FAST Application Software


# Automation Building Blocks

L_ICIA_CommunicationInterface Function Blocks _ _ _ _

Reference Manual EN

Lenze

This manual applies to the Automation Building Blocks *L_ICIA_COMMUNICATIONINTERFACE Function Blocks*.



**Imprint**

Lenze SE

Hans-Lenze-Strasse 1, D-31855 Aerzen, Germany

Phone: +49 (0)5154 / 82-0

Fax: +49 (0)5154 / 82-2111

E-Mail: Lenze@Lenze.de

# Contents

# 1 Function Blocks

## 1.1 Document History

| Version | | | Desciption |
|---|---|---|---|
| 0.1 | 24/11/2025 | LSE | first edition |
| 0.2 | 04/12/2025 | LSE | update to simplified GSD identification method |
| 1.0 | 03/02/2026 | LSE | version V1.0 published |
| | | | |
| | | | |
| | | | |

## 1.2 About Automation Building Blocks

This manual describes a software solution for a partial task.

It is the user's responsibility to verify if the solution proposed by the software corresponds to his requirement. If necessary, the solution must be adapted. Physical aspects such as drive design are not part of this manual.

**Note:**
The terminal connection diagrams appearing in this manual show the wiring required to operate the software on a sample demo rig.

## 1.3 Conventions used

This manual uses the following conventions to distinguish between different types of information:

| Type of information | Highlighting | Example/notes |
|---|---|---|
| Spelling of numbers | | |
| Decimal separator | Point | The decimal point is always used.<br>For example: 1234.56 |
| Text | | |
| Program name | » « | »PLC Designer« ... |
| Variable names | *italic* | By setting *xEnable* to TRUE... |
| Function blocks | **bold** | The **L_MC1P_AxisBasicControl** function block ... |
| Function libraries | | The **L_TT1P_TechnologyModules** function library ... |
| Buttons | | ... and confirm by clicking on **Continue**. |
| Source code | Courier | `...`<br>`dwNumerator := 1;`<br>`dwDenominator := 1;`<br>`...` |
| Key words | **Courier bold** | ...starts with **FUNCTION** and ends with **END FUNCTION**. |
| Keyboard commands | **<bold>** | Press the **<F2>** key to request input assistance |
| | | If a shortcut is required for a command to be executed, a „+“ separates the commands:<br>Press the **<Shift>+<ESC>** key to ... |

**Variable Names**

The conventions used by Lenze for the variable names of Lenze system blocks, function blocks and functions are based on the "Hungarian Notation". This notation makes it possible to identify the most important properties (e.g. the data type) of the corresponding variable by means of its name, e.g. *xAxisEnabled*.

## 1.4 System Requirements

### Software

| Product | Type | Version |
|---|---|---|
| PLC Designer | | 4.1 or higher |

### Hardware

| Product | Type | Hardware Version | Firmware Version |
|---|---|---|---|
| i950 | I95AExxxF1AV10Z02R | not relevant | 1.14 or higher |
| PROFIBUS slot module | I9MAFP0000000S | | |

# 2 Function Blocks

The function blocks **L_ICIA_PROFIBUS_Base**, **L_ICIA_PROFIBUS_In** and **L_ICIA_PROFIBUS_Out** aim at replacement scenarios of the 9300 servo inverter series by Lenze's latest CbM/DbM systems such as the i950.

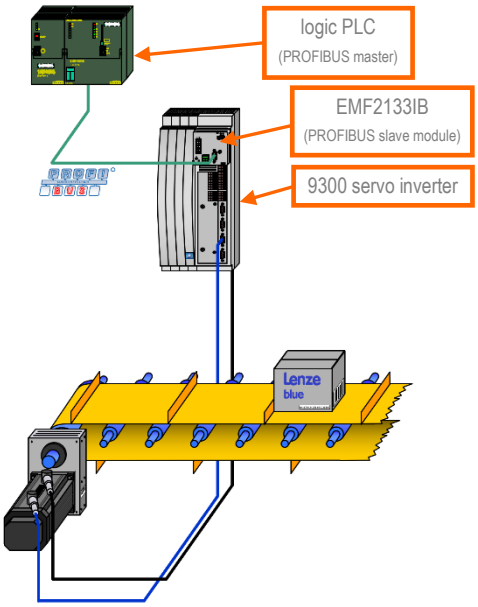



Illustration 1: 9300 servo inverter with PROFIBUS module[1]

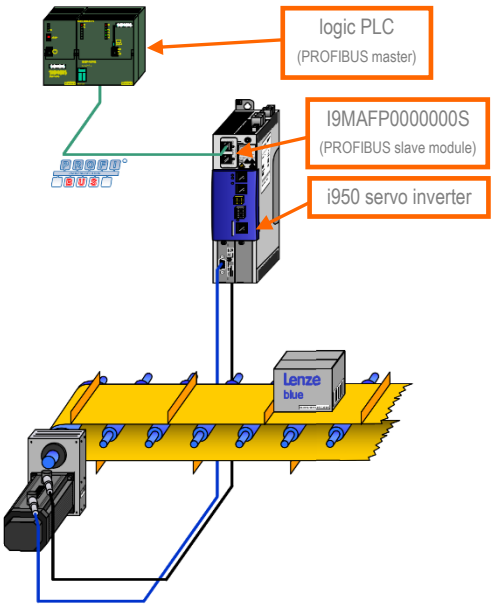



Illustration 2: i950 servo inverter with PROFIBUS slot module

**Note:**

In many cases, a one-to-one replacement might be required, not touching the logic PLC's program.

[1] For Lenze devices such as 9300, two AIF modules for PROFIBUS were available:
1 = EMF2133IB with an extended scope of GSD/GSE configurations (see chapter 4.1)
2 = EMF2131IB with a basic scope of GSD/GSE configurations (only Drivecom drive profile with 1 … 4 process data)

Three function blocks exist, splitting the AIF fieldbus communication into three sub-functions:

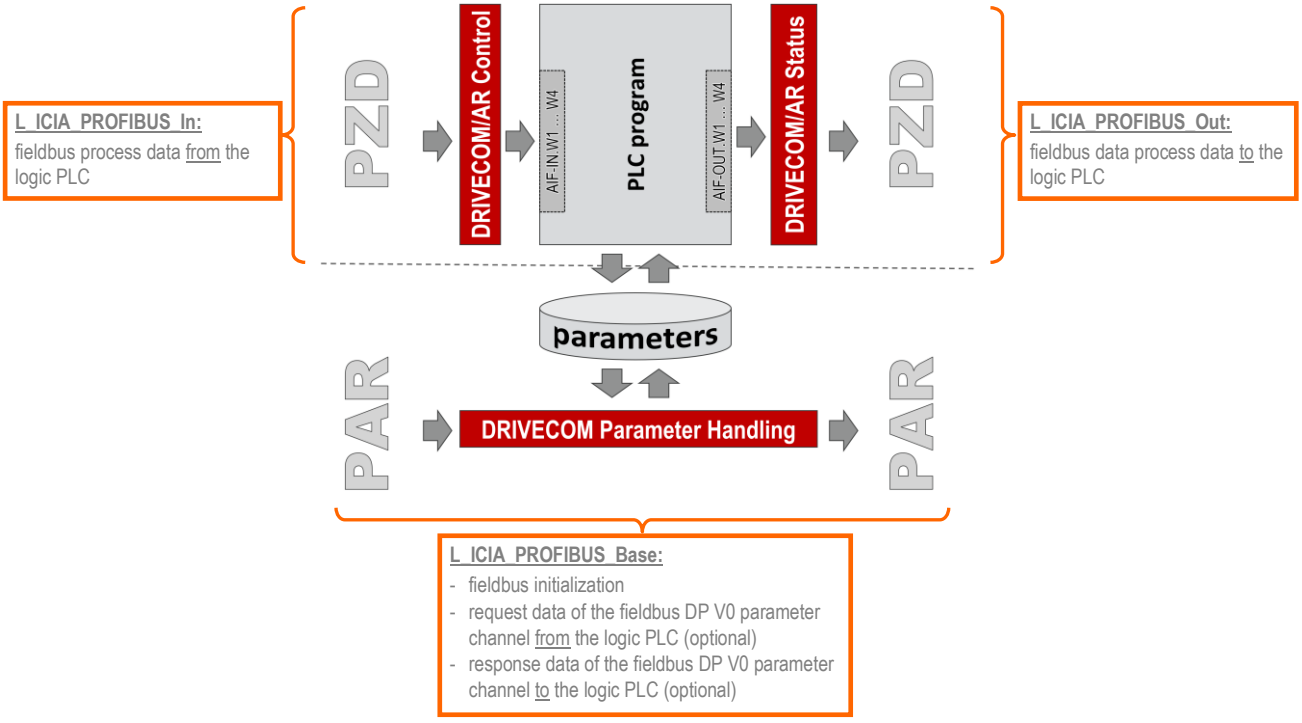



Illustration 3: overview on the process data communication (PDO) and parameter channel communication (SDO)

1. A function block **L_ICIA_PROFIBUS_In** to extract the process data control information from the raw data received on PROFIBUS. The method provides the process data in the form of the **AIF_IN** format of the 9300 servo inverter (see chapters 2.2 and 3.2).
2. A function block **L_ICIA_PROFIBUS_Out** to compile the complete fieldbus telegram to be transmitted to the logic PLC. The method reads the process data control information in the form of the **AIF_Out** format of the 9300 servo inverter (see chapters 2.3 and 3.2).
3. The function block **L_ICIA_PROFIBUS_Base**, processing the communication set-up and the DP V0 parameter channel (see chapter 2.1).

**Note:**

Always declare and call the PROFIBUS function blocks in the following order:

- **L_ICIA_PROFIBUS_Base**
- **L_ICIA_PROFIBUS_In**
- **L_ICIA_PROFIBUS_Out**

## 2.1 Function Block *L_ICIA_PROFIBUS_Base*

The function block **L_ICIA_PROFIBUS_Base** must always be integrated in the PLC project for correct initialization of the GSD/GSE configuration. If configured, the block additionally processes the DP V0 parameter channel if selected.

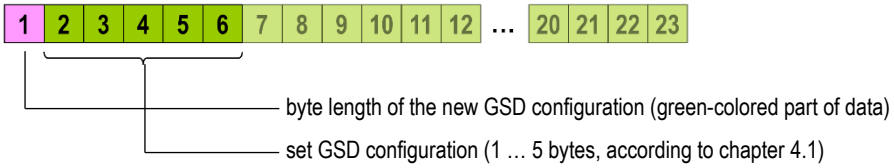
### 2.1.1 Configuration Mode Selection (GSD/GSE Configuration)

The configuration of the PROFIBUS communication for each slave is defined with the help of the GSD/GSE file. Typically, the scope of PROFIBUS Tx/Rx data is pre-defined during the programming the logic PLC. One out of a pool of possible configurations (see chapter 4.1) determines the structure of the PROFIBUS telegram to a slave device.

During the initialization of PROFIBUS communication, the function block **L_ICIA_PROFIBUS_Base** identifies the selected GSD/GSE configuration as listed in the appendix, chapter 4.1. For this purpose, an internal service[2] reads the received GSD configuration.

**Set GSD/GSE Configuration** (BYTE ARRAY[23])

The requested GSD configuration is polled by the function block **L_ICIA_PROFIBUS_Base** continuously to check for a new GSD/GSE configuration:




The active GSD configuration is displayed in index 0x2348:003.

[2]

## 2.1.2 Parameter Handling

Parameter communication must be selected via the GSD/GSE configuration from the logic PLC and is included in the complete Rx/Tx PROFIBUS data. On receiving a valid GSD/GSE configuration (*xInit* = FALSE), an optional parameter channel (DRIVECOM DP V0) is initialized, if included in the GSD/GSE configuration.

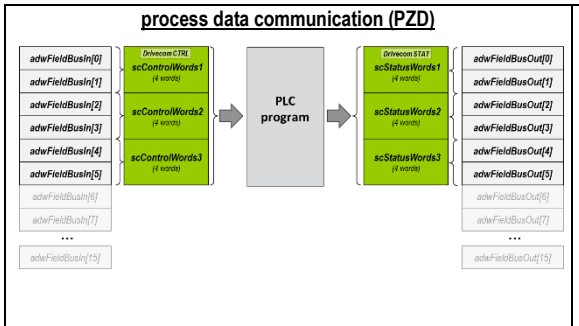



Illustration 4: no parameter channel (PZD only)

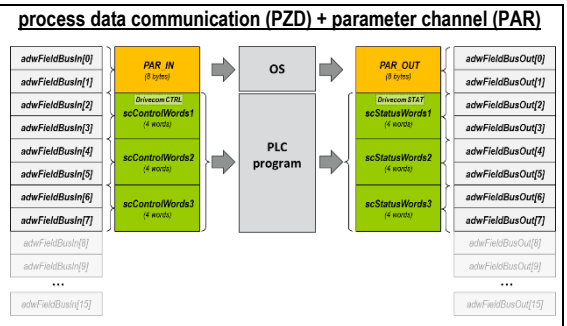



Illustration 5: with parameter channel (PZD + PAR)

In case of a parameter channel configured, the lowest 8 bytes of the raw data received on *L_ICIA_PROFIBUS_In.adwFieldBusIn* / transmit on *L_ICIA_PROFIBUS_Out.adwFieldBusOut* are interpreted as shown in the appendix, chapters 4.6 and 4.7.

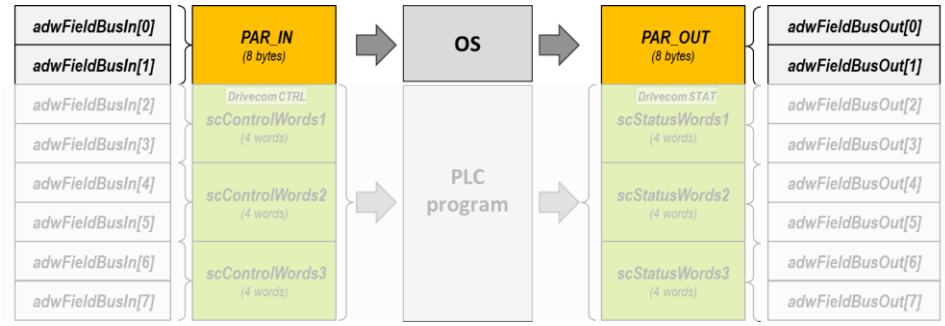
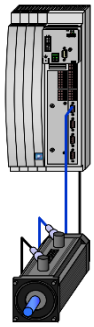



Illustration 6: DP V0 parameter channel data as a part of the complete PROFIBUS telegram

In migration scenarios, the superposed logic PLC might address codes/sub-codes of the Lenze GDC[3] devices 8200/9300 (legacy devices), not available on the i950 servo inverter.

[3] GDC = **G**lobal **D**rive **C**ontrol, one of the most successful inverter/servo inverter series of Lenze

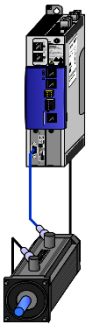
**Example:**



**actual motor speed:**

| | |
|---|---|
| code number: | 51 |
| sub-code number: | 0 |
| access: | read-only |
| unit: | [rpm] |
| size: | 4 byte |
| scaling factor: | 10000 : 1 (FIX32[4]) |

Illustration 7: 9300 servo inverter (legacy product)



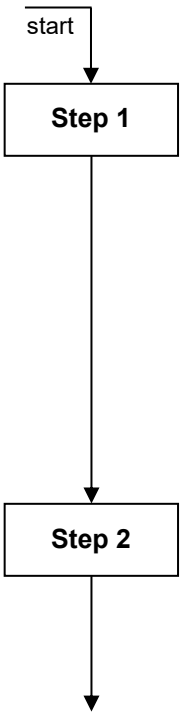
**actual motor speed:**

| | |
|---|---|
| index number: | 0x606C |
| sub-index number: | 0 |
| access: | read-only |
| unit: | [rpm] |
| size: | 4 byte |
| scaling factor: | $2^{31}$ : 480000 ('_s'[5]) |

Illustration 8: i950 servo inverter (actual product)

Because of the mismatch between the old GDC device's code list and the current i950's index list, an internal re-mapping of codes/sub-codes to current (user-) indexes is required. The correspondence between old GDC codes and the i950 indexes is defined via a mapping table on the input/output *ascParReference*.

The generation of the mapping list must be done by the user in the following way:




Define the number of objects in your mapping list: The amount of objects defines on how many different codes/indexes you would like to get access to.

Example: The logic PLC uses the DP V0 parameter channel to read the following codes from a 9300 servo inverter:

- C0051/000: actual motor speed, scaled in [rpm]
- C0053/000: actual DC bus voltage, scaled in [V]
- C0063/000: actual motor temperature, scaled in [°C]

In this example, the mapping list requires three entries.

In the i950's PLC program, declare a data array with a corresponding number of entries of type *L_ICIA_sc93ParReference* as follows:

```
ascParReference:        ARRAY[1..3] OF L_ICIA_sc93ParReference;              // mapping table
```

[4] The FIX32 format uses a 4-byte data size and a scaling factor of 10000, meaning a value of 10000 represents a physical value of 1.0000[rpm]. This format is also known as the '_e4' format in the Lenze terminology.
[5] The '_s' scaling was introduced with the Lenze 9400 series and scales the motor speed as a 32-bit value. A raw value of $2^{31}$ represents a physical value of 480000[rpm].

**Step 3**

Extend the declaration of step 2 by assigning initialization values to the mapping table's data array:

```
ascParReference:        ARRAY[1..4] OF L_ICIA_sc93ParReference  := [          // mapping table
 (wCode:=11, wSubCode:=0, wIndex:=16#5500, bySubIndex:=1, bySize:=8, diNum:=10000,   diDen:=1),
 (wCode:=51, wSubCode:=0, wIndex:=16#606C, bySubIndex:=0, bySize:=4, diNum:=1171875, diDen:=524288),
 (wCode:=53, wSubCode:=0, wIndex:=16#6079, bySubIndex:=0, bySize:=4, diNum:=10,      diDen:=1),
 (wCode:=63, wSubCode:=0, wIndex:=16#2D49, bySubIndex:=5, bySize:=2, diNum:=1000,    diDen:=1)];
```

**How to find the values for the numerator/denominator ratio?**

The numerator/denominator scales a raw value of the physical Lenze device to the raw value of the legacy Lenze device.

Example: The numerator/denominator value for the actual motor speed results from the scaling ratio of the i950 servo inverter (0x606C:000) and the Lenze legacy product (C0051/000):

scaling ratio of the physical Lenze device (i950)
You can look-up the index scaling factors in the parameter list's tool tips of »EASY Starter«.

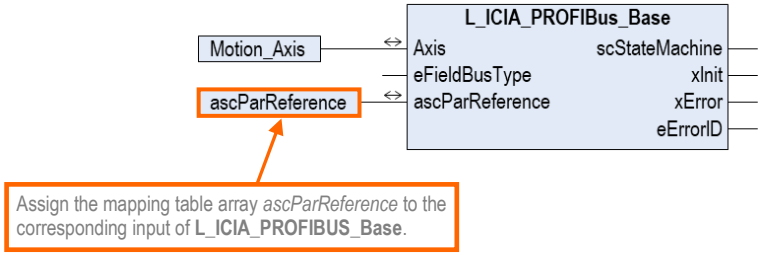
$$\frac{diNum}{diDen} = \frac{480000[rpm]}{2^{31}} \cdot \frac{10000}{1} = \frac{4800000000}{2147483648} = \frac{1171875}{524288}$$

scaling ratio of the legacy Lenze device (9300)
You can look-up the code scaling factors in the 9300's reference manual in the attribute table.

*) use greatest common denominator calculations to shorten the numbers for the numerator/denominator

**Step 4**

Assign the mapping table's data array to the function block **L_ICIA_PROFIBUS_Base**:




**Tip:**

Call the function block **L_ICIA_PROFIBUS_Base** in a freewheeling task with low priority to unload the high-priority motion task.

Still, the function blocks **L_ICIA_PROFIBUS_In** and **L_ICIA_PROFIBUS_Out** for process data may be called in the high-priority motion task.

