to the input values read with the rising edge at input EN.
### Function call in IL

<table>
<thead>
<tr>
<th>LD</th>
<th>STO_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>STO.DATA</td>
</tr>
<tr>
<td>ST</td>
<td>STO.DATA</td>
</tr>
<tr>
<td>CAL</td>
<td>STO(EN := STO_EN, CONO := STO_CONO)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LD</th>
<th>STO.DONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>STO.DONE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LD</th>
<th>STO.ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>STO.ERR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LD</th>
<th>STO.LEVDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>STO.LEVDS</td>
</tr>
</tbody>
</table>

**Note:**

In IL, the function call has to be performed in one line.

### Function call in ST

```
STO(EN := STO_EN,
    CONO := STO_CONO,
    DATA := ADR(STO_DATA));

STO_DONE := STO_DONE;
STO_ERR := STO_ERR;
STO_LEVDS := STO.LEVDS;
```
ETH_MODSTAT reads the status information of the OpenModbus on TCP/IP processing. It can be used for pure server (slave) or client (master) operation of the controller as well as for mixed operation.

ETH_MODSTAT

```
<table>
<thead>
<tr>
<th>Instance</th>
<th>ETH_MODSTAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>BOOL</td>
</tr>
<tr>
<td>CONO</td>
<td>BYTE</td>
</tr>
<tr>
<td>DONE</td>
<td>BOOL</td>
</tr>
<tr>
<td>ERR</td>
<td>WORD</td>
</tr>
<tr>
<td>IP_ADR</td>
<td>DWORD</td>
</tr>
<tr>
<td>NUM_SRV</td>
<td>BYTE</td>
</tr>
<tr>
<td>RDY</td>
<td>BOOL</td>
</tr>
<tr>
<td>STAT</td>
<td>BYTE</td>
</tr>
<tr>
<td>NUM_ERRS</td>
<td>DWORD</td>
</tr>
<tr>
<td>LAST_ERR</td>
<td>WORD</td>
</tr>
</tbody>
</table>
```

**Description**

Using the ETH_MODSTAT block various status information about the OpenModbus processing can be read.

Every time a FALSE → TRUE edge is applied to input EN, MODSTAT acquires the value applied at input CONO and reads the corresponding information from the coupler.

**Parameters**

- **EN**: BOOL
  - The block can be activated (EN = TRUE) or deactivated (EN = FALSE) via input EN. If the block is active, the current values are applied to the outputs. If the block finds an Ethernet coupler with OpenModbus on TCP/IP functionality in the specified slot CONO, this is indicated by DONE = TRUE and ERR = 0 and the corresponding status information are output at the block outputs.
- **DONE**: BOOL
  - Data package available or error occurred
- **ERR**: WORD
  - Error number
- **IP_ADR**: DWORD
  - IP address of the controller operating as OpenModbus subscriber
- **NUM_SRV**: BYTE
  - Number of configured server channels
- **RDY**: BOOL
  - Ready indication of the OpenModbus processing
- **STAT**: BYTE
  - Status of the OpenModbus processing
- **NUM_ERRS**: DWORD
  - Number of occurred errors
- **LAST_ERR**: WORD
  - Identifier of the last occurred error
CONO BYTE

At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

The following applies:
07 KT 9x R027x CONO = 0, 1
07 KT 9x R02x7 CONO = 2

DONE BOOL

DONE reflects the state of the request processing. The value at output DONE corresponds to the value at input EN.

ERR WORD

Output ERR provides an error identifier if an error occurred during processing. ERR has always to be considered together with output DONE. The value provided at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all Ethernet blocks and is explained at the beginning of the library descriptions (refer to Error messages of the Ethernet library blocks).

Example:
IP address 192.15.24.2,
IP_ADR (hex) 16#C00F1802,
IP_ADR (dez) 3222214658

NUM_SRV BYTE

Output NUM_SRV indicates the number of parallel server channels configured with 907 FB 1131.
NUM_SRV is only valid if DONE = TRUE and ERR = 0.

RDY BOOL

RDY indicates the readiness for operation of the OpenModbus on TCP/IP processing. If RDY = TRUE the server processing as well as the client processing are ready. RDY is only valid if DONE = TRUE and ERR = 0.

STAT BYTE

Output STAT displays the current operating state of the OpenModbus on TCP/IP processing. STAT is only valid if DONE = TRUE and ERR = 0.