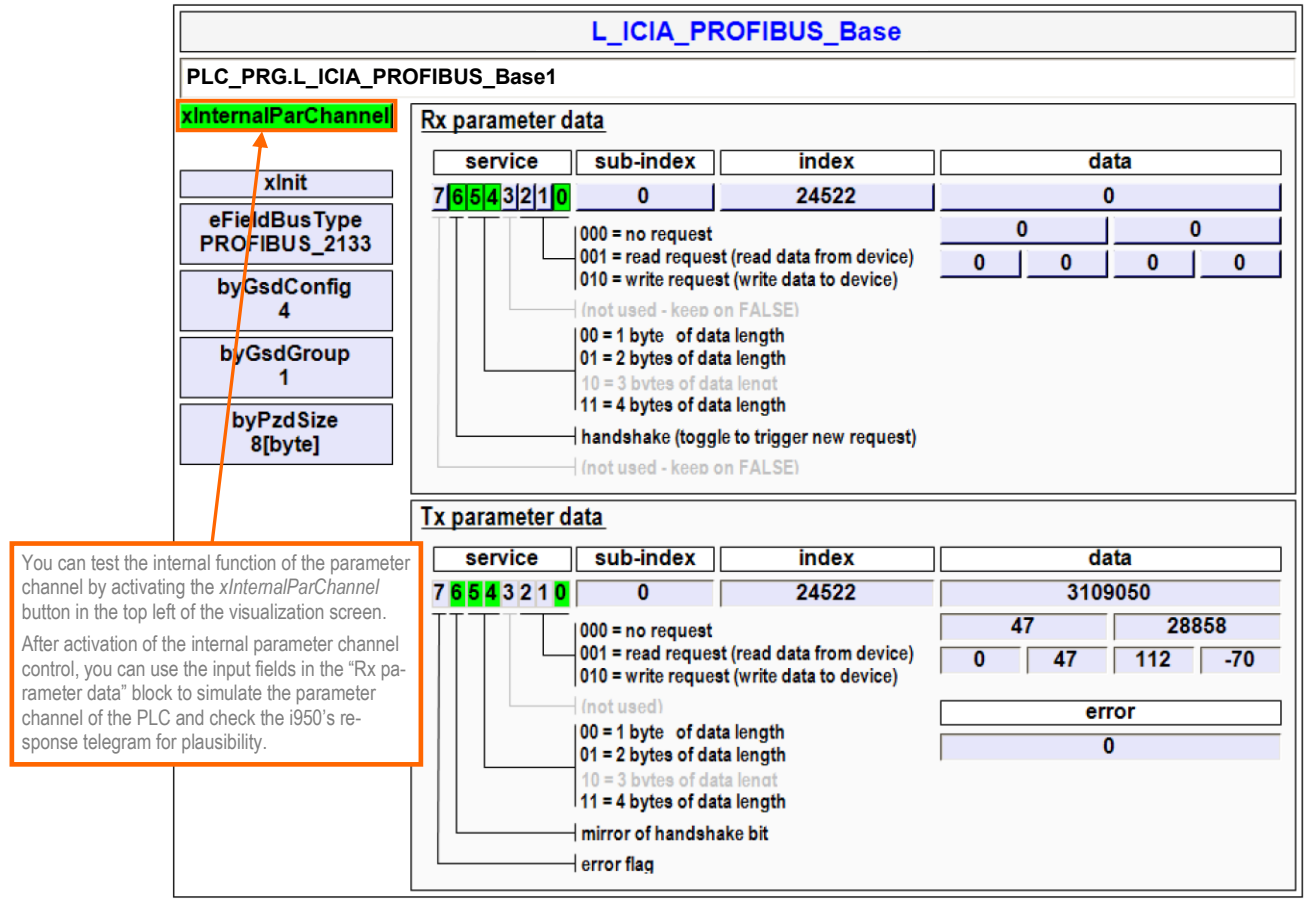
end

**Note:**

In contrary to the 9300 series, i950 allows parameters using a floating-point data type (*LREAL*) with 8 bytes data size. Even this i950 parameter type can be handled by the function block **L_ICIA_PROFIBUS_Base**.

To receive a data value with four decimal remainder digits on your machine PLC, apply a numerator/denominator scaling of *diNum* = 10000 and *diDen* = 1 in the parameter reference list.

To monitor the parameter channel, the function block **L_ICIA_PROFIBUS_Base** includes a built-in visualization screen:



**How to find out about which parameters/indexes are accessed by the logic PLC**

Sometimes the logic PLC program is not available. In this case the parameters accessed by the logic PLCs are not known in beforehand and cannot be considered in the reference list.

An easy way to get an overview is to trace the parameter request telegrams of the DP V0 parameter channel. Proceed as follows:

- In the PLC project of the i950 drive, insert a new trace in »PLC Designer«.
- Add the following variables of the parameter channel Rx/Tx telegrams to the trace:
  - Rx index (*L_ICIA_PROFIBUS_Base1.RxParData.wIndex*)
  - Rx sub-index (*L_ICIA_PROFIBUS_Base1.RxParData.bySubIndex*)
- Start the trace while the logic PLC tries to access the drive parameters via the DP V0 parameter channel.
- Activate a measuring cursor in the trace: The values measured on the Rx index and Rx sub-index indicate the drive parameters which the logic PLC tries to access.

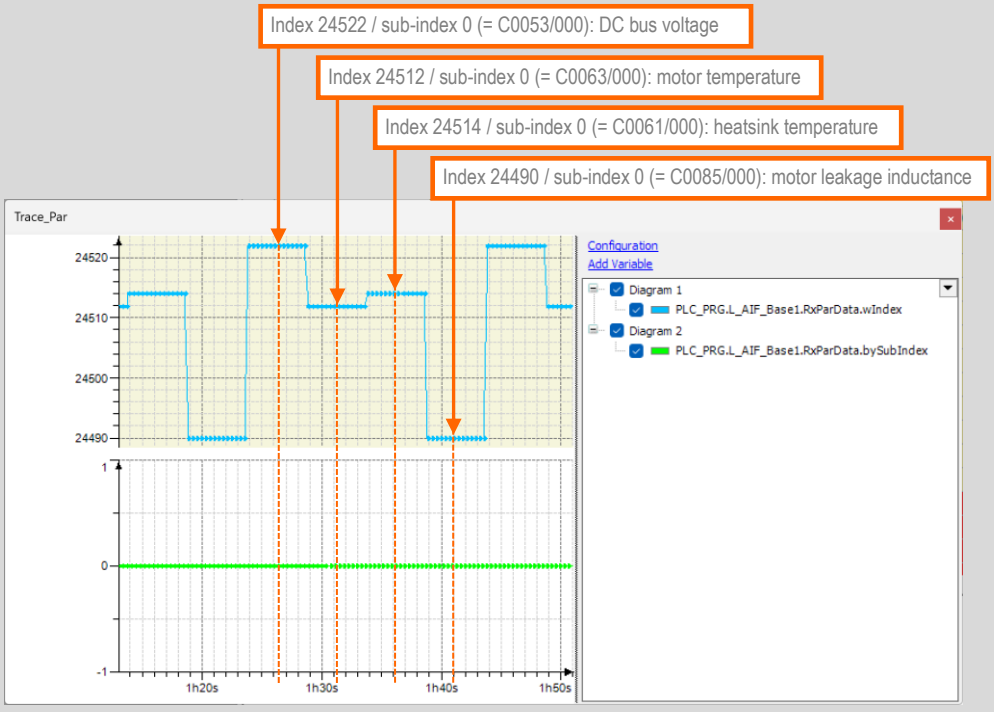



Illustration 10: example for a trace monitoring the parameter access of the PLC (SDO)

### 2.1.3 Incompatibility List

The following functions are not implemented in the function block **L_ICIA_PROFIBUS_Base**:

- No PROFIsafe protocol on i950 PROFIBUS is supported.
- Parameter/index numbers do not match between 9300 and i950. Apply a correspondence list as a reference between the parameters of a Lenze legacy device and an i950 drive controller as shown in the previous chapter 2.1.2.

## 2.1.4 Interface

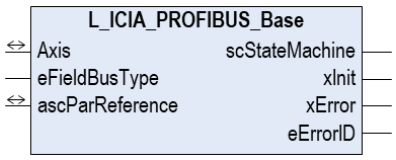



Illustration 11: interface of function block **L_ICIA_PROFIBUS_Base**

## 2.1.5 Task Information

| Call-up possible from: | ☒ freewheeling task | ☒ time-controlled task (INTERVAL) | ☐ event-controlled task (EVENT) | ☐ interrupt task |
|---|---|---|---|---|

**Note:**
Make sure to have included the CAA Memory library in your PLC project to get a fault-free built of your code.

## 2.1.6 Inputs and Outputs

| Identifier<br>Data type | Description |
|---|---|
| *Axis*<br>*AXIS_REF* | reference to the connected drive axis<br>In case of an i950 application, always assign the *Motion_Axis* to this signal. |
| *ascParReference*<br>*ARRAY [*] OF L_ICIA_sc93ParReference* | parameter correspondence list<br>This list defines the correspondence between 9300 codes and i950 indexes. As parameter values are stored in indexes, which have different numbers on 9300 and i950, the list allows to …<br>• … link a 9300 code to an i950 index<br>• … consider a scaling numerator/denominator factor between the 9300 parameter value and the i950 index value<br>Find a detailed overview of the structure *ascParReference* in chapter 2.1.7 (next page). |

## 2.1.7 Inputs

| Identifier<br>Data type | Description |
|---|---|
| *eFieldBusType*<br>*L_ICIA_eFieldBusType* | type of fieldbus<br>In the default, this signal is set to 1 ('PROFIBUS_2133')<br>So far, this variable is not used in the function block **L_ICIA_PROFIBUS_Base**, as the PROFIBUS function blocks only support PROFIBUS communication.<br>**Note:** In future, the input may allow to support various fieldbus systems Lenze offered on the 9300 series such as CAN, INTERBUS, … |

**User-Defined Variable Structure *L_ICIA_sc93ParReference***

The structure serves to define the parameter correspondence on the PROFIBUS level and the i950 level. The following elements are part of this variable structure:

| Identifier<br>Data type | Description |
|---|---|
| *wCode*<br>*WORD* | code number of the 9300 servo drive<br>**Note:** The 9300 code number results from subtracting the index value (byte 3 and 4 of the parameter channel) from a fixed value of 24575 (0x5FFF). |
| *bySubCode*<br>*BYTE* | sub-code number of the 9300 servo drive |
| *wIndex*<br>*WORD* | corresponding index number of the i950 servo drive |
| *bySubIndex*<br>*BYTE* | sub-index number of the i950 servo drive |
| *bySize*<br>*BYTE* | data size of the i950's index value<br>**Note:** This information is required, as the data size of the i950's index value does not necessarily match the data size of the 9300 code value. |
| *diNum*<br>*diDen*<br>*BYTE* | scaling factor between the 9300 code value and the i950 index value (split to numerator/denominator values)<br>The values for the scaling numerator/denominator can be obtained as shown on the next page. More details can be found in chapter 2.1.2. |

**Note:**
An application example is given in the appendix in chapter 2.1.2.

**Example:** calculation of scaling numerator/denominator values:

The actual motor speed 0x606C:000 of the i950 servo drive should be read via parameter channel and be returned to the PLC in the 9300 format of code C0051/000:

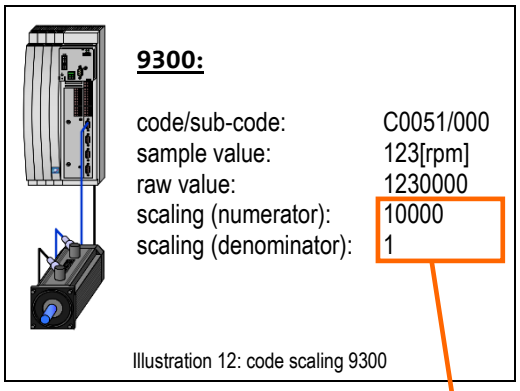



Illustration 12: code scaling 9300

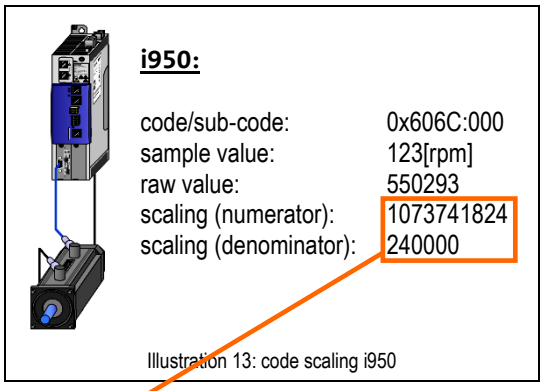



Illustration 13: code scaling i950

From the above example, the total numerator/denominator values *diNum* and *diDen* can be calculated:

$$\frac{diNum}{diDen} = \frac{\frac{10000}{1}}{\frac{1073741824}{240000}} = \frac{2400000000}{1073741824} = \frac{1171875}{524288}$$

## 2.1.8 Outputs

| Identifier<br>Data type | Description | |
|---|---|---|
| *scStateMachine*<br>*L_ICIA_scStateMachine* | data of the communication state machine<br>This value must be connected to the corresponding input/output variables of the function blocks **L_ICIA_PROFIBUS_In** and **L_ICIA_PROFIBUS_Out** to ensure consistent operation of the PROFIBUS function blocks. A detailed description is given on the next page. | |
| *xInit*<br>*BOOL* | status signal: initialization of the GSD/GSE configuration ongoing | |
| | FALSE | GSD/GSE configuration has finished without errors |
| | TRUE | GSD/GSE configuration ongoing/has finished with errors |
| *xError*<br>*BOOL* | status signal: error during GSD/GSE configuration | |
| | FALSE | no error during GSD/GSE configuration. |
| | TRUE | An error occurred during GSD/GSE configuration:<br>• Initialization sequence cannot be terminated – status signal *xInit* remains on TRUE.<br>• Refer to the *wError* output for more information. |
| *eErrorID*<br>*WORD* | current error ID: | |
| | 0: | no error active |
| | 110: | GSD/GSE configuration could not be identified – please select a GSD/GSE configuration as listed in chapter.4.1 |
| | **Notes:**<br>- | |

### User-Defined Variable Structure *L_ICIA_scStateMachine*

This variable structure comprises general data of the fieldbus communication, depending on the fieldbus type active:

| Identifier<br>Data type | Description |
|---|---|
| *eFieldBusType*<br>*L_ICIA_eFieldBusType* | current error ID:<br>*PROFIBUS_2133*: The **L_ICIA_PROFIBUS** function blocks behave like the EMF2133IB PROFIBUS module for 9300.<br>**Notes:** So far, only '*PROFIBUS_2131*' and '*PROFIBUS_2133*' are supported. |
| *byGsdConfig*<br>*BYTE* | number of the active GSD/GSE configuration found during communication initialization<br>A complete overview on all possible GSD/GSE configurations is listed in chapter 4.1. |
| *byGsdGroup*<br>*BYTE* | group of the active GSD/GSE configuration of the active GSD/GSE configuration, set-up during communication initialization<br>A complete overview on all possible GSD/GSE configurations is listed in chapter 4.1. |
| *byGsdGroup*<br>*BYTE* | current error ID:<br>1: no parameter channel / process data (Drivecom control)<br>2: consistent Drivecom parameter channel / process data (Drivecom control)<br>3: consistent Drivecom parameter channel / consistent process data (Drivecom control)<br>4: Drivecom parameter channel / process data (Drivecom control)<br>5: Drivecom parameter channel / consistent process data (Drivecom control)<br>6: no parameter channel / consistent process data (Drivecom control)<br>7: no parameter channel / process data (Lenze device control)<br>8: consistent Drivecom parameter channel / process data (Lenze device control)<br>9: consistent Drivecom parameter channel / consistent process data (Lenze device control)<br>10: Drivecom parameter channel / process data (Lenze device control)<br>11: Drivecom parameter channel / consistent process data (Lenze device control)<br>12: no parameter channel / consistent process data (Lenze device control)<br>**Notes:** A complete overview on all possible GSD/GSE configurations is listed in chapter 4.1. |
| *byPzdSize*<br>*BYTE* | size of the process data (PZD), scaled in [byte] |
| *wDrivecomCtrl*<br>*BYTE* | Drivecom control word<br>This variable is only used in a configuration with Drivecom process data communication (*byGsdGroup* = 1 … 6).<br>The meaning of each control bit of *wDrivecomCtrl* is explained in the appendix in chapter 4.4. |
| *wDrivecomStat*<br>*BYTE* | Drivecom status word<br>This variable is only used in a configuration with Drivecom process data communication (*byGsdGroup* = 1 … 6).<br>The meaning of each control bit of *wDrivecomStat* is explained in the appendix in chapter 4.5. |
| *eDrivecomState*<br>*L_ICIA_eDrivecomState* | current state of the Drivecom state machine:<br>0: NOT_READY_TO_SWITCH_ON<br>32: SWITCH_ON_INHIBIT<br>1: READY_TO_SWITCH_ON<br>3: SWITCHED_ON<br>23: QUICK_STOP_ACTIVE<br>7: OPERATION_ENABLED<br>15: FAULT_REACTION_ACTIVE<br>8: FAULT<br>**Notes:** This variable is only used in a configuration with Drivecom process data communication (*byGsdGroup* = 1 … 6). The Drivecom state machine is shown in chapters 2.2.2 and 2.3.2. |
| *xInit*<br>*BOOL* | status signal: initialization of the GSD/GSE configuration ongoing<br>FALSE: GSD/GSE configuration has finished without errors<br>TRUE: GSD/GSE configuration ongoing/has finished with errors<br>**Note:** This signal mirrors the output signal *L_ICIA_PROFIBUS_Base.xInit*. |
| *xError*<br>*BOOL* | status signal: error during GSD/GSE configuration<br>FALSE: no error during GSD/GSE configuration.<br>TRUE: An error occurred during GSD/GSE configuration:<br>• Initialization sequence cannot be terminated – status signal *xInit* remains on TRUE.<br>• Refer to the *L_ICIA_PROFIBUS_Base.wError* output for more information.<br>**Note:** This signal mirrors the output signal *L_ICIA_PROFIBUS_Base.xError*. |

| Identifier<br>Data type | Description | |
|---|---|---|
| *adwRawDataIn*<br>*ARRAY [0..15] OF DWORD* | raw input data on the fieldbus interface<br>The variable data array is a copy of the fieldbus raw input data in, received on the input *adwFieldBusIn* of function block **L_ICIA_PROFIBUS_In**. | |
| *adwRawDataOut*<br>*ARRAY [0..15] OF DWORD* | raw output data on the fieldbus interface<br>The variable data array is a copy of the fieldbus raw output data in, generated on the output *adwFieldBusOut* of function block **L_ICIA_PROFIBUS_Out**. | |
| *AxisState*<br>*MC_ReadAxisInfo* | This structure includes important status signals of the i950 drive: | |
| | *LimitSwitchPos*: | positive limit switch has triggered (i.e. on *L_TF2P_SpeedControlBase1.scCtrl-BasicMotion.xHWLimitPos*) |
| | *LimitSwitchNeg*: | negative limit switch has triggered (i.e. on *L_TF2P_SpeedControlBase1.scCtrl-BasicMotion.xHWLimitNeg*) |
| | *Simulation*: | axis is operated in the virtual mode<br>On an i950 axis, this signal is always FALSE. |
| | *CommunicationReady*: | motion bus communication interface between axis driver (*AXIS_REF*) and motor control is in operation<br>On an i950 axis, this signal is always TRUE. |
| | *ReadyForPowerOn*: | drive is ready for being powered on (i.e. via control signal *L_TF2P_SpeedControlBase1.xEnableOperation*).<br>This signal state comprises the following states:<br>• drive is fault-free<br>• no STO command is active (safe torque off)<br>• DC bus voltage is switched on |
| | *PowerOn*: | i950 drive is powered on (same status as *L_TF2P_SpeedControlBase1.xOperationEnabled*) |
| | *IsHomed*: | zero position of the i950 drive's measuring system is known |
| | *AxisError*: | error in the axis driver (*AXIS_REF*) |
| | *AxisWarning*: | warning in the axis driver (*AXIS_REF*) |
| | *DriveError*: | error in the inverter's motor control |
| | *DriveWarning*: | warning in the inverter's motor control |
| | *SWLimitSwitchPos*: | positive software limit has triggered |
| | *SWLimitSwitchNeg*: | negative software limit has triggered |
| | *ReadyForMotion*: | drive is ready for receiving motion commands<br>This signal state comprises the following states:<br>• drive is enabled<br>• drive is fault-free<br>• a motor brake (if available) has opened |
| | *STOActive*: | STO command is active (safe torque off) |
| | *VoltageEnabled*: | DC bus voltage is switched on |
| | *MotorMagnetised*: | motor is magnetization complete |
| | *QSPApplActive*: | quickstop command of the axis driver (*AXIS_REF*) is active |
| | *QSPDriveActive*: | quickstop command of the inverter's motor control is active |
| | Find more information about **MC_ReadAxisInfo** in the »PLC Designer« online help. | |

**Caution:**

The above-listed variables are read-only! Never change any of these variables as this may have unpredictable consequences in fieldbus communication and drive behaviour!

## 2.2 Function Block *L_ICIA_PROFIBUS_In*

The function block **L_ICIA_PROFIBUS_In** reads 16 double words of the fieldbus input data on the input data array *adwFieldBusIn*. Once a valid GSD/GSE configuration was detected (*scStateMachine.xInit* = FALSE), the raw data on the input signal *adwFieldBusIn* of the function block **L_ICIA_PROFIBUS_In** are mapped to ...

- process data PZD
- parameter data PAR (optional, if selected, see chapter 2.1.2)

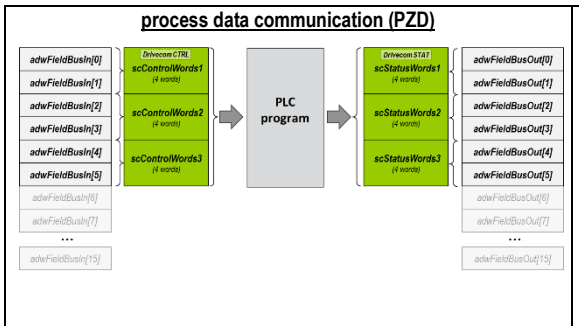



Illustration 14: no parameter channel (PZD only)

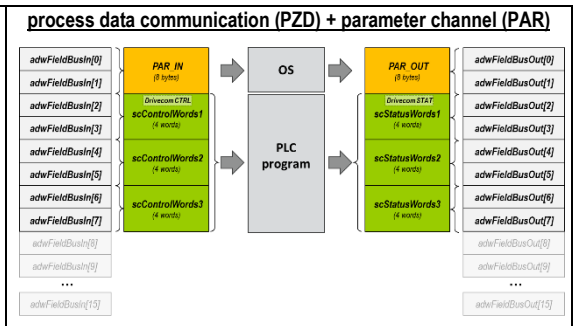



Illustration 15: with parameter channel (PZD + PAR)

**Note:**

The i950 PROFIBUS slot module handles up to 16 double words of input data. The function block **L_ICIA_PROFIBUS_In** only processes the double words 0 to 7. The double word 8 to 15 are not considered in the evaluation of the fieldbus raw data.

However, always assign a data array *ARRAY [0..15] OF DWORD* to the input signal *L_ICIA_PROFIBUS_In.adwFieldBusIn*.

## 2.2.1 Process Data (PZD)

In any case, process data exchange is part of the fieldbus communication. The function block **L_ICIA_PROFIBUS_In** handles the process input data of the fieldbus system and converts the raw data received on *adwFieldBusIn* to the data structures known from the 8200/9300 device series.

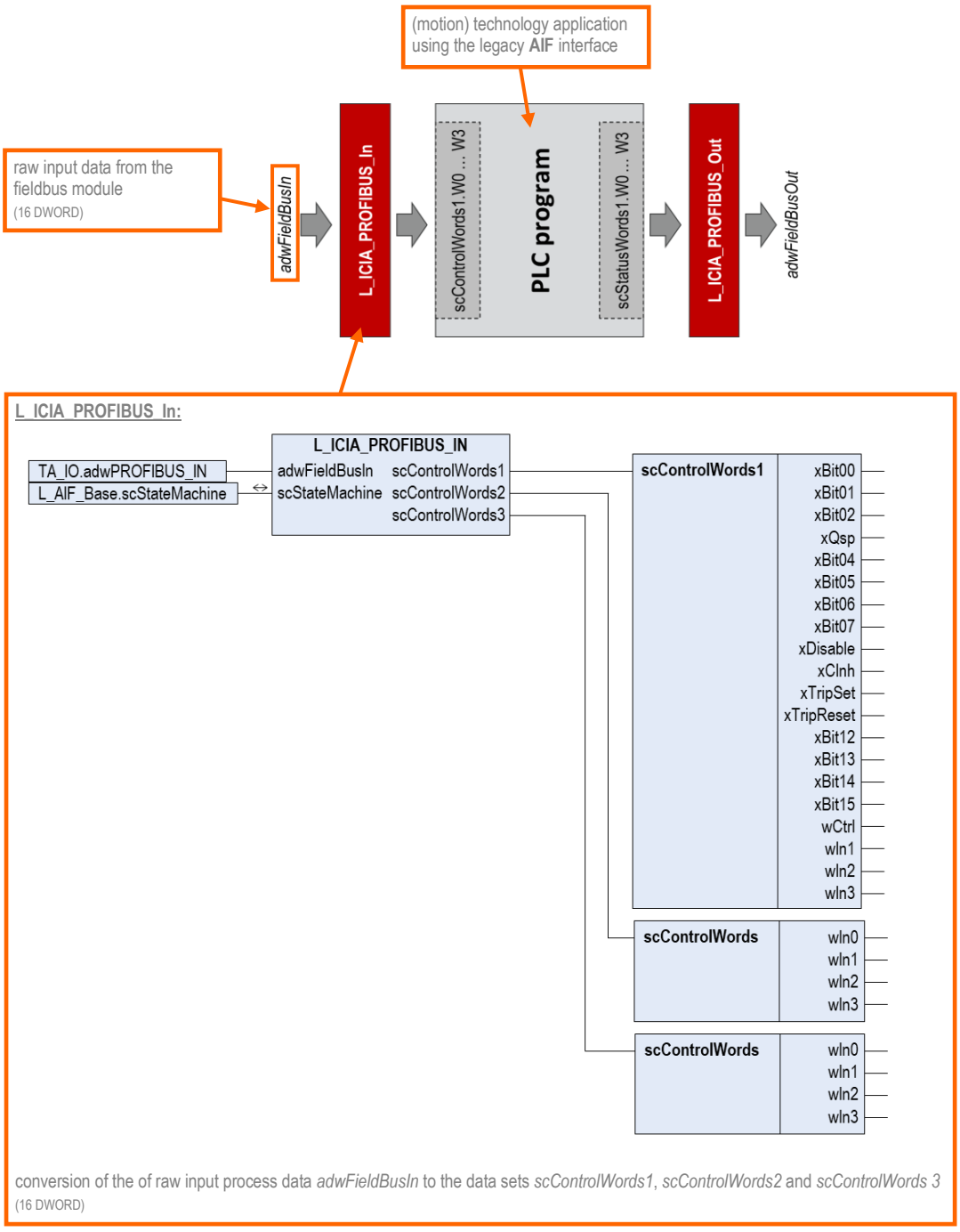



Illustration 16: principle of process input data handling / detailed signal list of the *scControlWords* interfaces of function block **L_ICIA_PROFIBUS_In**

## 2.2.2 Drivecom State Machine

Depending on the GSD/GSE configuration, the first process input data word *scControlWords1.wCtrl* is processed via the Drivecom state machine:

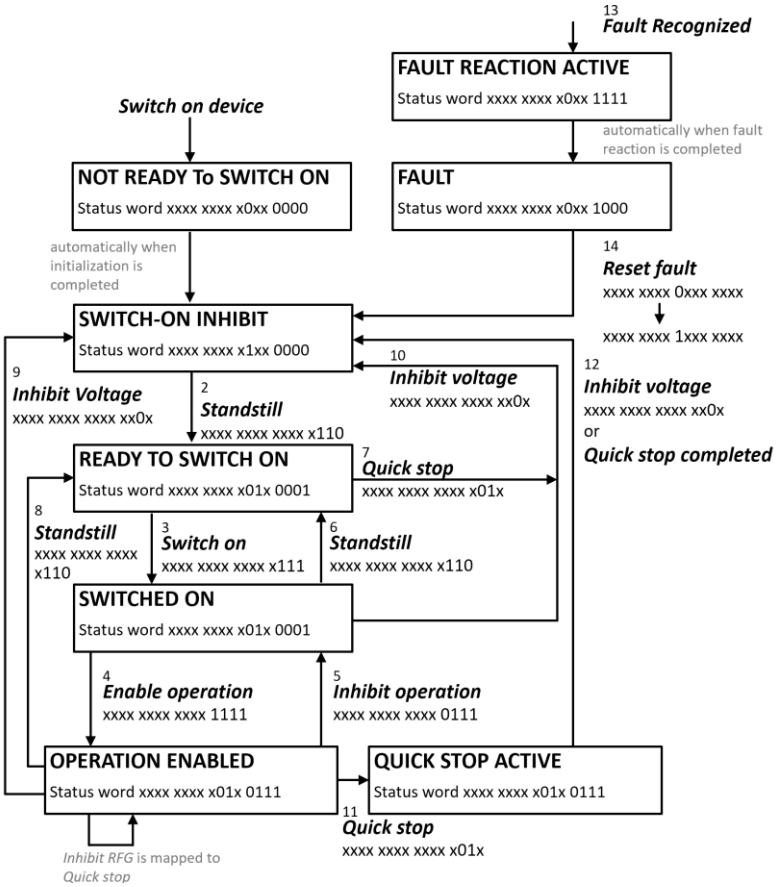



Illustration 17: flow chart of the Drivecom state machine (affecting control/status word 1)

The actual state of the Drivecom state machine is displayed on the variable *scStateMachine.eDrivecomState*.

### 2.2.3 Incompatibility List

The following functions are not implemented in the function block **L_ICIA_PROFIBUS_In**:

- The output *scControlWords1.wCtrl.xTripSet* does not find a corresponding function in the FAST technology modules. The user can evaluate this signal to set a user-defined error.
- Facing an undervoltage state during drive operation leads to an error, as the PLCopen state machine is violated. On 9300 an undervoltage state during drive operating was resulting in a message only.
- The STO command of i950 must be released to achieve the same behavior of the Drivecom state machine as on 9300. If the i950's STO command is active, the Drivecom state machine remains in the state *Switch-On Inhibited*.
  Using GSD configurations with Lenze device control (AR), the STO command keeps the *xDisable* control signal active, meaning the drive cannot be activated.
- The **L_ICIA_PROFIBUS_In** function block supports the following device control methods:
  - Drivecom
  - Lenze device control (AR)

  The PROFIdrive control method is not supported.

## 2.2.4 Interface

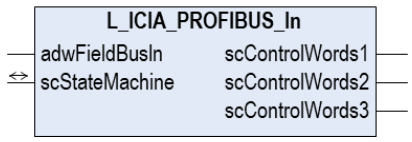



Illustration 18: interface of function block **L_ICIA_PROFIBUS_In**

## 2.2.5 Task Information

| **Call-up possible from:** | ☒ freewheeling task | ☒ time-controlled task (INTERVAL) | ☐ event-controlled task (EVENT) | ☐ interrupt task |
|---|---|---|---|---|

**Note:**
Make sure to have included the *CAA Memory* library in your PLC project to get a fault-free built of your code.

## 2.2.6 Inputs and Outputs

| **Identifier**<br>**Data type** | **Description** |
|---|---|
| *scStateMachine*<br>*L_ICIA_scStateMachine* | data of the communication state machine<br>Connect the corresponding output *scStateMachine* of function block **L_ICIA_PROFIBUS_Base** to ensure consistent operation of the PROFIBUS function blocks. A detailed description of this variable structure is given in chapter 2.1.8. |

## 2.2.7 Inputs

| **Identifier**<br>**Data type** | **Description** |
|---|---|
| *adwFieldBusIn*<br>*ARRAY [0..15] OF DWORD* | input of the fieldbus raw data<br>These values can directly be mapped to the input data of the fieldbus IO interface. |

## 2.2.8 Outputs

| **Identifier**<br>**Data type** | **Description** |
|---|---|
| *scControlWords1*<br>*L_ICIA_scControlWords1* | AIF fieldbus input data (first group)<br>The values comprise a four-word data structure, following the structure of the AIF-IN system block of the 9300 servo inverter. A detailed description is given on the next page. |
| *scControlWords2*<br>*L_ICIA_scControlWords* | AIF fieldbus input data (second group)<br>The values comprise a four-word data structure, following the structure of the AIF-IN system block of the 9300 ServoPLC inverter. A detailed description is given on the next pages. |
| *scControlWords3*<br>*L_ICIA_scControlWords* | AIF fieldbus input data (third group)<br>The values comprise a four-word data structure, following the structure of the AIF-IN system block of the 9300 ServoPLC inverter. A detailed description is given on the next pages. |

**User-Defined Variable Structure *L_ICIA_scControlWords1***

This structure implements the AIF-IN interface known from the 9300 servo inverter series. It includes the following elements:

| Identifier<br>Data type | Description |
|---|---|
| *xBit00*<br>*BIT* | bit 0 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit01*<br>*BIT* | bit 1 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit02*<br>*BIT* | bit 2 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xQsp*<br>*BIT* | bit 3 of the control word: activate quick stop<br>FALSE: quick stop not activated<br>TRUE: quick stop activated<br>**Note:** This bit must be connected to a quick stop command in the application (i.e. implemented by the function blocks **MC_Stop**, **L_MC1P_SetQuickStopAppl**, …). |
| *xBit04*<br>*BIT* | bit 4 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit05*<br>*BIT* | bit 5 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit06*<br>*BIT* | bit 6 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit07*<br>*BIT* | bit 7 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xDisable*<br>*BIT* | bit 08 of the control word: disable the drive<br>FALSE: do not disable drive (*xCInh*=FALSE leads to power-up the drive)<br>TRUE: disable drive (*xCInh*=FALSE has no effect)<br>**Notes:**<br>- Use this bit to interlock enabling the drive's operation. On *xDisable*=TRUE, the drive must remain shut-down, even if *xCInh* is on FALSE.<br>- If *xDisable* is on TRUE, the 'drive ready' status remains on FALSE. |
| *xCInh*<br>*BIT* | bit 09 of the control word: inhibit the drive controller<br>FALSE: drive controller enabled<br>TRUE: drive controller inhibited<br>**Notes:**<br>- The bit is used to power-up the drive (i.e. by means of the **MC_Power** function block).<br>- If *xDisable* is on TRUE, the *xCInh* control bit has no effect. |

| Identifier<br>Data type | Description |
|---|---|
| *xTripSet*<br>*BIT* | bit 10 of the control word: set a user error on the drive<br>FALSE: user error is triggered<br>TRUE: no user error is triggered<br>**Note:** As there is no corresponding function in the operating system of i950 available, the *xTripSet* bit has no practical meaning. |
| *xTripReset*<br>*BIT* | bit 11 of the control word: error reset command<br>FALSE=>TRUE reset error command<br>**Notes:**<br>- The bit is used to reset an error on the drive (i.e. by means of the **MC_Reset** function block).<br>- Resetting an error only works if the cause of the error does not apply any more. |
| *xBit12*<br>*BIT* | bit 12 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit13*<br>*BIT* | bit 13 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit14*<br>*BIT* | bit 14 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit15*<br>*BIT* | bit 15 of the control word<br>FALSE: control function deactivated<br>TRUE: control function activated<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *wCtrl*<br>*WORD* | Control word<br>This control word mirrors the 16 control bits as listed above in a WORD format. |
| *wIn1*<br>*WORD* | input of a 16 bit integer number<br>Typically, the second WORD on scControlWords1 is interpreted as the drive's speed set value, scaled in [%] (0 … 16384 = 0.0 … 100.0[%]). However, it is up to the user to define the meaning in the application. |
| *wIn2*<br>*WORD* | input of a free 16 bit WORD value |
| *wIn3*<br>*WORD* | input of a free 16 bit WORD value |

**Tip:**

Do you need to merge and *scControlWords1.wIn3* to a 32-bit value? The function **PackWordsToDword**[6] provides this function. Use it in the following way:

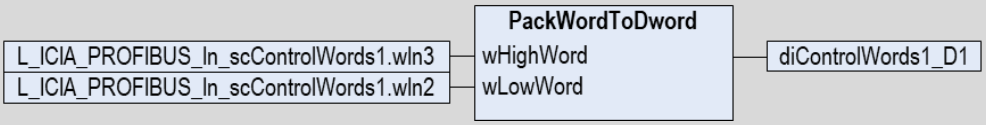



Illustration 19: conversion of two 16-bit *WORD* values to a 32-bit *DWORD* value

[6] included the *CAA Memory* library

**User-Defined Variable Structure *L_ICIA_scControlWords***

This structure implements the extended AIF-IN interface known from the 9300 ServoPLC inverter series. It is applied on the objects scControlWords2 and scControlWords3, and includes the following elements:

| Identifier<br>Data type | Description |
|---|---|
| *wIn0*<br>*WORD* | input of a free 16 bit WORD value |
| *wIn1*<br>*WORD* | input of a free 16 bit WORD value |
| *wIn2*<br>*WORD* | input of a free 16 bit WORD value |
| *wIn3*<br>*WORD* | input of a free 16 bit WORD value |

## 2.3 Function Block *L_ICIA_PROFIBUS_Out*

The function block **L_ICIA_PROFIBUS_Out** reads the AIF data structure known from the 8200/9300 device series and transfers its information to the 16 fieldbus output double-words on a data array. Once a valid GSD/GSE configuration was detected (*scStateMachine.xInit* = FALSE), the following data are mapped to the output data array *adwFieldBusOut*:

- process data PZD from the AIF-OUT objects
- parameter data PAR (optional, if selected, see chapter 2.1.2)

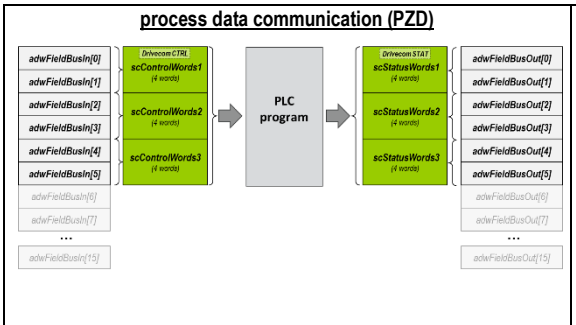



Illustration 20: no parameter channel (PZD only)

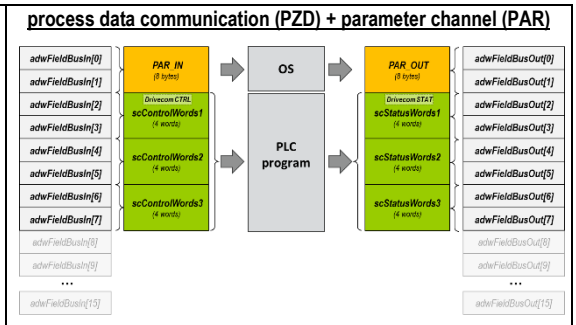



Illustration 21: with parameter channel (PZD + PAR)

**Note:**
The i950 PROFIBUS slot module handles up to 16 double words of output data. The output data range of the function block **L_ICIA_PROFIBUS_Out** (output *adwFieldBusOut*) comprises the full scope of 16 double words, even if only double words 0 to 7 are in use.

## 2.3.1 Process Data (PZD)

In any case, process data exchange is part of the fieldbus communication. The function block **L_ICIA_PROFIBUS_Out** generates the raw data on *adwFieldBusOut* from the AIF-OUT objects known from the 8200/9300 series.

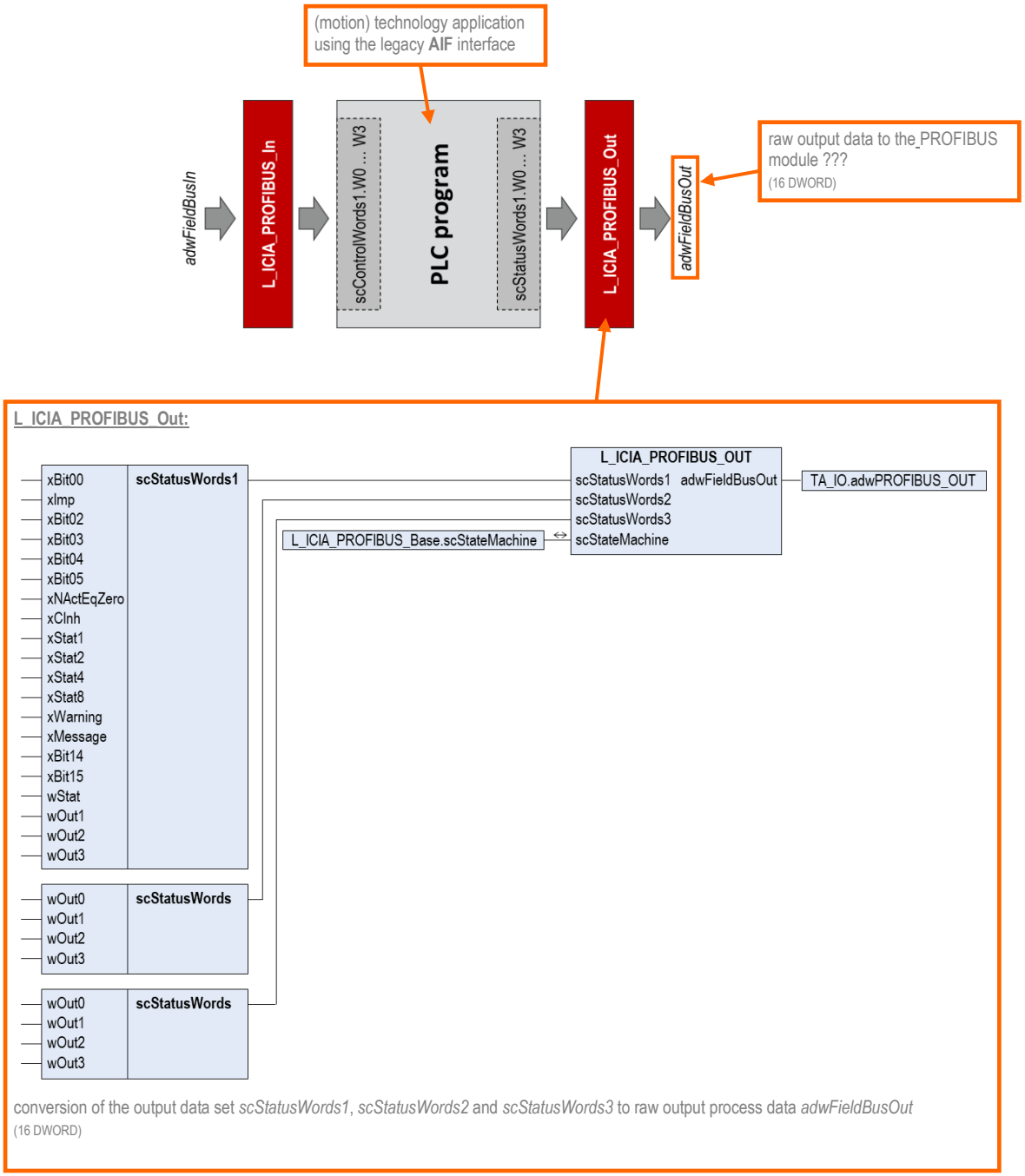



Illustration 22: principle of process output data handling / detailed signal list of the *scStatusWords* interfaces of function block **L_ICIA_PROFIBUS_In**

**Tip:**
Use the user-defined function block **L_STAT** to generate the status signals on *L_ICIA_PROFIBUS_Out.xStat1* ... *L_ICIA_PROFIBUS_Out.xStat8*.