<table>
<thead>
<tr>
<th>DEC</th>
<th>HEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Processing not initialized</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Processing initialized and running</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Processing initialization in progress</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>Initialization error</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>Processing initialized and waiting for TCP task</td>
</tr>
</tbody>
</table>

NUM_ERRS DWORD

Output NUM_ERRS displays the number of errors that occurred on the Ethernet coupler.

LAST_ERR WORD

LAST_ERR outputs the last occurring error on the coupler. The error message encoding at output LAST_ERR applies to all Ethernet blocks and is explained at the beginning of the library descriptions.
Function call in IL

CAL  STAT(EN := STAT_EN, CONO := STAT_CONO)
LD   STAT_DONE
ST   STAT_DONE
LD   STAT_ERR
ST   STAT_ERR
LD   STAT.IP_ADR
ST   STAT.IP_ADR
LD   STAT.NUM_SRV
ST   STAT.NUM_SRV
LD   STAT.RDY
ST   STAT.RDY
LD   STAT.STATE
ST   STAT.STATE
LD   STAT.NUM_ERRS
ST   STAT_NUM_ERRS
LD   STAT.LAST_ERR
ST   STAT.LAST_ERR

Note:
In IL, the function call has to be performed in one line.

Function call in ST

STAT(EN := STAT_EN,
    CONO := STAT_CONO);
STAT_DONE := STAT_DONE;
STAT_ERR := STAT.ERR;
STAT.IP_ADR := STAT.IP_ADR;
STAT_NUM_SRV := STAT.NUM_SRV;
STAT.RDY := STAT.RDY;
STAT.STATE := STAT.STATE;
STAT_NUM_ERRS := STAT.NUM_ERRS;
STAT_LAST_ERR := STAT.LAST_ERR
The ETH_MODMAST block can be used to send an OpenModbus on TCP/IP telegram to a server (slave) and to process the corresponding response.

**Block data**
- **Available as of PLC runtime system:** V5.0
- **Included in library:** Ethernet_S90_V50.LIB

**Block type**
- Function block with historical values

**Parameters**
- **Instance** ETH_MODMAST
- **EN** BOOL: Enabling of the block processing
- **CONO** BYTE: Slot (module number) of the coupler
- **IP_ADR** DWORD: IP address of the server
- **FCT** BYTE: Modbus function code
- **ADDR** WORD: Operand/register address in the server
- **NB** WORD: Number of the data to be read/written
- **DATA** DWORD: Address of the first operand in the client from which data shall be written to the server or where the data read from the server shall be stored
- **DONE** BOOL: Ready message of the block
- **ERR** WORD: Error message

**Description**

The ETH_MODMAST block implements the OpenModbus on TCP/IP client functionality for the Ethernet coupler specified at input CONO. Depending on the configuration of the coupler, several ETH_MODMAST blocks can be used in parallel. Prior to the use of ETH_MODMAST for an Ethernet coupler the coupler has to be configured accordingly by means of 907 FB 1131.

With each FALSE -> TRUE edge at input EN, the function block ETH_MODMAST reads the values at the inputs, generates a telegram according to the inputs and then sends this telegram to the slave.

**EN** BOOL
- If a FALSE -> TRUE edge is applied to input EN, all further inputs are read in. If the input values are valid, a request telegram is sent to the specified server. If at least one input is invalid, no telegram is generated and the error is displayed at output ERR instead.
- While the request is processed, state changes at input EN are recognized but not evaluated.
CONO BYTE
At input CONO the slot (module number) of the coupler is applied which should process the request. The module number depends on the PLC category.

The following applies:
07 KT 9x R027x CONO = 0, 1
07 KT 9x R02x7 CONO = 2

IP_ADR DWORD
At IP_ADR the IP address of the server has to be specified to which the telegram should be sent. Each byte in IP_ADR represents one octet of the address.

Example:
IP address 192.15.24.2,
IP_ADR (hex) 16#C00F1802,
IP_ADR (dez) 3222214658

FCT BYTE
The function code of the request telegram is specified at input FCT. The following function codes are supported:
01 or 02 read n bits
03 or 04 read n words
05 write one bit
06 write one word
07 read M01,00...M01,07
15 write n bits
16 write n words

ADDR WORD
Input ADDR is used to specify the operand/register address in the server from which on data should be read or written. The access to the operands of AC31 devices operating in OpenModbus on TCP/IP server mode is defined via the MODBUS cross-reference list (refer to documentation for the MODMAST block). Only operands that are listed in the cross-reference list may be used. When accessing other devices ADDR is freely selectable and the valid ranges have to be gathered from the corresponding device description.