## 2.3.2 Drivecom State Machine

Depending on the GSD/GSE configuration, the first process output data word *scStatusWords1.wStat* is processed via the Drivecom state machine:

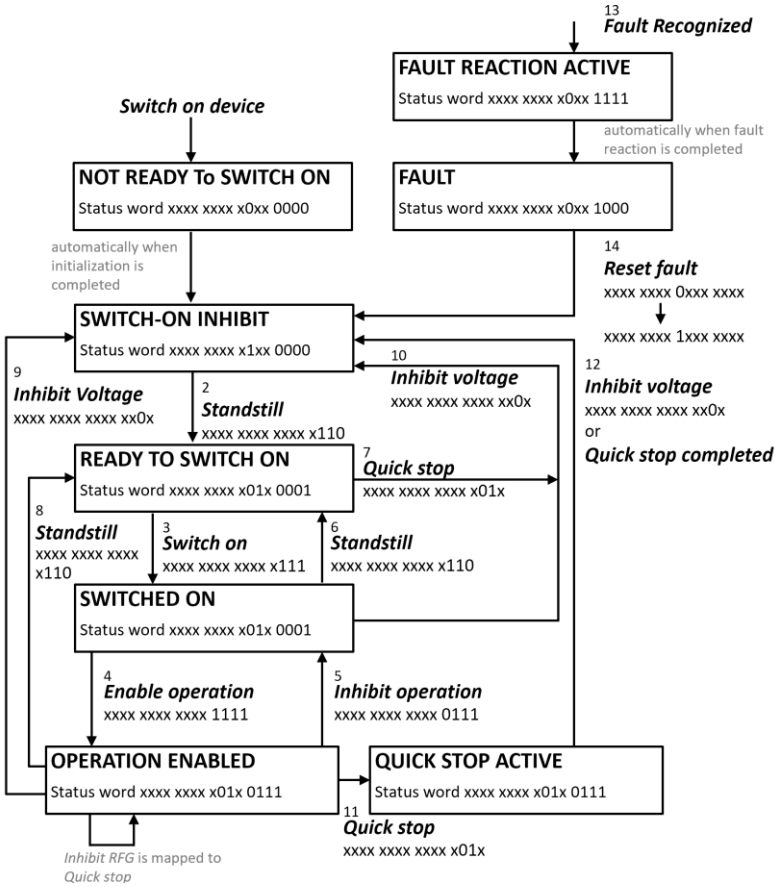



Illustration 23: flow chart of the Drivecom state machine (affecting control/status word 1)

The actual state of the Drivecom state machine is displayed on the variable *scStateMachine.eDrivecomState*.

### 2.3.3 Incompatibility List

The following functions are not implemented in the function block **L_ICIA_PROFIBUS_Out**:

- The status bits *DCTRL-STAT*1*, ... *DCTRL-STAT*8* do not comprise the full scope of 9300 states. The red-marked states are not supported:

| value | *DCTRL-STAT*8* | *DCTRL-STAT*4* | *DCTRL-STAT*2* | *DCTRL-STAT*1* | note |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | initialization after the supply voltage has been connected |
| 1 | 0 | 0 | 0 | 1 | lock mode, restart protection is active C0142 |
| 3 | 0 | 0 | 1 | 1 | drive is in controller inhibit mode |
| 4 | 0 | 1 | 0 | 0 | flying restart active |
| 5 | 0 | 1 | 0 | 1 | DC brake active |
| 6 | 0 | 1 | 1 | 0 | controller enabled |
| 7 | 0 | 1 | 1 | 1 | the release of a monitoring function resulted in a "message" |
| 8 | 1 | 0 | 0 | 0 | the release of a monitoring function resulted in a "trip" |
| 10 | 1 | 0 | 1 | 0 | the release of a monitoring function resulted in a "FAIL-QSP" |
| 15 | 1 | 1 | 1 | 1 | communication fail (PROFIBUS communication module ⇔ inverter) |

- According to PLCopen, an undervoltage on the DC bus results in an error instead of a message. Before restarting the drive, the user must reset the drive error.
- The **L_ICIA_PROFIBUS_In** function block supports the following device control methods:
  - Drivecom
  - Lenze device control (AR)

  The PROFIdrve control method is not supported.

## 2.3.4 Interface

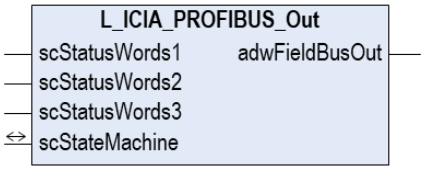



Illustration 24: interface of function block **L_ICIA_PROFIBUS_Out**

## 2.3.5 Task Information

| | | | | |
|---|---|---|---|---|
| **Call-up possible from:** | ☒ freewheeling task | ☒ time-controlled task (INTERVAL) | ☐ event-controlled task (EVENT) | ☐ interrupt task |

**Note:**
Make sure to have included the *CAA Memory* library in your PLC project to get a fault-free built of your code.

## 2.3.6 Inputs and Outputs

| Identifier<br>Data type | Description |
|---|---|
| *scStateMachine*<br>*L_ICIA_scStateMachine* | data of the communication state machine<br>Connect the corresponding output *scStateMachine* of function block **L_ICIA_PROFIBUS_Base** to ensure consistent operation of the AIF function blocks. A detailed description of this variable structure is given in chapter 2.1.8. |

## 2.3.7 Inputs

| Identifier<br>Data type | Description |
|---|---|
| *scStatusWords1*<br>*L_ICIA_scStatusWords1* | AIF fieldbus output data (first group)<br>The values comprise a four-word data structure, following the structure of the AIF-OUT system block of the 9300 servo inverter. A detailed description is given on the next page. |
| *scStatusWords2*<br>*L_ICIA_scStatusWords* | AIF fieldbus output data (second group)<br>The values comprise a four-word data structure, following the structure of the AIF-OUT system block of the 9300 ServoPLC inverter. A detailed description is given on the next pages. |
| *scStatusWords3*<br>*L_ICIA_scStatusWords* | AIF fieldbus output data (third group)<br>The values comprise a four-word data structure, following the structure of the AIF-OUT system block of the 9300 ServoPLC inverter. A detailed description is given on the next pages. |

## 2.3.8 Outputs

| Identifier<br>Data type | Description |
|---|---|
| *adwFieldBusOut*<br>*ARRAY [0..15] OF DWORD* | output of the fieldbus raw data<br>These values can directly be mapped to the output data of the fieldbus IO interface. |

### User-Defined Variable Structure *scStatusWords1*

This structure implements the AIF-OUT1 interface known from the 9300 servo inverter series. It includes the following elements:

| Identifier<br>Data type | Description |
|---|---|
| *xBit00*<br>*BIT* | bit 0 of the AIF-OUT status word<br>FALSE: status inactive<br>TRUE: status active<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xImp*<br>*BIT* | bit 1 of the AIF-OUT status word: pulse inhibit active<br>FALSE: The drive's power stage is active and provides voltage/current to the motor.<br>TRUE: The drive's power stage is inactive and no current is applied to the motor.<br>**Note:** This bit must be connected to the corresponding signal in the application (i.e. by the status signal *xImpActive* of function block **L_TB2P_AxisInterface**). |
| *xBit02*<br>*BIT* | bit 2 of the AIF-OUT status word<br>FALSE: status inactive<br>TRUE: status active<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit03*<br>*BIT* | bit 3 of the AIF-OUT status word<br>FALSE: status inactive<br>TRUE: status active<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit04*<br>*BIT* | bit 4 of the AIF-OUT status word<br>FALSE: status inactive<br>TRUE: status active<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit05*<br>*BIT* | bit 5 of the AIF-OUT status word<br>FALSE: status inactive<br>TRUE: status active<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xNActEqZero*<br>*BIT* | bit 6 of the AIF-OUT status word: drive speed signal is zero<br>FALSE: drive is moving (absolute drive speed is greater than the speed tolerance window)<br>TRUE: drive is in standstill (absolute drive speed below the speed tolerance window)<br>**Note:** Generate this signal by a suitable logic (i.e. `(ABS(MCTRL_nNAct_v)<=scPar.wC0019_Nmin)`). |
| *xCInh*<br>*BIT* | bit 7 of the AIF-OUT status word: drive controllers are inhibited<br>FALSE: position/speed/current control is active<br>TRUE: position/speed/current control is reset<br>**Note:** This bit must be connected to the corresponding signal in the application (i.e. by the status signal *Status* of function block **MC_Power**). |
| *xStat1*<br>*xStat2*<br>*xStat4*<br>*xStat8*<br>*BIT* | bits 8 to 11 of the AIF-OUT status word: indication of the drive state<br>*xStat8* / *xStat4* / *xStat2* / *xStat1*<br>0 0 0 0 initialisation after the supply voltage has been connected<br>0 0 1 1 drive is in controller inhibit state<br>0 1 1 0 controller is enabled<br>0 1 1 1 a monitoring function triggered in a "message"<br>1 0 0 0 a monitoring function triggered in a "fault"<br>1 0 1 0 a monitoring function triggered in a "FAIL-QSP"<br>**Notes:** These bits must be connected to the corresponding signal in the application (i.e. by the status signals of function block **L_TB2P_AxisInterface**). Some states known from 9300 may not be possible to be indicated (see chapter 2.3.3). |

| Identifier<br>Data type | Description |
|---|---|
| *xWarning*<br>*BIT* | bit 12 of the AIF-OUT status word: warning active<br>FALSE: no drive warning is active<br>TRUE: a drive warning is active<br>**Note:** This bit must be connected to the corresponding signal in the application (i.e. by the status signals of function block **MC_ReadAxisError**). |
| *xMessage*<br>*BIT* | bit 13 of the AIF-OUT status word: message is active (i.e. under-/overvoltage state)<br>FALSE: no message is active<br>TRUE: a message is active (i.e. under-/overvoltage state)<br>**Note:** This bit must be connected to the corresponding signal in the application (i.e. by the inverted status signal *xVoltageEnabled* of function block **L_TB2P_AxisInterface**). |
| *xBit14*<br>*BIT* | bit 14 of the AIF-OUT status word<br>FALSE: status inactive<br>TRUE: status active<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *xBit15*<br>*BIT* | bit 15 of the AIF-OUT status word<br>FALSE: status inactive<br>TRUE: status active<br>**Note:** This bit does not have a fixed meaning but can be connected freely by the user. |
| *wStat*<br>*WORD* | AIF-OUT status word<br>The *wStat* signal is logically OR-connected with the bits *xBit00* … *xBit15*. This leaves it up to the user if the status is compiled indivually via the Boolean inputs *xBit00* … *xBit15* or via the *wStat* status word. |
| *wOut1*<br>*WORD* | output of a 16 bit integer number<br>Typically, the second WORD on AIF-OUT1 is interpreted as the drive's speed set value, scaled in [%] (0 … 16384 = 0.0 … 100.0[%]). However, it is up to the user to define the meaning in the application. |
| *wOut2*<br>*WORD* | output of a free 16 bit WORD value |
| *wOut3*<br>*WORD* | output of a free 16 bit WORD value |

**Tip:**

Do you need to split a 32-bit value to two 16-bit values on *scStatusWords1.wOut2* and *ScStatusWords1.wOut3*? The function block **UnpackDword**[7] provides this function. Use it in the following way:

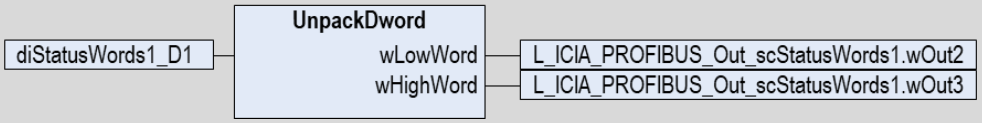



Illustration 25: conversion of a 32-bit *DWORD* value to two 16-bit *WORD* values

[7] included the *CAA Memory* library

**User-Defined Variable Structure *L_ICIA_scStatusWords***

This structure implements the extended AIF-OUT interface known from the 9300 ServoPLC inverter series. It is applied on the objects scStatusWords2 and scStatusWords3, and includes the following elements:

| Identifier<br>Data type | Description |
|---|---|
| *wOut0*<br>*WORD* | output of a free 16 bit WORD value |
| *wOut1*<br>*WORD* | output of a free 16 bit WORD value |
| *wOut2*<br>*WORD* | output of a free 16 bit WORD value |
| *wOut3*<br>*WORD* | output of a free 16 bit WORD value |

# 3 Application Example

## 3.1 Commissioning Sequence (Motion Application)

Typically, PROFIBUS is not used in new machines as there are more advanced fieldbus systems available such as EtherCAT or PROFInet. The PROFIBUS fieldbus moreover appears in existing machines in operation. This document focusses on previous Lenze servo inverters[8], which now need to be replaced by the latest device generation of i950 drives. In the best case, the replacement i950 unit requires a functional twin of the previous servo inverter. Instead of the well-known function block connection of the GDC, the PLC program of the i950 bases on Lenze's technology modules with some slight extensions to generate a 100% functional compatibility between the previous and actual drive system.

The following example shows how to migrate a 9300 servo inverter in speed control[9] to a compatible i950 signal flow, using the Lenze technology module **L_TF2P_SpeedControlBase**.

start

**Pre-Requisites:**

- »PLC Designer« is already open on your PC[10].
- No project is open in »PLC Designer«.

**Step 1**

Create a new project in »PLC Designer«:

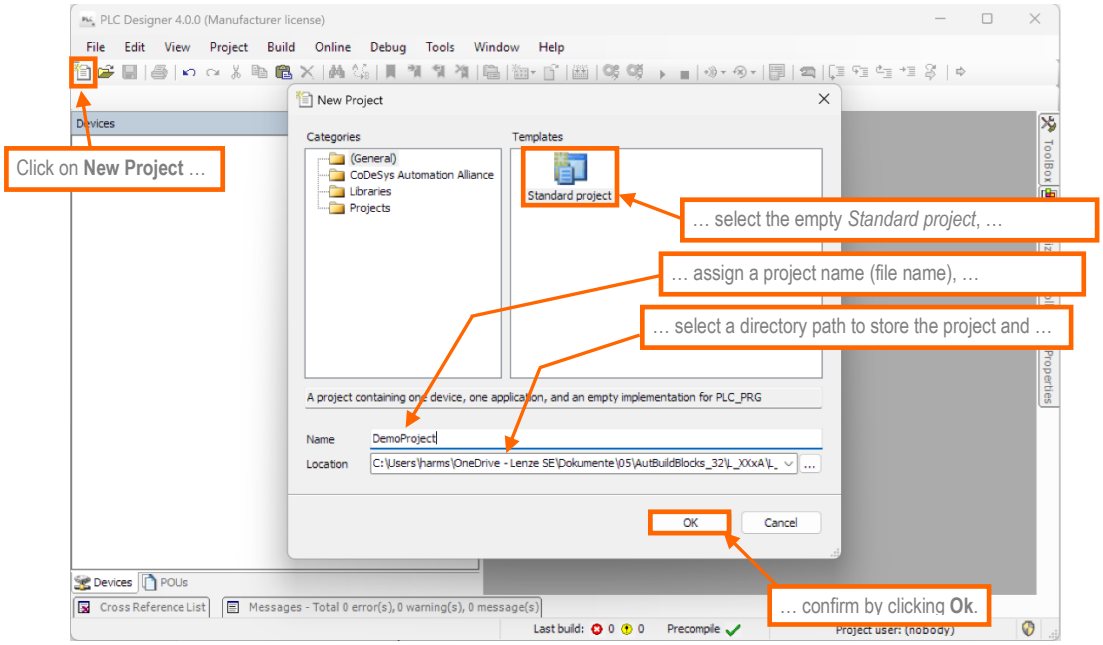



Illustration 26: creation of a new project in »PLC Designer«

[8] in particular the 9300 servo inverter series
[9] basic configuration "speed control via AIF" (C0005/000 = 1003)
[10] In this example, we use »PLC Designer« V4.x.

**Step 2**

Specify the i950 target system:

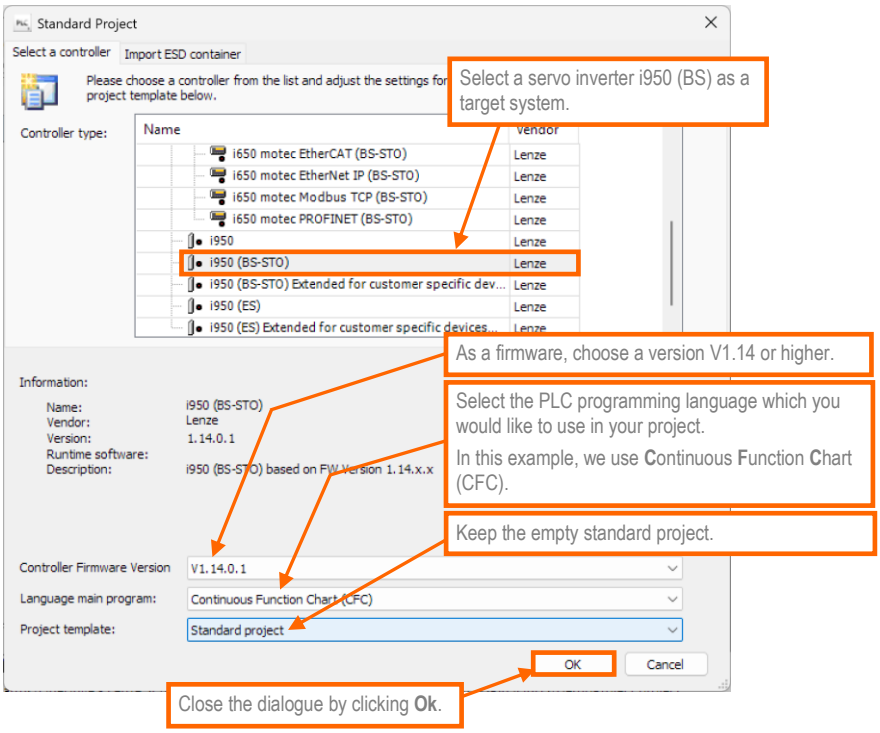



Illustration 27: select an i950 (BS) with firmware version V1.14 or higher as a target system

**Step 3**

Execute a **Build** process to enable access to the commissioning dialogues:

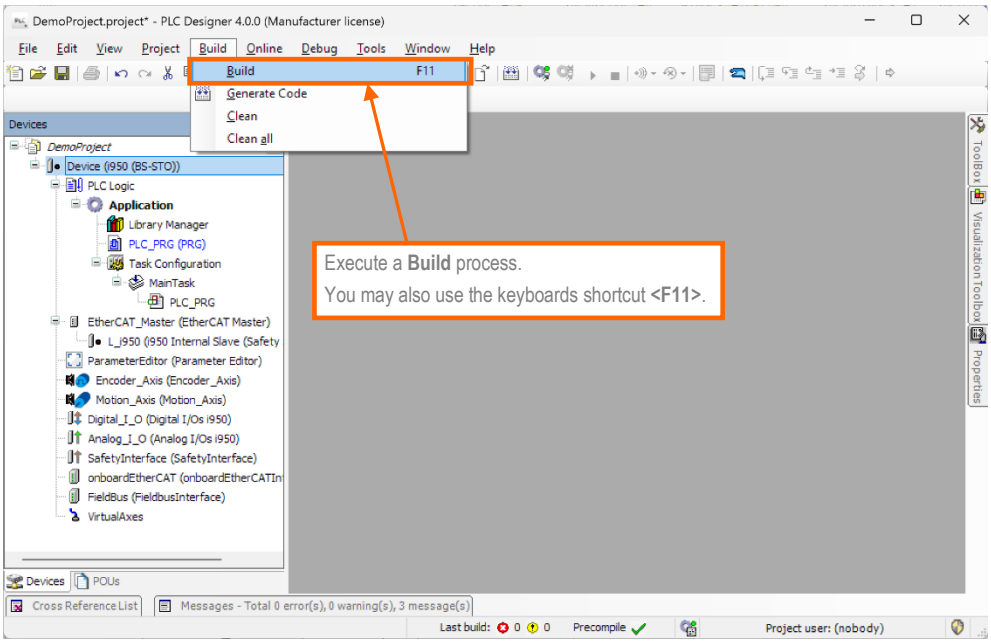



Illustration 28: Build the project to allow access to the commissioning dialogues of »PLC Designer«

**Step 4**

Set the important data in the commissioning dialogues of the device:

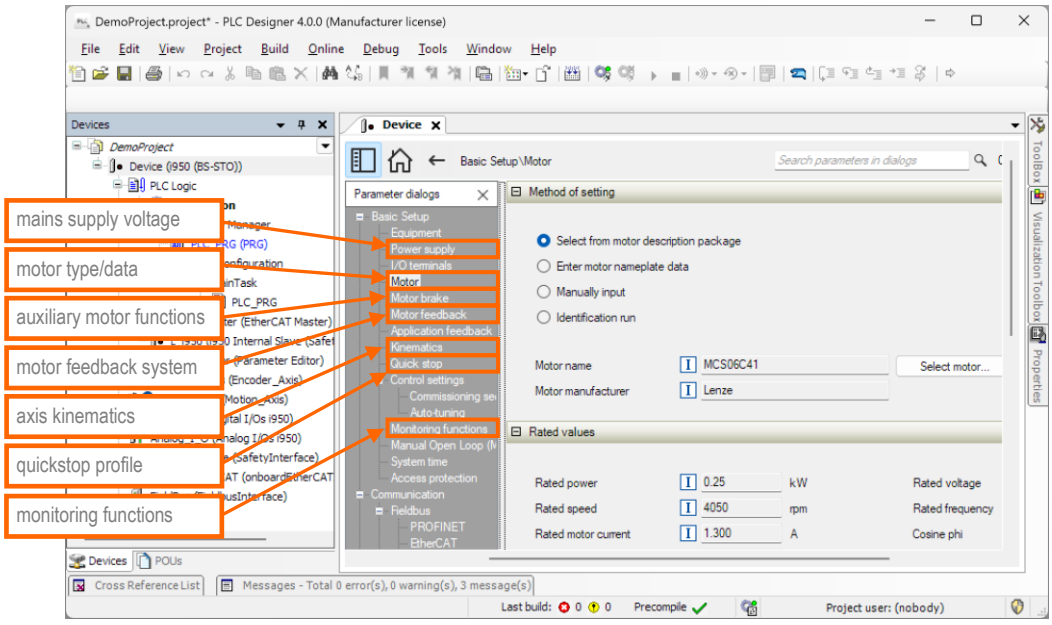



Illustration 29: basic settings of the i950 drive

- mains supply voltage
- motor data
- motor brake (if mounted/wired)
- motor feedback system
- axis kinematics (gearbox ratio, feed constant, ...)
- quickstop profile parameters
- monitoring functions (following error, end switches, ...)

**Tips**:

- Use the motor catalogue of »PLC Designer« to quickly find/set the motor data.
- The auto-tuning feature of the i950 allows to find optimum controller settings for dynamic response of the servo drive.

**Step 5**

Open the **Library Manager** to add the *L_TF2P_TechModulesFollowingPositioning* library to your project:

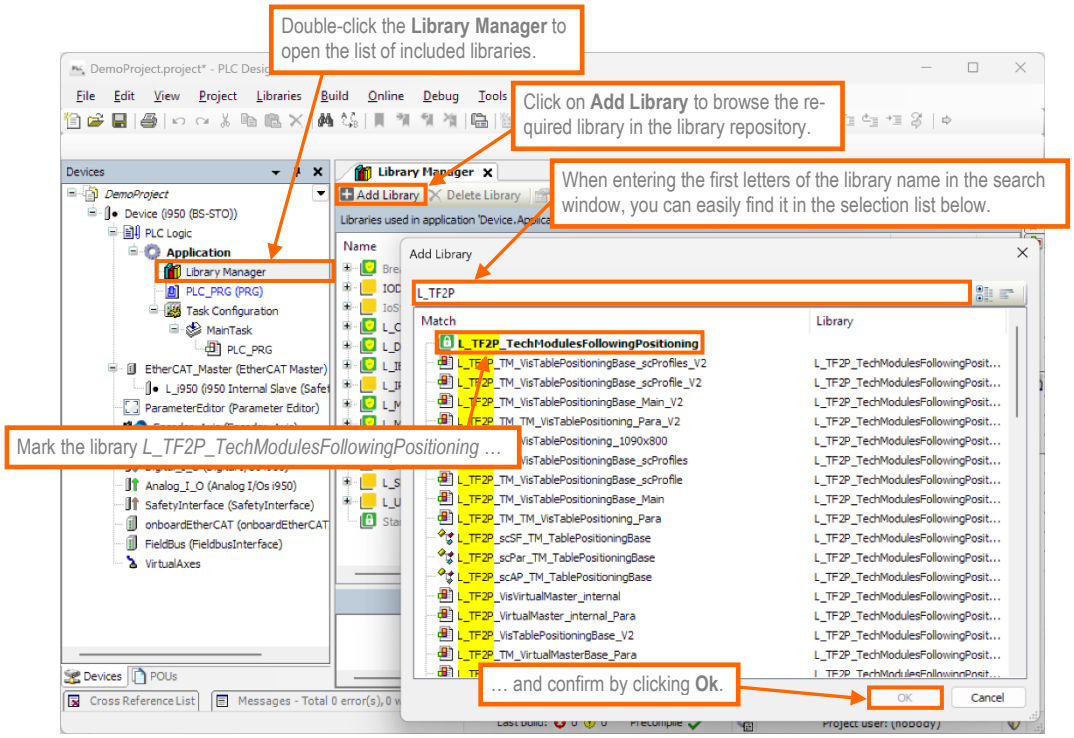



Illustration 30: adding the *L_TF2P_TechModulesFollowingPositioning* library to your project

**Step 6**

In the same way as shown in step 5, also include the *L_TB2P_TechModulesBasic* library in the **Library Manager** of your project.

**Step 7**

Open the **PLC_PRG** program and write a small CFC program as follows:

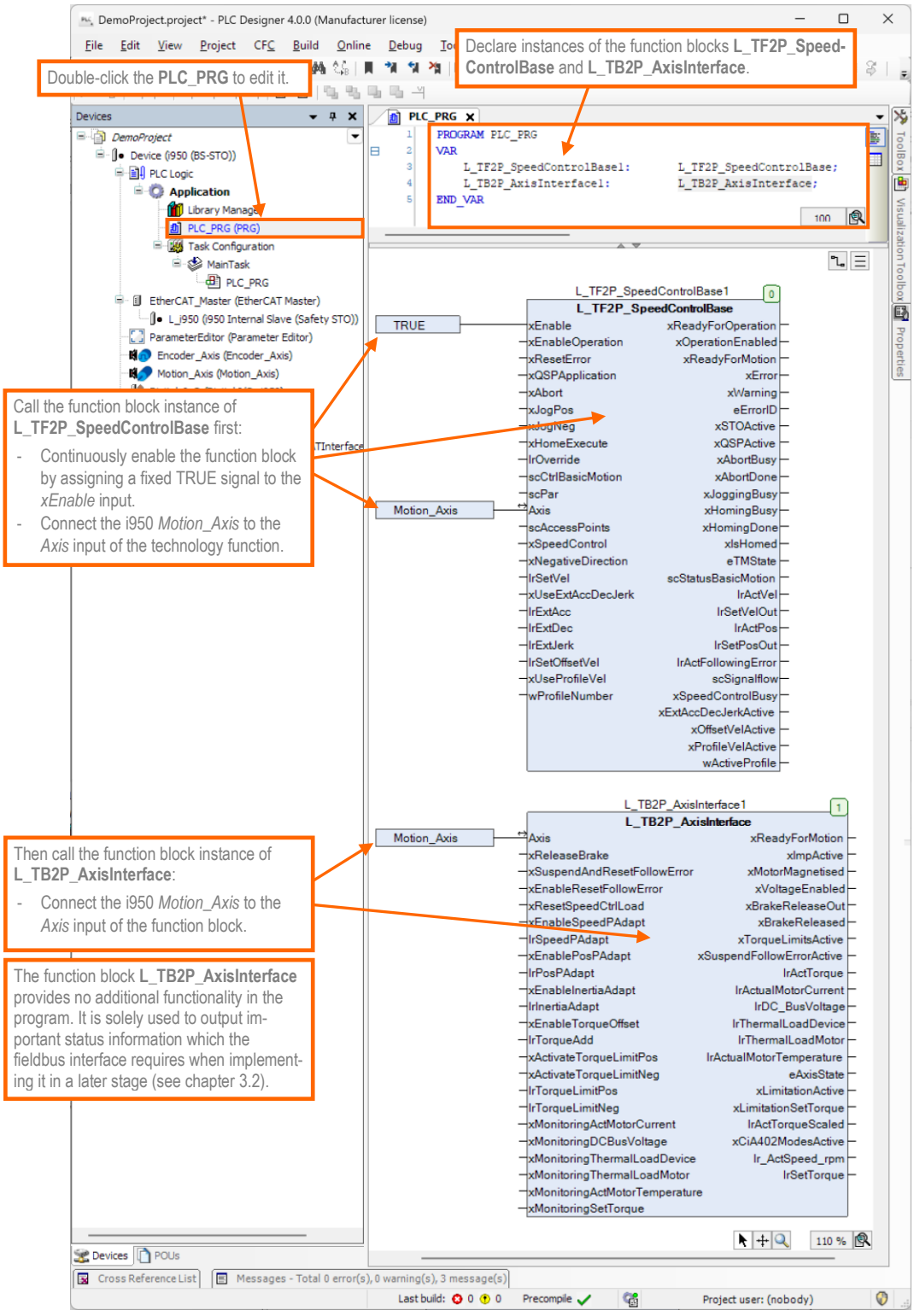



Illustration 31: calling the function blocks **L_TF2P_SpeedControlBase1** and **L_TB2P_AxisInterface1** in **PLC_PRG**

**Step 8**

Insert an empty visualization panel:

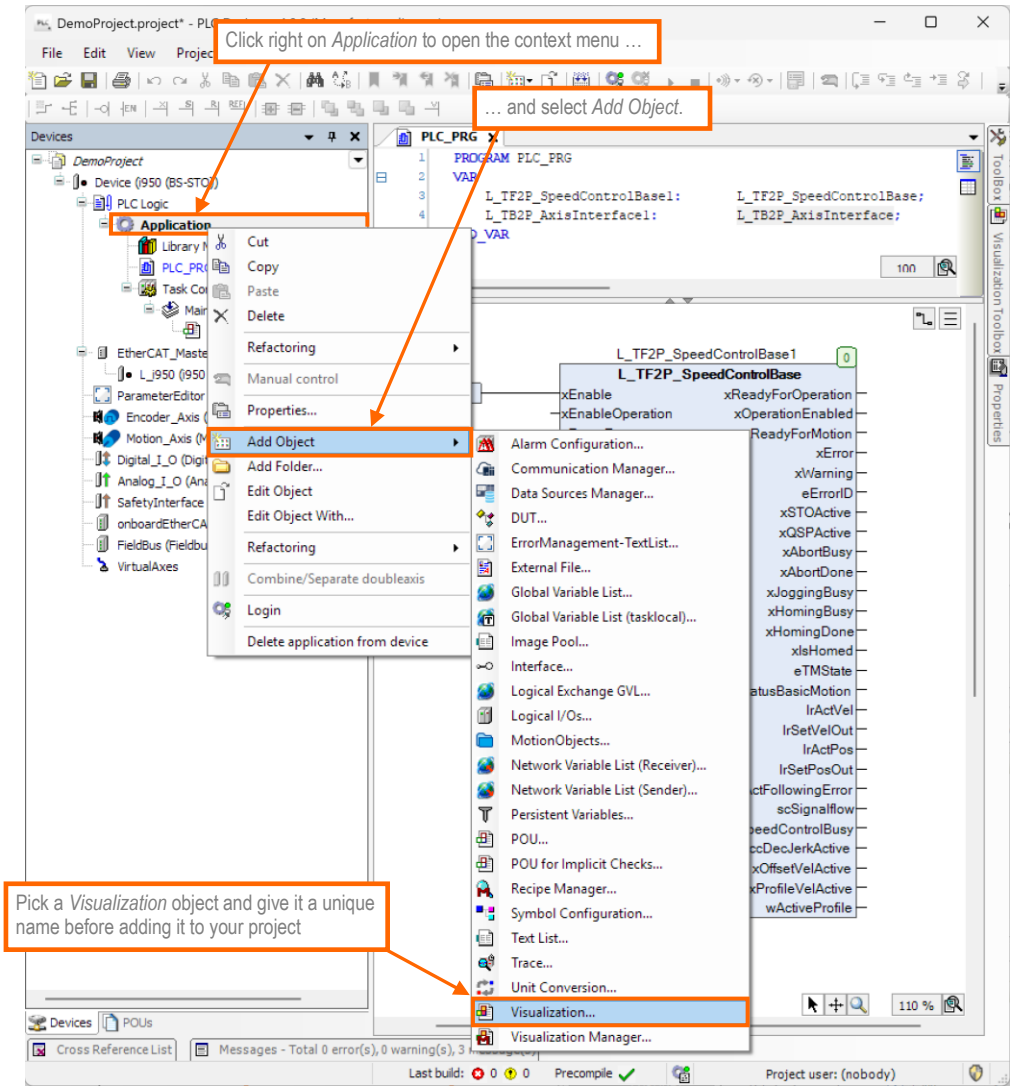



Illustration 32: adding a visualization screen to operate the function block **L_TF2P_SpeedControlBase**

**Step 9**

For a first test, insert the visualization template of the **L_TF2P_SpeedControlBase** technology module to operate it via the visu screen:

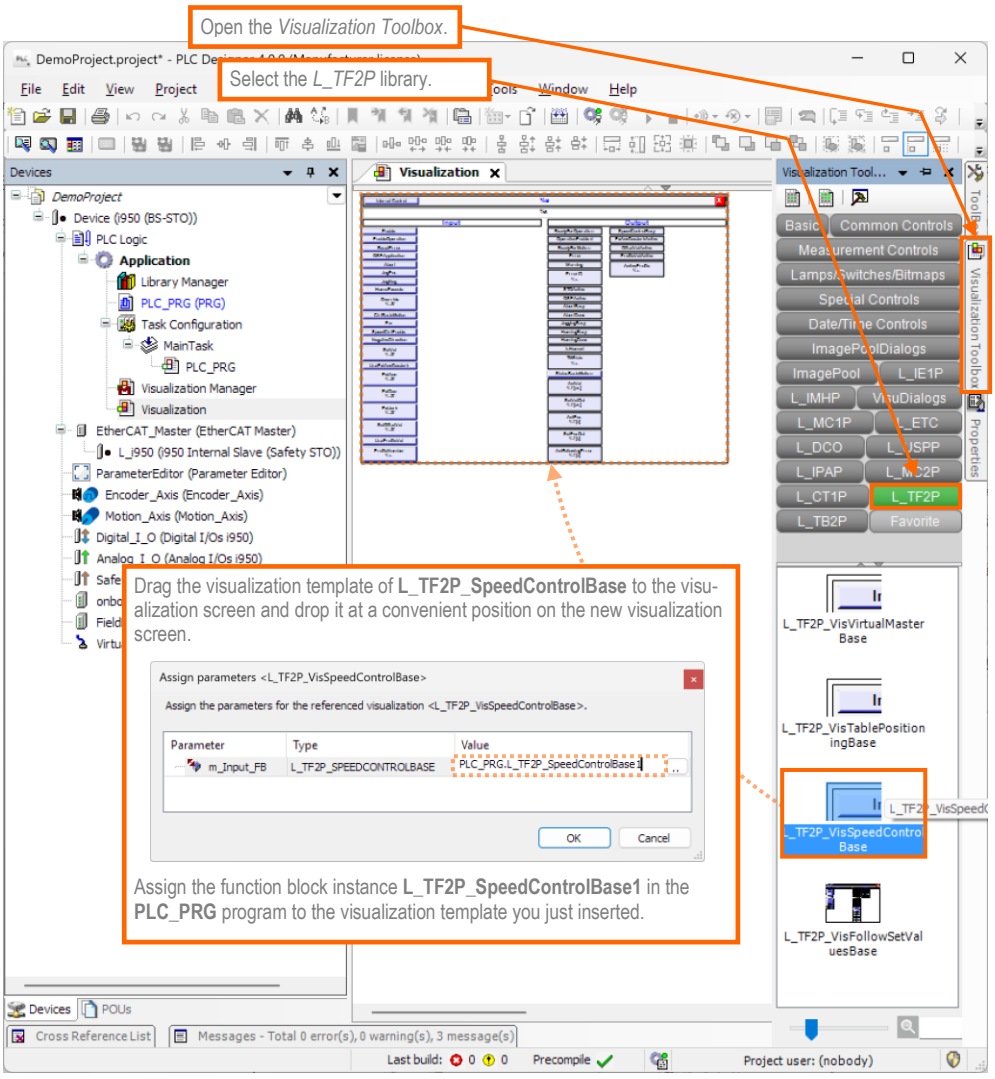



Illustration 33: adding the visualization template of **L_TF2P_SpeedControlBase**

**Step 10**

Test your PLC program:

- Switch on mains power and 24V control power on your i950 drive.
- Download the project to your i950 drive controller and start the PLC program.
- Release the STO command on the i950 drive.

**Step 11**

By means of the visualization screen, operate the i950 drive in different operation modes and check all functions:

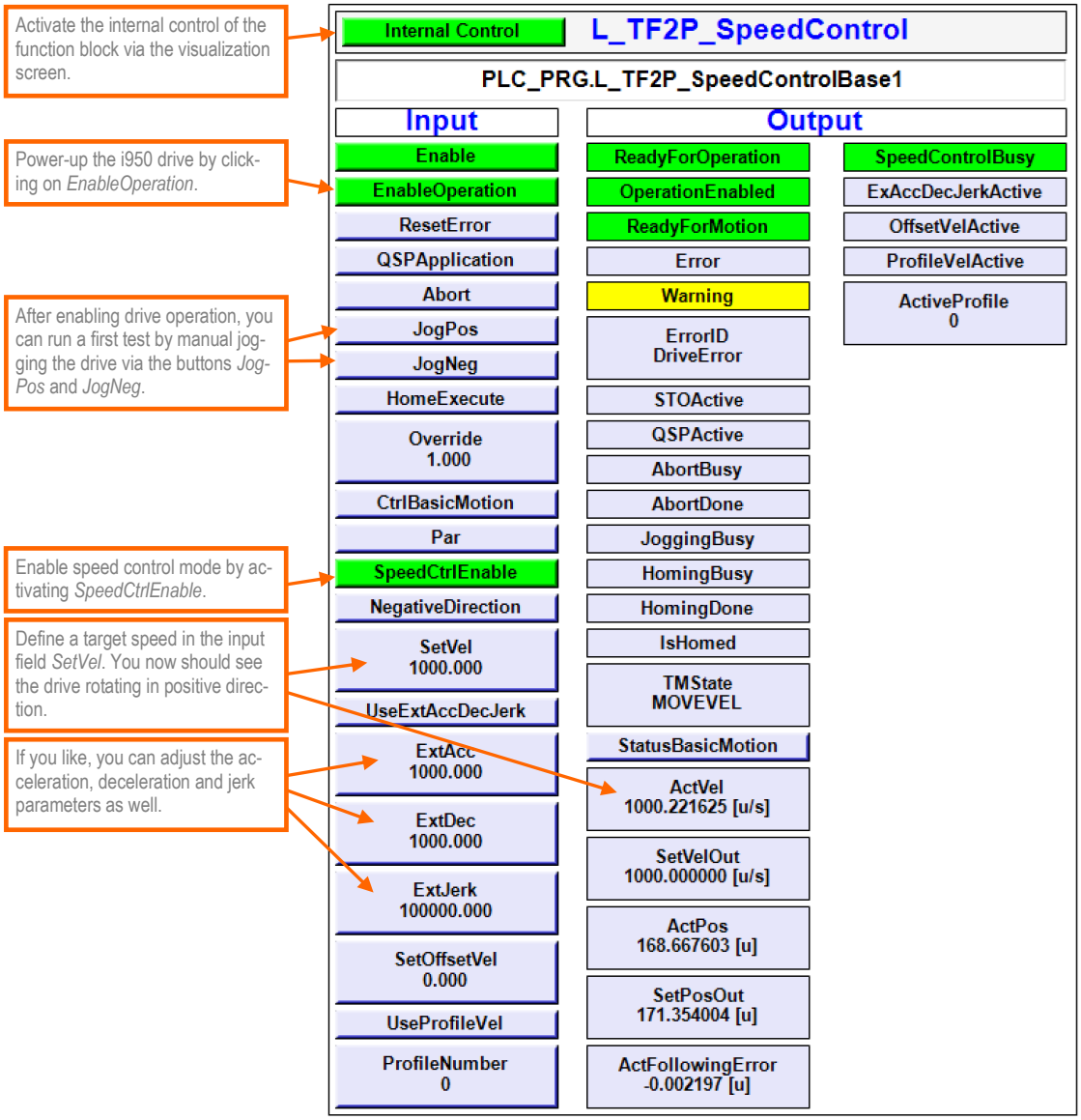



Illustration 34: visualization screen of **L_TF2P_SpeedControlBase**

end

## 3.2 Commissioning Sequence (PROFIBUS)

The following chapter describes how to set the PROFIBUS communication into operation with the help of the **L_ICIA_CommunicationInterface** function blocks.