NB WORD
At input NB the number of data to be written or read is specified. The unit of NB depends on the selected function. For bit accesses the number of bits, for word and double word accesses the number of words is specified at NB. The following restrictions apply to the length:

FCT Nb max
01 or 02 255 bits
03 or 04 100 words / 50 double words
05 1 bit
06 1 word
07 8 bits
15 255 bits
16 100 words / 50 double words

DATA DWORD
At input DATA the address of the first operand in the client, from which on the data shall be written to the server or where the data read by the server shall be stored, is specified via the ADR operator. For this purpose it is necessary that the operand type (e.g. bit) matches the selected function (e.g. FCT 1, read n bits).

DONE BOOL
Output DONE reflects the state of the request processing. If the block is trigged by a FALSE → TRUE edge at input EN, DONE is set to FALSE in order to indicate that the request processing is running. If task processing is completed, DONE is set to TRUE for one cycle.

ERR WORD
Output ERR provides an error identifier for a period of one cycle if an invalid value was applied to an input or if an error occurred during processing the job. ERR has always to be considered together with output DONE. The value provided at ERR is only valid if DONE = TRUE. The error message encoding at output ERR applies to all Ethernet blocks and is explained at the beginning of the library descriptions. (Refer to Error messages of the Ethernet library blocks.)
Function call in IL

LD    MAST_DATA
ADR   ST    MAST.DATA
CAL   MAST
      (EN := MAST_EN,
       CONO := MAST_CONO,
       IP_ADR := MAST_IP_ADR,
       FCT := MAST_FCT,
       ADDR := MAST_ADDR,
       NB := MAST_NB,
       DATA := MAST_DATA)

LD MAST.DONE
ST MAST_DONE
LD MAST.ERR
ST MAST_ERR

Note:
In IL, the function call has to be performed in one line.

Function call in ST

MAST(EN := MAST_EN,
     CONO := MAST_CONO,
     IP_ADR := MAST_IP_ADR,
     FCT := MAST_FCT,
     ADDR := MAST_ADDR,
     NB := MAST_NB,
     DATA := ADR(MAST_DATA),);

MAST_DONE := MAST_DONE;
MAST_ERR := MAST_ERR;
Glossary

**BOOL**
Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit memory space are reserved.

**DINT**
DINT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>-2147483648</td>
<td>2147483647</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

**DWORD**
DWORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWORD</td>
<td>0</td>
<td>4294967295</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

**INT**
INT belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>-32768</td>
<td>32767</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.

**WORD**
WORD belongs to the integer data types.

The different numerical types are responsible for a different numerical range. For integer data types the following range limits are valid:

<table>
<thead>
<tr>
<th>Type</th>
<th>Lower limit</th>
<th>Upper limit</th>
<th>Memory space</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>0</td>
<td>65535</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Due to this, it is possible that information are lost when converting greater data types to smaller data types.
Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on.

For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can be derived:

- Within functions, global variables can neither be read nor written.
- Within functions, absolute operands can neither be read nor written.
- Within functions, function blocks must not be called.

Function blocks

Function blocks are subroutines which can have as many inputs, outputs and internal variables as required. They are called from a program or from another function block.

As they can be used several times (with different data records), function blocks (code and interface) can be considered as type. When assigning an individual data record (declaration) to the function block, a function block instance is generated.

In contrast to functions, function blocks can contain statically local data which are saved from one call to the next. Therefore e.g. counters can be realized which may not forget their counter value. I.e. function blocks can have an internal memory.

Functions and function blocks differ in two essential points:

- A function block has multiple output parameters, a function only one.
  The output parameters of functions and function blocks differ syntactically.
- In contrast to a function, a function block can have an internal memory.

1. Function blocks with historical values (memory):

   For function blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

2. Function blocks without historical values (memory):

   For function blocks without historical values only one instance has to be defined for the FB type. This instance can be used for several calls of the FB (also with different I/O values).

The instance name can be defined without any restrictions. The type is preset and identical to the function block name.
Index

C
Components of the Ethernet library 2

E
ETH_AINIT S90 9
ETH_AREC S90 14
ETH_ASEND S90 17
ETH_ASTO S90 20
ETH_INFO S90 7
ETH_MODMAST S90 26
ETH_MODSTAT S90 23
Ethernet S90
  ETH_AINIT 9
  ETH_AREC 14
  ETH_ASEND 17
  ETH_ASTO 20
  ETH_INFO 7
  ETH_MODMAST 26
  ETH_MODSTAT 23