Start

**Pre-Requisites:**

- The fieldbus system is wired according to the PROFIBUS specifications.
- The logic PLC (PROFIBUS master) as well as all PROFIBUS slave devices are supplied with control voltage ($24V_{DC}$).
- The PLC program of the i950 is open in »PLC Designer« but not yet online.
- The application signal flow has been implemented in the i950's PLC program as described in the previous chapter 3.1, for example migrating motion applications of competitors or Lenze legacy devices.

**Step 1**

Open the **Library Repository** and install the *L_ICIA_CommunicationInterface* library:

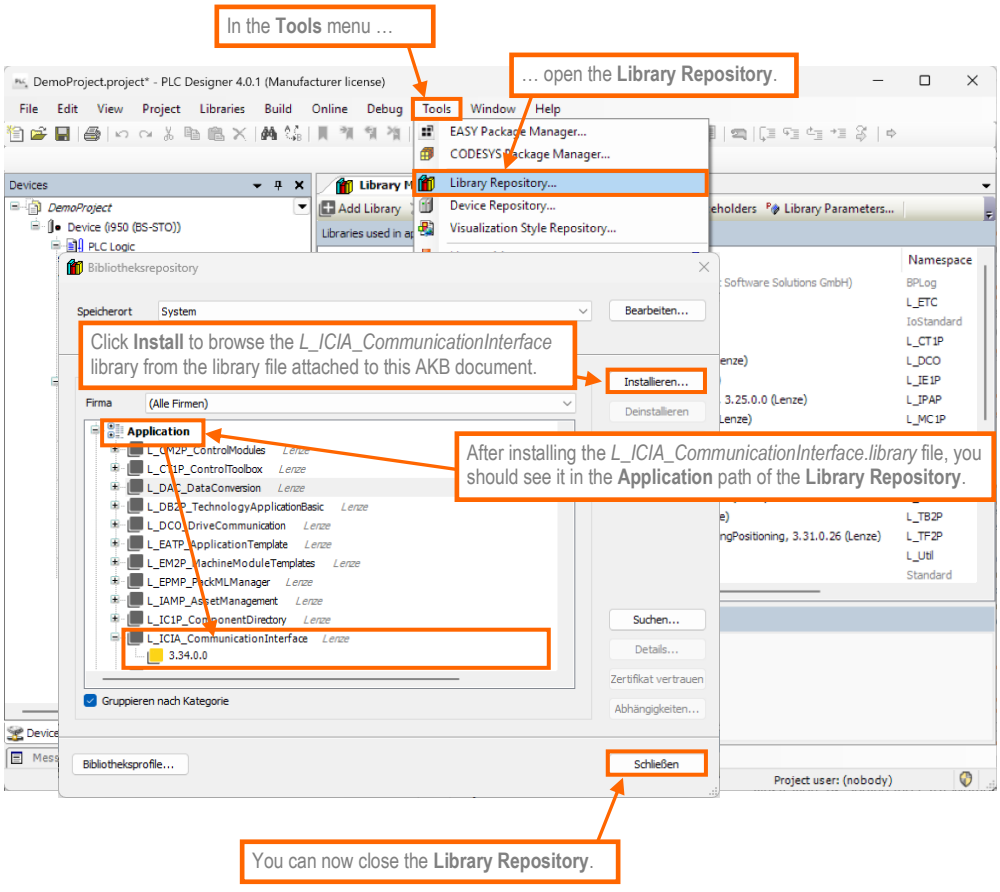



Illustration 35: adding the *L_ICIA_CommunicationInterface* library to the **Library Repository**

**Step 2**

Open the **Library Manager** to add the *L_ICIA_CommunicationInterface* library to your project:

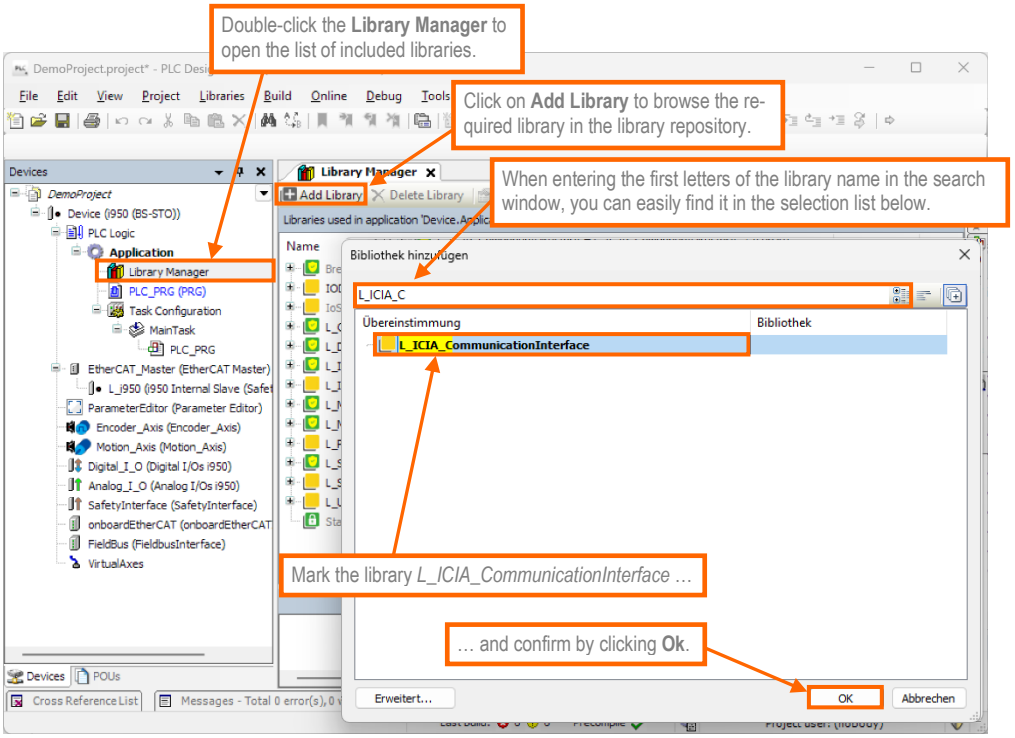



Illustration 36: adding the *L_ICIA_CommunicationInterface* library to your project

**Step 3**

In the same way as shown in step 2, also include the following libraries in the **Library Manager** of your project:

- *CAA Memory* (V03.05)
- *L_SI9P_IoDrvi900* (V03.33 – file attached to AKB document 202500431)

**Note:**
In the **Library Manager**, please resolve the placeholder for the *L_SI9P_IoDrvi900* library and change it from a device-dependent version to a fixed version V03.33 or higher.

**Tip:**
To be sure to have the correct libraries in your project, you can open the project archive *TM_SpeedControl.projectarchive* attached to the AKB article 202500431.

Create your project on the base of this project archive.

**Step 4**

Declare the following global interface variable arrays for fieldbus communication in a separate GVL item *TA_IO*:

```
{attribute 'qualified_only'}
VAR_GLOBAL
    adwPROFIBUS_IN:     ARRAY [0..15] OF DWORD;          // raw data input from PROFIBUS
    adwPROFIBUS_OUT:    ARRAY [0..15] OF DWORD;          // raw data output to PROFIBUS
END_VAR
```

**Step 5**

Map the variable arrays declared in step 4 to the fieldbus interface as follows:

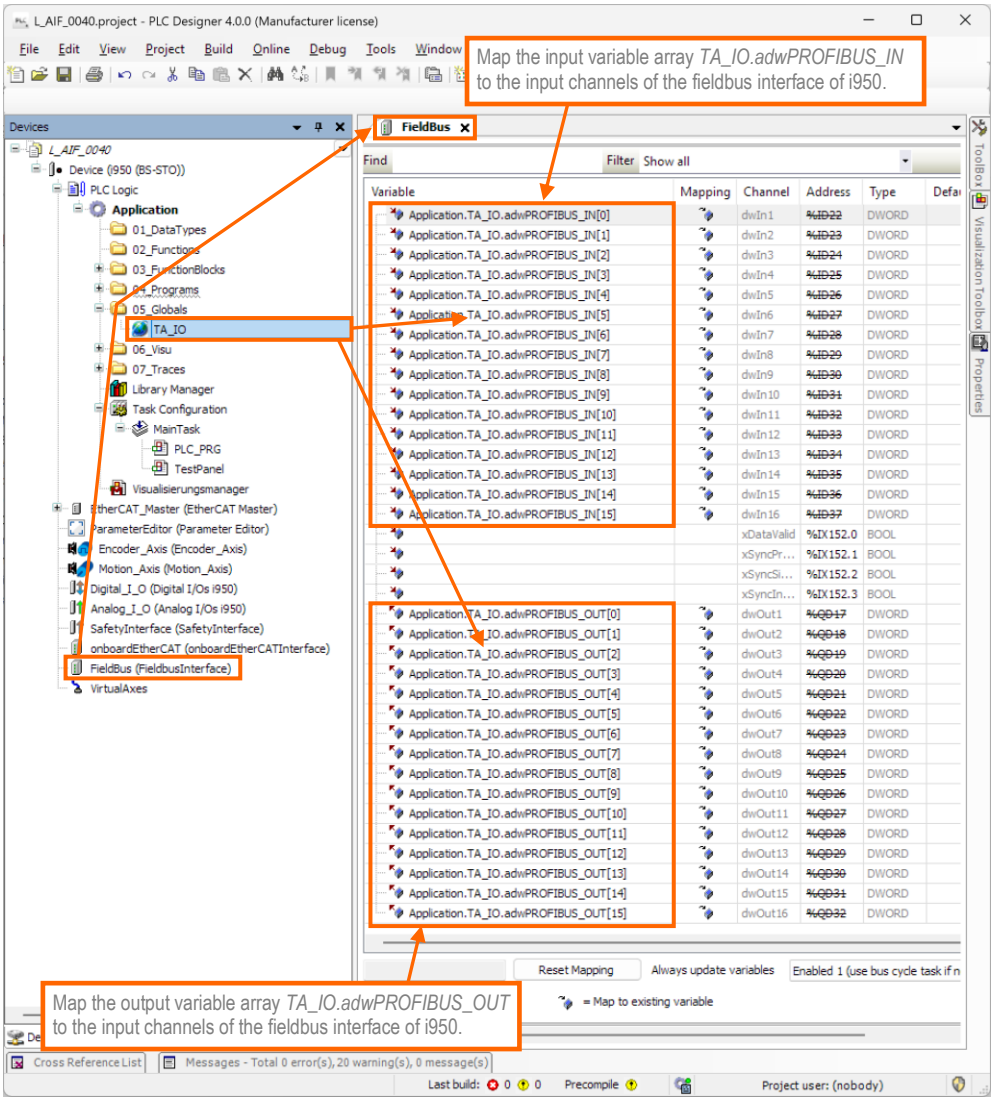



Illustration 37: assignment of global variable arrays to the i950's fieldbus interface

**Step 6**

In the i950 motion program *PLC_PRG*, declare the following function blocks:

```
VAR
   L_ICIA_PROFIBUS_Base1:L_ICIA_PROFIBUS_Base;   // handling of basic PROFIBUS communication
   L_ICIA_PROFIBUS_In1:  L_ICIA_PROFIBUS_In;     // reading/processing fieldbus inputs to
                                                 // scControlWords
   L_ICIA_PROFIBUS_Out1: L_ICIA_PROFIBUS_Out;    // processing/writing fieldbus outputs from
                                                 // scStatusWords
   L_STAT1:              L_STAT;                 // generating the 4-bit pattern of the drive status
   L_MC1A_ZeroDetect1:   L_MC1A_ZeroDetect;      // detection of rotation sense/zero speed
END_VAR
```

**Step 7**

Prepare the parameter correspondence list[11] by extending the declaration list in the i950's motion program *PLC_PRG*:

```
…
ascParReference:     ARRAY [0..15] OF L_ICIA_sc93ParReference;      // parameter reference list
…
```

**?** NO ⇒ 10

YES

Does your GSD configuration include a Drivecom V0 parameter channel?

**Step 8**

Extend the parameter correspondence list started in step 5 with the necessary initialization values as described in chapter 2.1.2:

```
…
ascParReference: ARRAY [0..15] OF L_ICIA_sc93ParReference; // parameter reference list
  (wCode:=51, bySubCode:=0, wIndex:=16#606C, bySubIndex:=0, bySize:=4, diNum:=1171875, diDen:=524288),
  (wCode:=53, bySubCode:=0, wIndex:=16#6079, bySubIndex:=0, bySize:=4, diNum:=10,      diDen:=1),
  (wCode:=63, bySubCode:=0, wIndex:=16#2D49, bySubIndex:=5, bySize:=2, diNum:=1000,    diDen:=1),
  (wCode:=52, bySubCode:=0, wIndex:=16#2D82, bySubIndex:=0, bySize:=4, diNum:=10,      diDen:=1),
  (wCode:=54, bySubCode:=0, wIndex:=16#2DD1, bySubIndex:=5, bySize:=4, diNum:=10,      diDen:=1),
  (wCode:=61, bySubCode:=0, wIndex:=16#2D84, bySubIndex:=1, bySize:=2, diNum:=10,      diDen:=1),
  (wCode:=64, bySubCode:=0, wIndex:=16#2D40, bySubIndex:=7, bySize:=2, diNum:=1,       diDen:=1),
   …
  (wCode:=84, bySubCode:=0, wIndex:=16#2C01, bySubIndex:=2, bySize:=4, diNum:=100,     diDen:=1),
  (wCode:=85, bySubCode:=0, wIndex:=16#2C01, bySubIndex:=3, bySize:=4, diNum:=10,      diDen:=1)];
…
```

**Step 9**

First, call the function block **L_ICIA_PROFIBUS_Base** in your PLC program and connect the variables as shown:

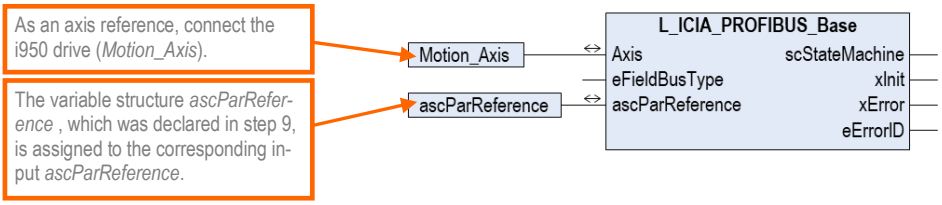



Illustration 38: call of function block **L_ICIA_PROFIBUS_Base** at the beginning of the PLC program

[11] If no parameter channel is used, still the declaration is necessary as a dummy assignment.

**Step 10**

Call the function block **L_ICIA_PROFIBUS_In** directly after the function block **L_ICIA_PROFIBUS_Base** in your PLC program and connect the variables as shown:

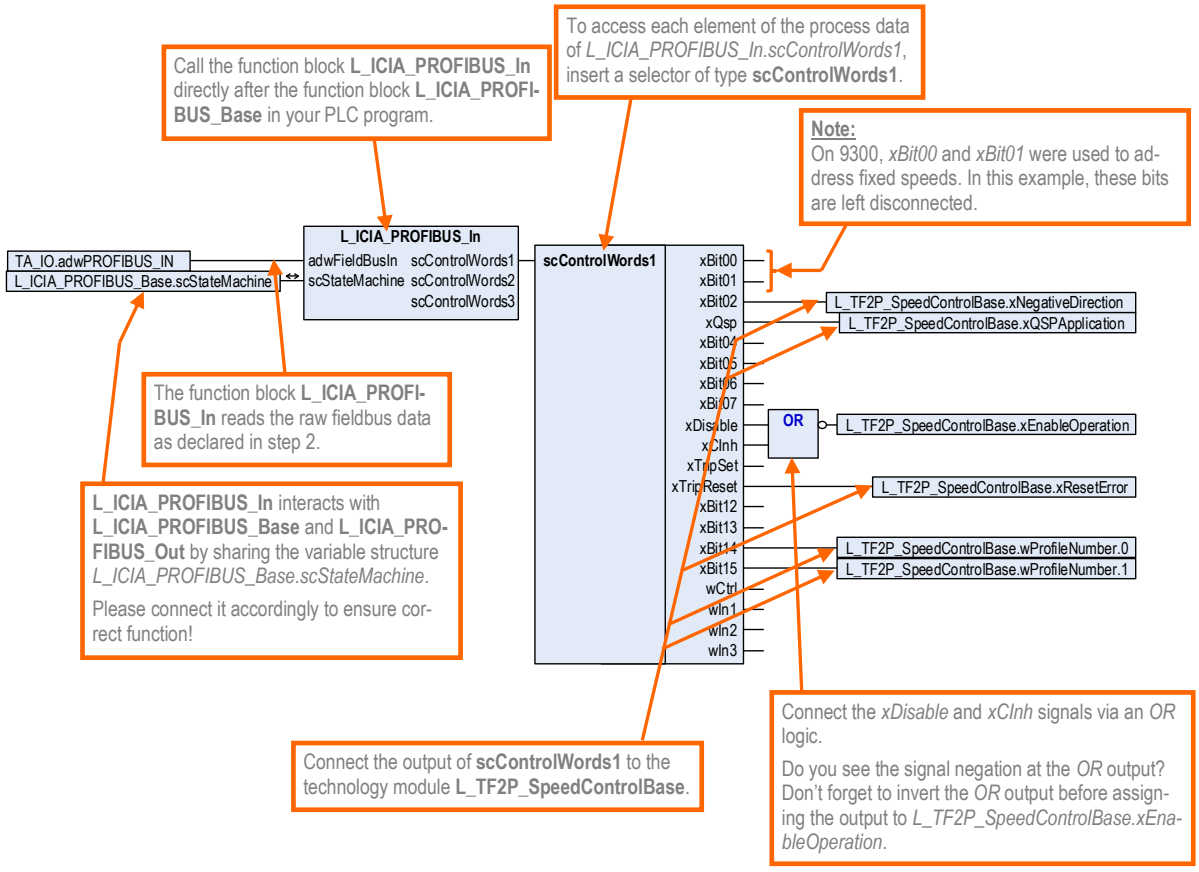



Illustration 39: call of function block **L_ICIA_PROFIBUS_In** directly behind the function block **L_ICIA_PROFIBUS_Base**

**Step 11**

Read the maximum speed value $n_{max}$[12] from the legacy drives[13] and convert it with the help of the drive axis' kinematic parameters to a reference velocity $V_{max}$.

**Example:** $n_{max}$ = 3000[rpm]

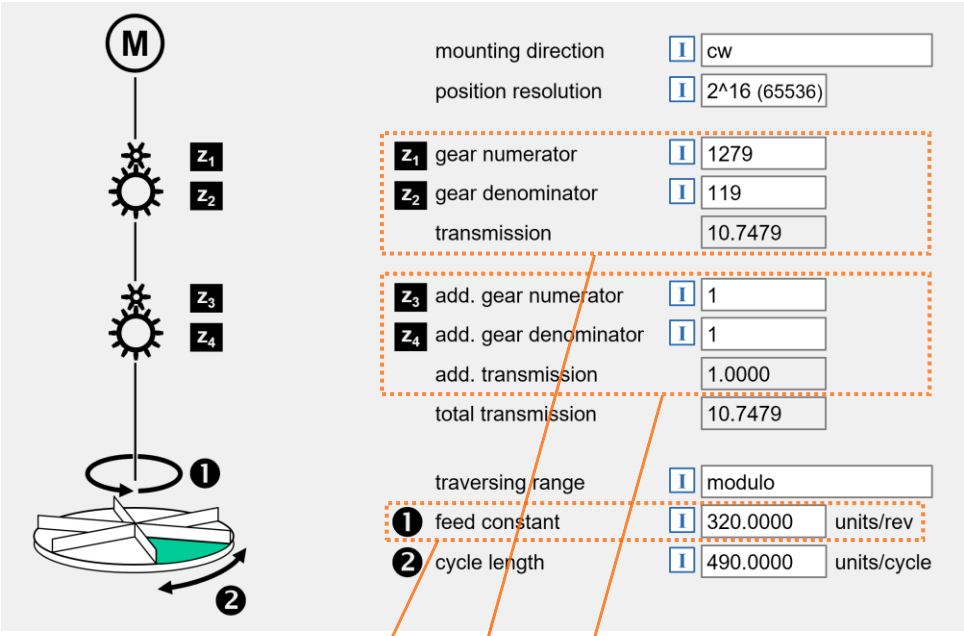



Illustration 40: kinematic parameters of the i950 drive

$$V_{max} = \frac{n_{max}}{60\frac{s}{min}} \cdot FeedConstant \cdot \frac{1}{i_{gear}} \cdot \frac{1}{i_{gear,add}} =$$

$$= \frac{n_{max}}{60\frac{s}{min}} \cdot 0x500A{:}032 \cdot \frac{0x500A{:}034}{0x500A{:}033} \cdot \frac{0x500A{:}026}{0x500A{:}025} =$$

$$= \frac{3000\frac{rev.}{min}}{60\frac{s}{min}} \cdot 320.0000\frac{units}{rev.} \cdot \frac{119}{1279} \cdot \frac{1}{1} = 1488.663 \ldots \frac{units}{s}$$

**Step 12**

Declare a constant variable *C_lrMaxVelocity* with an initialization value of $V_{max}$ as calculated in the previous step:

```
VAR CONSTANT
   C_lrMaxVelocity:  LREAL := 1488.663;          // maximum drive velocity, scaled in [units/s]
END_VAR
```

[12] scaled in [rpm]
[13] In the Lenze legacy devices, the maximum speed was set in code C0011/000.

**Step 13**

Read and scale the speed set value on *L_ICIA_PROFIBUS_In.scControlWords1.wIn1*, and route it via the following signal flow to the speed control technology module **L_TF2P_SpeedControlBase**. Modify your program as follows:

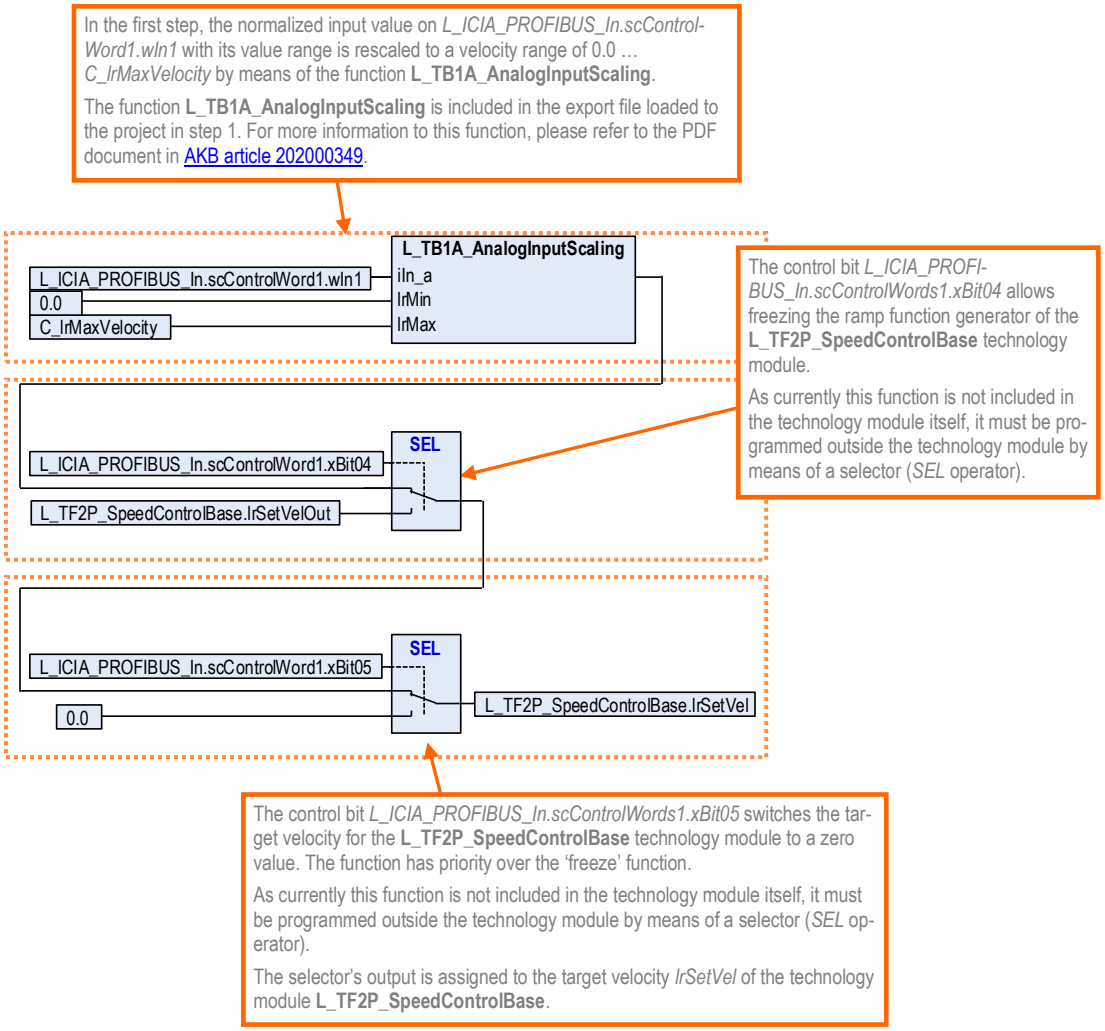



Illustration 41: reading/scaling/modifying the target velocity value *IrSetVel* of **L_TF2P_SpeedControlBase**

**Step 14**

To prepare the status word on the **L_ICIA_PROFIBUS_Out** function block, three steps of preparation work is necessary:

In the first step, declare a variable *lrZeroSpeedThreshold* for a velocity tolerance window. The tolerance window defines the behavior/stability of the zero-speed signal.

```
…
lrZeroSpeedThreshold:      LREAL := 1.0;     // zero-speed tolerance threshold, scaled in [units/s]
…
```

Initialize the variable with a velocity threshold limit value. Whenever the drive's actual velocity is smaller/equal the threshold value, the (n=0) status flag goes to a TRUE level.

**Step 15**

In the next step, call the function block **L_MC1A_ZeroDetect** at the end of your program and connect it in the signal flow as follows:

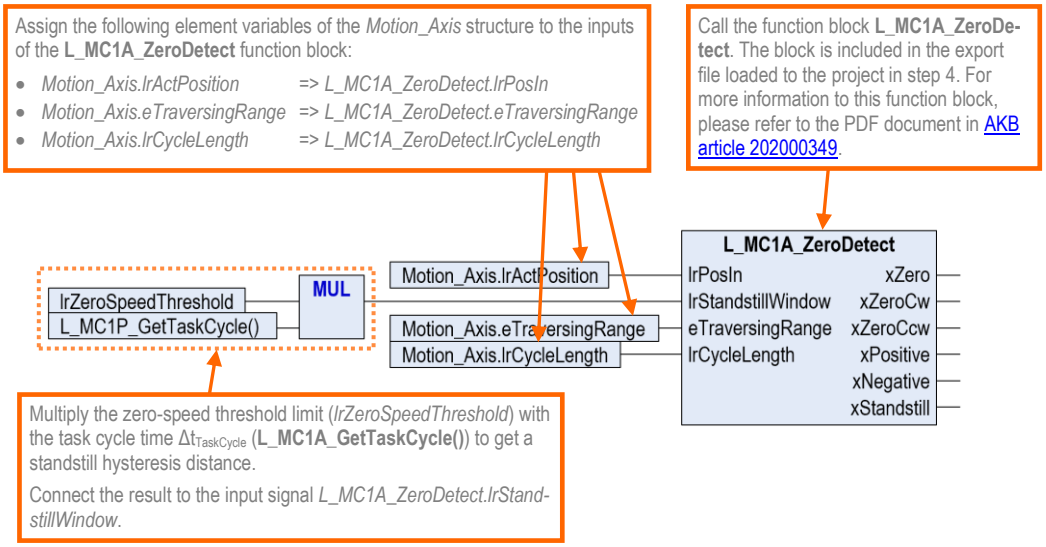



Illustration 42: zero speed detection by means of the function block **L_MC1A_ZeroDetect**

**Step 16**

Call the function block **L_STAT** at the end of your program and connect the following variables:

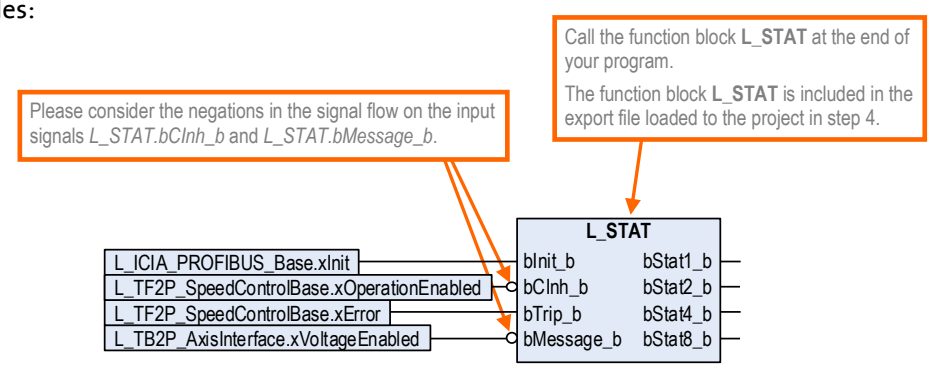



Illustration 43: generation of status bits *bStat1_b* … *bStat8_b* by means of the function block **L_STAT**

**Step 17**

Insert a composer block of *scStatusWords1* and connect the following signals:

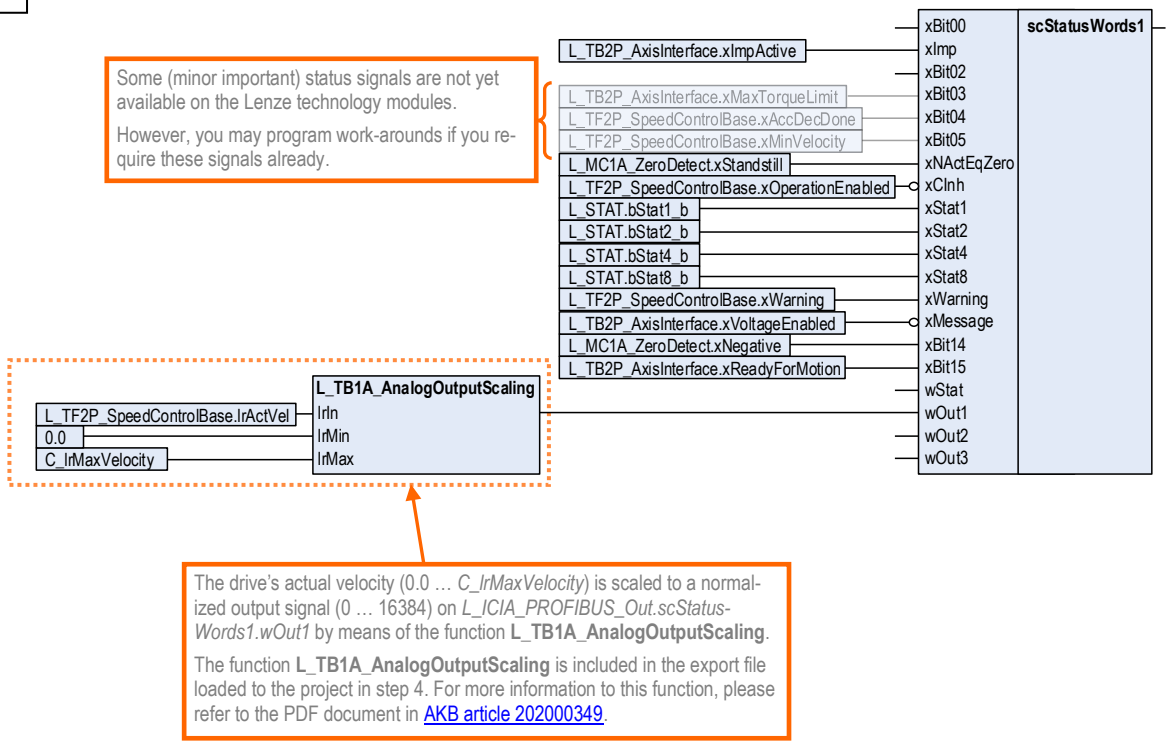



Illustration 44: assignment of status signals/of normalized motor speed to the *scStatusWords1* composer

**Step 18**

Insert the **L_ICIA_PROFIBUS_Out** function block at the end of your program, link the *L_ICIA_scStatusWords1* composer's output to the *scStatusWords1* input of the **L_ICIA_PROFIBUS_Out** block and assign the generation of the AIF status word via function block **L_ICIA_PROFIBUS_Out**:

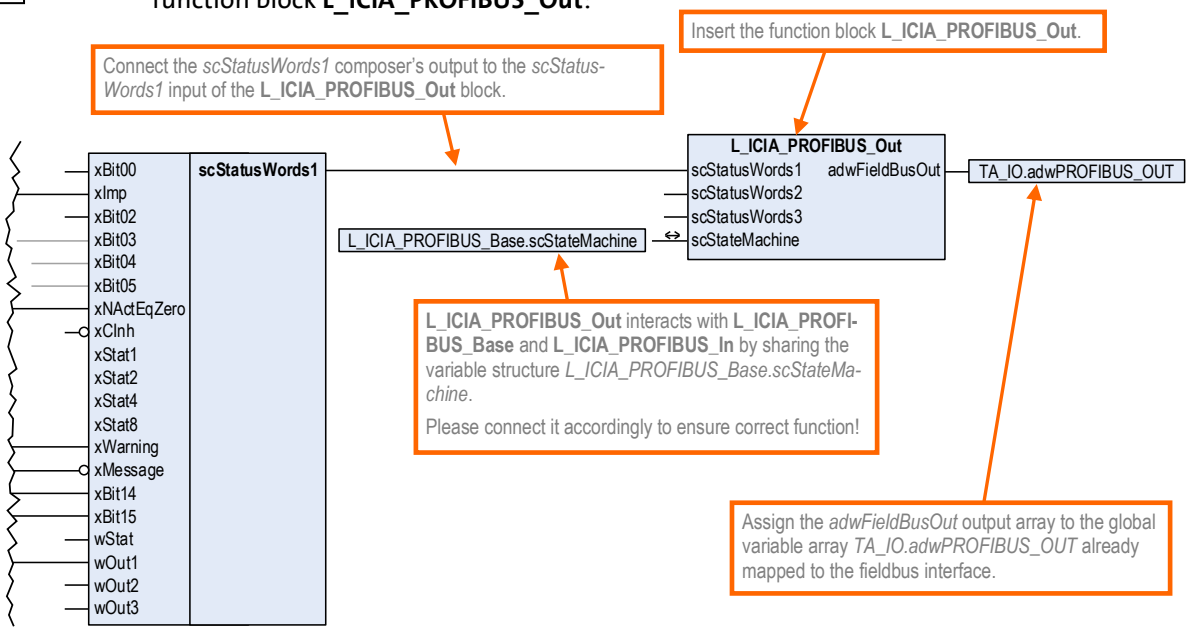
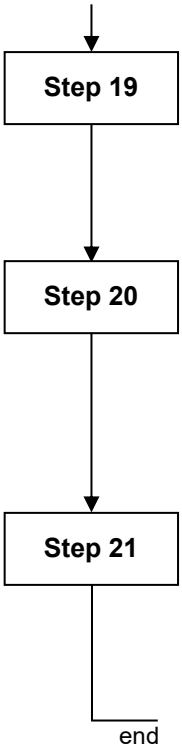



Illustration 45: call of function block **L_ICIA_PROFIBUS_Out** at program end and assignment to the fieldbus output variables




Set the PROFIBUS station address of the i950 in index 0x2341:001:

| Index | SubIndex | Name | Value | Unit |
|---|---|---|---|---|
| 0x2341 | 1 | PROFIBUS: Station address | 4 | |

Make sure that index 0x2341:011 is set to a value of 1 (‘*EMF2133IB (ID:2133)*’)[14] or of 2 (‘*EMF2131IB (ID:2131)*’)[15].

| Index | SubIndex | Name | Value | Unit |
|---|---|---|---|---|
| 0x2341 | 11 | PROFIBUS: Compatibility mode | EMF2133IB (ID: 0x2133) [1] | |

Compile, download and start the project to the i950 drive. You are now ready to control the i950 drive in the same way as the 9300 drive.

**Tip:**

After downloading, restarting of the PROFIBUS slave might be necessary due to changed values in indexes 0x2341:001 and 0x2341:011.

Restart the PROFIBUS communication with the current settings with the following command:

| Index | SubIndex | Name | Value | Unit |
|---|---|---|---|---|
| 0x2340 | 0 | PROFIBUS communication | no action / no error [0]<br>restart with current values [1]<br>restart with default values [2]<br>stop communication [5]<br>in progress [10]<br>action cancelled [11]<br>fault [12] | |

[14] This setting defines which device type is reported via PROFIBUS to the logic PLC. A setting of 0x2341:11=1 makes the logic PLC believe that the connected device is a 8200/9300 with a PROFIBUS module EMF2133IB.
[15] A setting of 0x2341:11=2 makes the logic PLC believe that the connected device is a 8200/9300 with a PROFIBUS module EMF2131IB.

# 4 Appendix

## 4.1 Supported GSD Configurations[16]

| # | GSD configuration | PROFIBUS config value(s) | corresponding value in 0x2348:003 |
|---|---|---|---|
| ***1. no parameter channel / process data (Drivecom control)*** | | | |
| 1 | PZD( 1W) | 0x70 | xx0170 |
| … | … | … | … |
| 12 | PZD(12W) | 0x7B | xx017B |
| ***2. consistent Drivecom parameter channel / process data (Drivecom control)*** | | | |
| 13 | PAR(cons.) + PZD( 1W) | 0xF3, 0x70 | xx02F370 |
| … | … | … | .. |
| 24 | PAR(cons.) + PZD(12W) | 0xF3, 0x7B | xx02F37B |
| ***3. consistent Drivecom parameter channel / consistent process data (Drivecom control)*** | | | |
| 25 | PAR(cons.) + PZD( 1W cons.) | 0xF3, 0xF0 | xx02F3F0 |
| … | … | … | … |
| 36 | PAR(cons.) + PZD(12W cons.) | 0xF3, 0xFB | xx02F3FB |
| ***4. Drivecom parameter channel / process data (Drivecom control)*** | | | |
| 37 | PAR + PZD( 1W) | 0x73, 0x70 | xx027370 |
| … | … | … | … |
| 48 | PAR + PZD(12W) | 0x73, 0x7B | xx02737B |
| ***5. Drivecom parameter channel / consistent process data (Drivecom control)*** | | | |
| 49 | PAR + PZD( 1W cons.) | 0x73, 0xF0 | xx0273F0 |
| … | … | … | … |
| 60 | PAR + PZD(12W cons.) | 0x73, 0xFB | xx0273FB |
| ***6. no parameter channel / consistent process data (Drivecom control)*** | | | |
| 61 | PZD( 1W cons.) | 0xF0 | xx01F0 |
| … | … | … | … |
| 72 | PZD(12W cons.) | 0xFB | xx01FB |
| ***7. no parameter channel / process data (Lenze device control)*** | | | |
| 73 | PZD( 1W) AR | 0x00, 0x00, 0x00, 0x70 | xx0400000070 |
| … | … | … | … |
| 84 | PZD(12W) AR | 0x00, 0x00, 0x00, 0x7B | xx040000007B |
| ***8. consistent Drivecom parameter channel / process data (Lenze device control)*** | | | |
| 85 | PAR(cons.) + PZD( 1W) AR | 0x00, 0x00, 0x00, 0xF3, 0x70 | xx05000000F370 |
| … | … | … | … |
| 96 | PAR(cons.) + PZD(12W) AR | 0x00, 0x00, 0x00, 0xF3, 0x7B | xx05000000F37B |
| ***9. consistent Drivecom parameter channel / consistent process data (Lenze device control)*** | | | |
| 97 | PAR(cons.) + PZD( 1W cons.) AR | 0x00, 0x00, 0x00, 0xF3, 0xF0 | xx05000000F3F0 |
| … | … | … | … |
| 108 | PAR(cons.) + PZD(12W cons.) AR | 0x00, 0x00, 0x00, 0xF3, 0xFB | xx05000000F3FB |
| ***10. Drivecom parameter channel / process data (Lenze device control)*** | | | |
| 109 | PAR + PZD( 1W) AR | 0x00, 0x00, 0x00, 0x73, 0x70 | xx050000007370 |
| … | … | … | … |
| 120 | PAR + PZD(12W) AR | 0x00, 0x00, 0x00, 0x73, 0x7B | xx05000000737B |
| ***11. Drivecom parameter channel / consistent process data (Lenze device control)*** | | | |
| 121 | PAR + PZD( 1W cons.) AR | 0x00, 0x00, 0x00, 0x73, 0xF0 | xx0500000073F0 |
| … | … | … | … |
| 132 | PAR + PZD(12W cons.) AR | 0x00, 0x00, 0x00, 0x73, 0xFB | xx0500000073FB |
| ***12. no parameter channel / consistent process data (Lenze device control)*** | | | |
| 133 | PZD( 1W cons.) AR | 0x00, 0x00, 0x00, 0xF0 | xx04000000F0 |
| … | … | … | … |
| 144 | PZD(12W cons.) AR | 0x00, 0x00, 0x00, 0xFB | xx04000000FB |

[16] no distinguishing between inconsistent/consistent data transmission

## 4.2 AIF-IN Interface of 9300

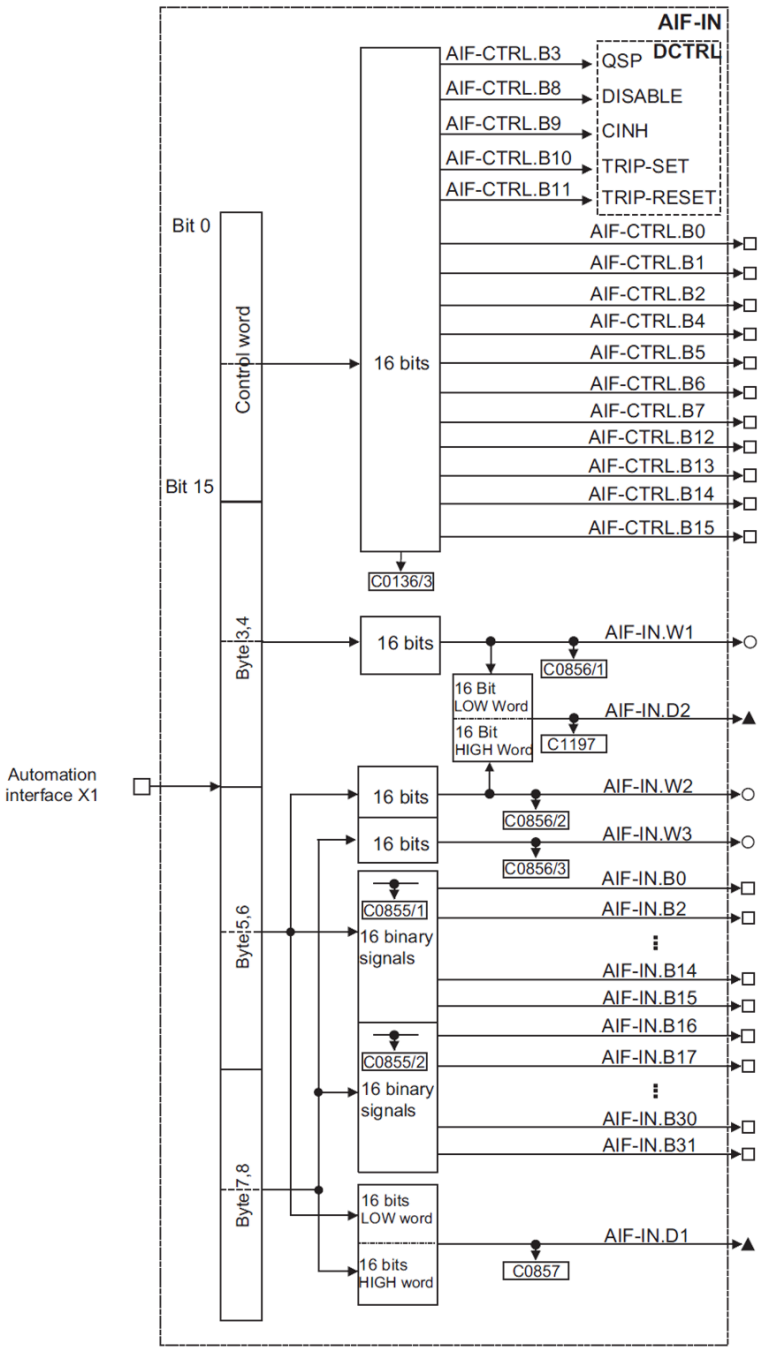



Illustration 46: signal flow of the **AIF-IN** interface on the 9300 servo inverter (excerpt from GDC help)

## 4.3 AIF-OUT Interface of 9300

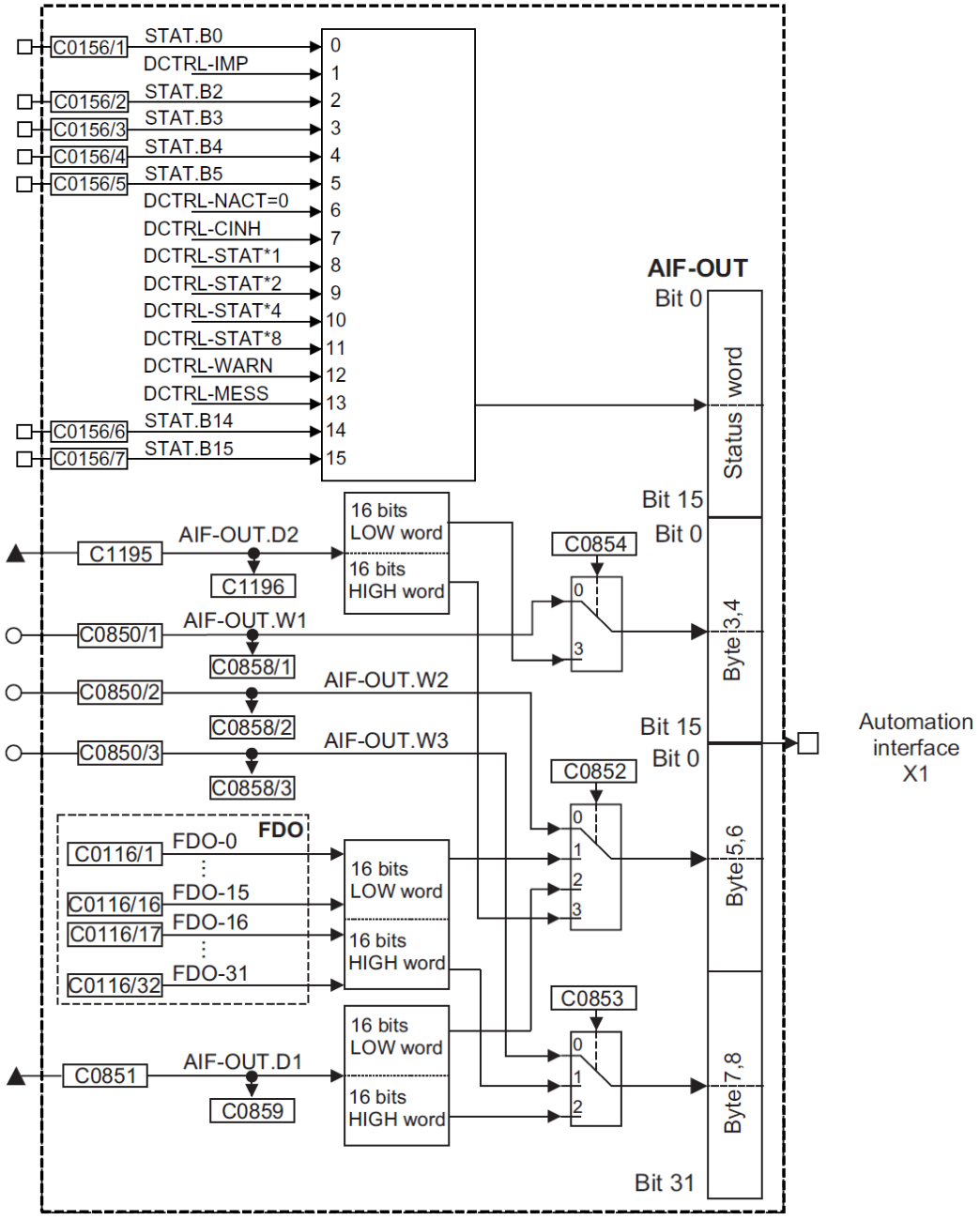



Illustration 47: signal flow of the **AIF-OUT** interface on the 9300 servo inverter (excerpt from GDC help)

## 4.4 Drivecom Control Word

| Bit | Name | Meaning |
|---|---|---|
| 0 | *Switch On* | command bit:<br>FALSE: commands 2, 6, 8 (controller inhibit)<br>TRUE: command 3 (controller enable) |
| 1 | *Voltage Inhibit* | command bit: disable/enable motor voltage<br>FALSE: inhibit voltage<br>TRUE: enable voltage |
| 2 | *Quick Stop* | command bit: activate quick stop<br>FALSE: activate quick stop<br>TRUE: release quick stop |
| 3 | *Enable Operation* | command bit: enable drive operation<br>FALSE: disable drive operation<br>TRUE: enable drive operation |
| 4 | *RFG Inhibit* | command bit: application quick stop (QSP)<br>FALSE: activate application quick stop (QSP)<br>TRUE: release application quick stop (QSP)<br>**Note:** The negated signal of this bit is directly output on *scControlWords1.xQsp*. |
| 5 | *RFG Stop* | command bit: stop ramp function generator<br>FALSE: ramp function generator freezes<br>The drive maintains the actual speed even if the target speed on *scControlWords1.iIn2* is not reached yet.<br>TRUE: ramp function generator is active<br>The drive accelerates/decelerates to the target speed on *scControlWords1.iIn2*.<br>**Notes:**<br>• The negated signal of this bit is directly output on *scControlWords1.xBit04*.<br>• In the basic application 'SpeedControl', bit 5 (*RFG Stop*) has minor priority against bit 6 (*RFG Zero*). |
| 6 | *RFG Zero* | command bit: ramp down set speed to zero<br>FALSE: zero target speed<br>The drive ramps down to a zero speed. The value received on *scControlWords1.iIn2* is ignored.<br>TRUE: external target speed<br>The drive follows the target speed on *scControlWords1.iIn2*.<br>**Notes:**<br>• The negated signal of this bit is directly output on *scControlWords1.xBit05*.<br>• In the basic application 'SpeedControl', bit 6 (*RFG Zero*) has priority over bit 5 (*RFG Stop*). |
| 7 | *Error Reset* | command bit: reset drive error<br>FALSE=>TRUE: resets a drive error<br>**Notes:**<br>• A drive error can only be reset in case the error cause has been removed before.<br>• This bit is directly output on *scControlWords1.xTripReset*. |
| 8 … 10 | (reserved) | |
| 11 | Manufacturer | free bit (directly output on *scControlWords1.xBit07*) |
| 12 | Manufacturer | free bit (directly output on *scControlWords1.xBit12*) |
| 13 | Manufacturer | free bit (directly output on *scControlWords1.xBit13*) |
| 14 | Manufacturer | free bit (directly output on *scControlWords1.xBit14*) |
| 15 | Manufacturer | free bit (directly output on *scControlWords1.xBit15*) |

## 4.5 Drivecom Status Word

| Bit | Name | Meaning |
|---|---|---|
| 0 | *Ready To Start* | device state machine information:<br>FALSE: The device status is lower than *Ready To Start*.<br>TRUE: The device status is at least *Ready To Start*. |
| 1 | *Switched On* | device state machine information:<br>FALSE: The device status is lower than *Switched On*.<br>TRUE: The device status is at least *Switched On*. |
| 2 | *Operation Enabled* | device state machine information:<br>FALSE: The device status is lower than *Operation Enabled*.<br>TRUE: The device status is at least *Operation Enabled*. |
| 3 | *Fault* | device is in error state:<br>FALSE: no error is active on the device<br>TRUE: an error is active on the device<br>**Note:** The signal is derived from *scStatusWords1.xStat8*, *scStatusWords1.xStat10* and *scStatusWords1.xStat11*. |
| 4 | *Voltage Inhibited* | handshake signal: return of control bit 1 ("*Voltage Inhibit*")<br>FALSE: no error is active on the device<br>TRUE: an error is active on the device<br>**Note:** The signal is directly copied from bit 1 of the Drivecom control word (see previous chapter 4.4). |
| 5 | *Quick Stop* | handshake signal: return of control bit 2 (*Quick Stop*)<br>FALSE: quick stop command is active on the device<br>TRUE: no quick stop command is active on the device<br>**Note:** The signal is directly copied from bit 2 or bit 4 of the Drivecom control word (see previous chapter 4.4). |
| 6 | *Switch-On Inhibited* | device state machine information:<br>FALSE: The device is not in state *Switch-On Inhibited*.<br>TRUE: The device is in state *Switch-On Inhibited*. |
| 7 | *Warning* | device is in warning state:<br>FALSE: no warning is active on the device<br>TRUE: a warning is active on the device<br>**Note:** The signal of this bit is directly copied from *scStatusWords1.xWarning*. |
| 8 | *Message* | message is active on the device:<br>FALSE: no message is active on the device<br>TRUE: a message is active on the device<br>**Notes:**<br>• A message state typically occurs in at an undervoltage state (main power switched off).<br>• The signal of this bit is directly copied from *scStatusWords1.xMessage*. |
| 9 | *Remote* | fieldbus access authorization:<br>FALSE: -<br>TRUE: (this signal is set always TRUE in Drivecom operation mode) |
| 10 | *Set Point Reached* | status of the internal ramp generator:<br>FALSE: The actual drive speed does not match the target value.<br>TRUE: The actual drive speed matches the target value.<br>**Notes:**<br>• In default speed control, the signal represents the *Set Point Reached* status of the speed ramp generator. In this case, the following Drivecom command bits may suppress the *Set Point Reached* status signal:<br>– *RFG Inhibit* (command bit 4)<br>– *RFG Stop* (command bit 5)<br>– *RFG Zero* (command bit 6)<br>• Generally, the signal of this bit is directly copied from *scStatusWords1.xBit04*. |
| 11 | *Limit Value* | Status of the Drivecom speed limitation (not supported):<br>FALSE: (this signal is set always FALSE in Drivecom operation mode)<br>TRUE: - |
| 12 | Manufacturer | free bit (signal directly copied from *scStatusWords1.xBit14*) |
| 13 | Manufacturer | free bit (signal directly copied from *scStatusWords1.xBit03*) |
| 14 | Manufacturer | free bit (signal directly copied from *scStatusWords1.xBit02*) |
| 15 | Manufacturer | free bit (signal directly copied from *scStatusWords1.xBit05*) |

## 4.6 Drivecom DP V0 Parameter Channel (Tx)

The following chart describes the meaning of the transmit parameter channel request (8 bytes), sent by the PLC to the slave device (drive):

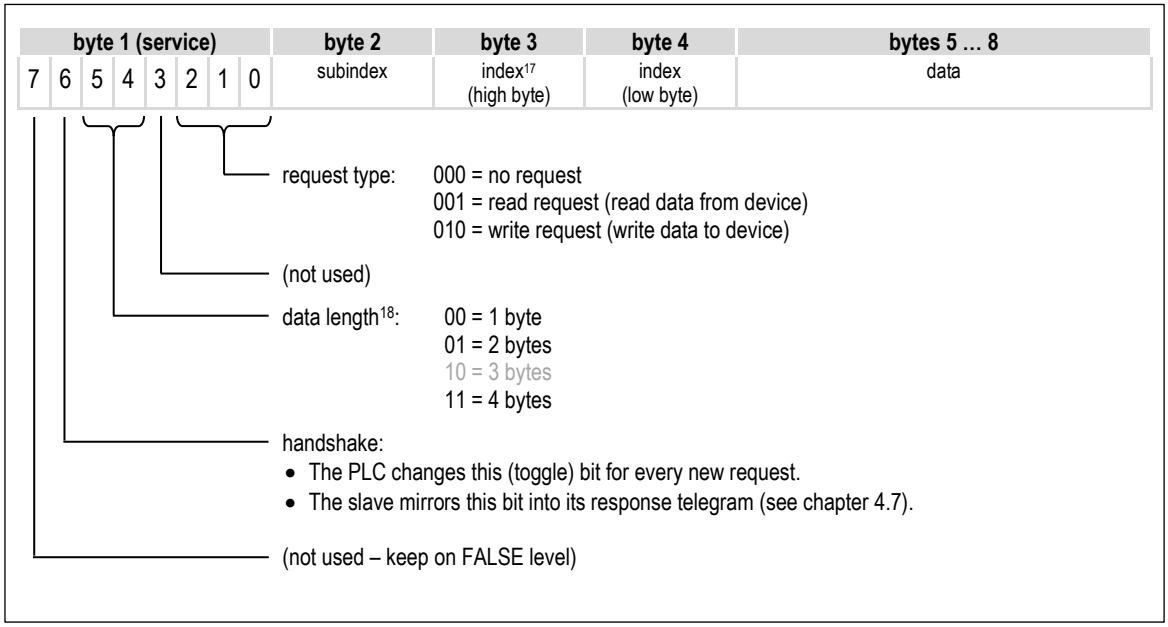
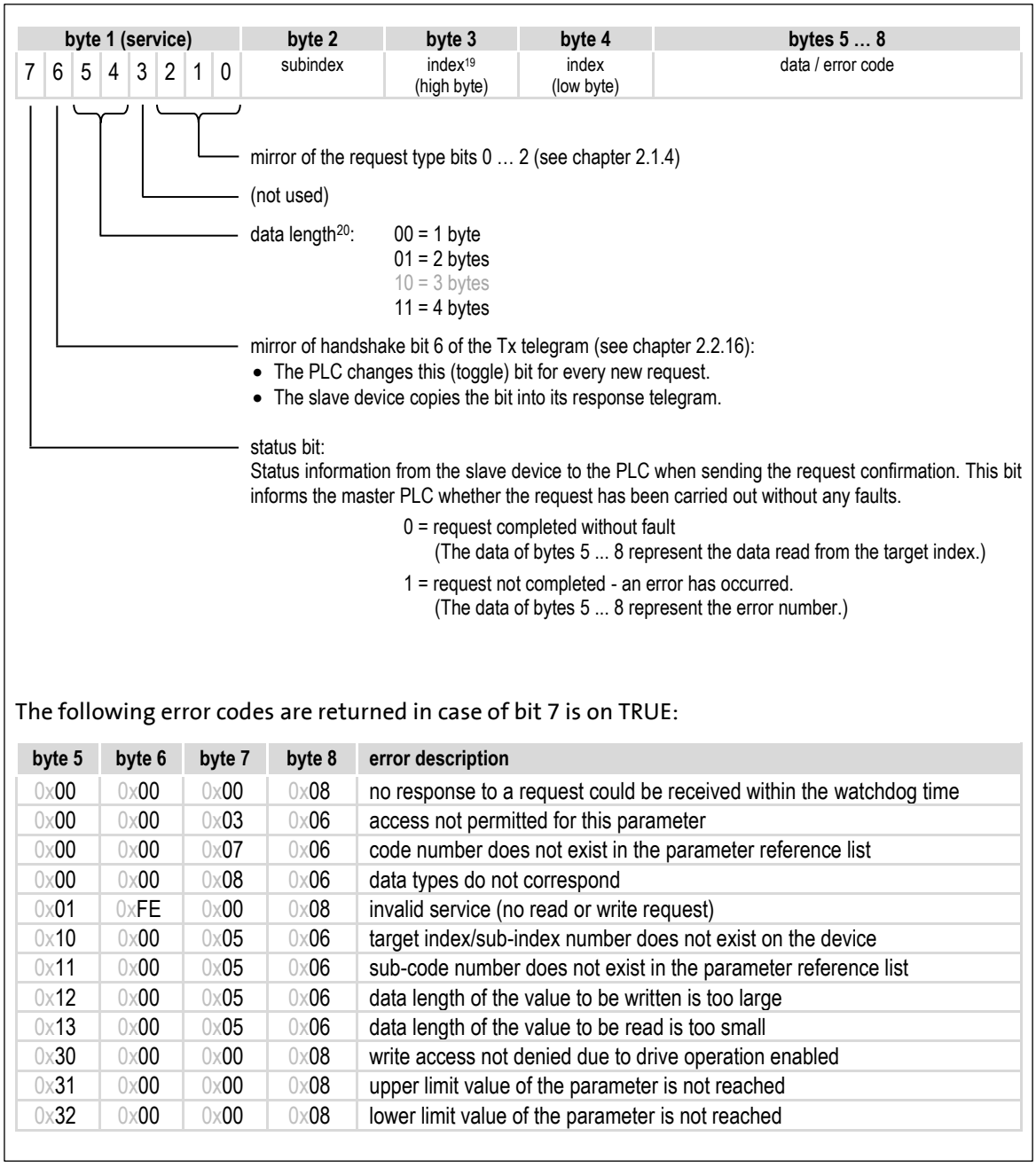



Illustration 48: structure of the Drivecom DP V0 parameter channel Tx telegram on the 9300 servo inverter (PLC => drive)

[17] The 9300 index number results from subtracting the 9300 code number from a fixed value of 24575 (=0x5FFF).
[18] length of data in bytes 5 ... 8 (data/error 1 ... 4) to be read/written to the slave device index

## 4.7 Drivecom DP V0 Parameter Channel (Rx)

The following chart describes the meaning of the receive parameter channel response (8 bytes), returned by the slave device (drive) to the PLC:




The following error codes are returned in case of bit 7 is on TRUE:

| byte 5 | byte 6 | byte 7 | byte 8 | error description |
|---|---|---|---|---|
| 0x00 | 0x00 | 0x00 | 0x08 | no response to a request could be received within the watchdog time |
| 0x00 | 0x00 | 0x03 | 0x06 | access not permitted for this parameter |
| 0x00 | 0x00 | 0x07 | 0x06 | code number does not exist in the parameter reference list |
| 0x00 | 0x00 | 0x08 | 0x06 | data types do not correspond |
| 0x01 | 0xFE | 0x00 | 0x08 | invalid service (no read or write request) |
| 0x10 | 0x00 | 0x05 | 0x06 | target index/sub-index number does not exist on the device |
| 0x11 | 0x00 | 0x05 | 0x06 | sub-code number does not exist in the parameter reference list |
| 0x12 | 0x00 | 0x05 | 0x06 | data length of the value to be written is too large |
| 0x13 | 0x00 | 0x05 | 0x06 | data length of the value to be read is too small |
| 0x30 | 0x00 | 0x00 | 0x08 | write access not denied due to drive operation enabled |
| 0x31 | 0x00 | 0x00 | 0x08 | upper limit value of the parameter is not reached |
| 0x32 | 0x00 | 0x00 | 0x08 | lower limit value of the parameter is not reached |

Illustration 49: structure of the Drivecom DP V0 parameter channel Rx telegram on the 9300 servo inverter (drive => PLC)

[19] The 9300 index number results from subtracting the 9300 code number from a fixed value of 24575 (=0x5FFF).
[20] length of the return data in bytes 5 ... 8 (data/error 1 ... 4)