Function block library

AsynCom_9

for PLCnext Engineer

Documentation for
PHOENIX CONTACT function blocks
PHOENIX CONTACT GmbH Co. KG
Flachsmarktstrasse 8
D32825 Blomberg, Germany

This documentation is available in English only.

Table of Contents

1 General information

The function blocks of the AsynCom library are working as an interface between function blocks which are intended to communicate acyclically and bus system dependent firmware function blocks (PCP_CONNECT, PCP_READ, PCP_WRITE, PDI_READ, PDI_WRITE, RDREC, WRREC). The standard udtAsynCom exchange structure can be used to control acyclic writing or reading. The AsynCom function block, compatibly selected for the bus system, uses the information from the exchange structure and sets the communication by controlling the firmware function blocks internally.

A function block that uses the standard exchange structure can be used for Profinet, INTERBUS or Axioline. With the help of the AsynCom function blocks, the same programming can be used for the three bus systems.

In case of an error, the xError bit of the udtAsynCom structure is set. In addition, the dwDiagCodeConnect, dwDiagCodeRead, and dwDiagCodeWrite outputs provide further information on the error cause.

2 Change notes

Library version	Library build	PLCnext Engineer version	Change notes	Supported PLCs
9	20200330	>= 2020.0 LTS	• Hint for AXL SE IOL4 (1088132) added	AXC F 1152 (1151412) AXC F 2152 (2404267) AXC F 3152 (1069208)
9	20200316	>= 2020.0 LTS	• Example AsynCom_9_EXA_1027843 documented • Example AsynCom_9_EXA_2702373_PN documented	AXC F 1152 (1151412) AXC F 2152 (2404267) AXC F 3152 (1069208)
9	20200306	>= 2020.0 LTS	Revised .chm documentation error “No access to page”.	AXC F 1152 (1151412) AXC F 2152 (2404267) AXC F 3152 (1069208)
9	20200206	>= 2020.0 LTS	Released for 2020.0 LTS	AXC F 1152 (1151412) AXC F 2152 (2404267)
9	20190930	2019.0 LTS 2019.3 2019.6 2019.9	Released for PLCnext Engineer 2019.9	AXC F 2152 (2404267)
8	20190814	2019.0 LTS 2019.3 2019.6	AsynCom_IBS_1: New	AXC F 2152 (2404267)
7	20190717	2019.0 LTS 2019.3 2019.6	Adapted to PLCnext Engineer 2019.6	AXC F 2152 (2404267)
6	20190627	2019.0 LTS	Update of documentation	AXC F 2152 (2404267)

6	20190607	2019.0 LTS	Added AsynCom_6_EXA_2702373_PN example for IB IL SGI 2/P/EF-PAC (2702373)	AXC F 2152 (2404267)
6	20190221	2019.0 LTS	AsynCom_AXL_3: Supports sub slots	AXC F 2152 (2404267)
5	20190225	2019.0 LTS	Supports “Allow extended identifiers” = ON	AXC F 2152 (2404267)
5	20190218	2019.0 LTS	Adapted to PLCnext Engineer 2019.0 LTS	AXC F 2152 (2404267)
4	-	7.2.3	Adapted to PLCnext Engineer 7.2.3	AXC F 2152 (2404267)
3	-	-	ASYN_UDT_COM: xOccupied added.	AXC F 2152 (2404267)
2	-	-	• Does not need PCWEngineerAdaption anymore. • Without AsynCom_PN_Info(): Not supported by PC Worx Engineer.	AXC F 2152 (2404267)
1	-	-	• Converted from PC Worx 6	AXC F 2152 (2404267)

New version number: Functional changes of at least one function block

New build number: No functional changes, but changes in the MSI file (e.g. documentation update, additional examples)

3 Function blocks

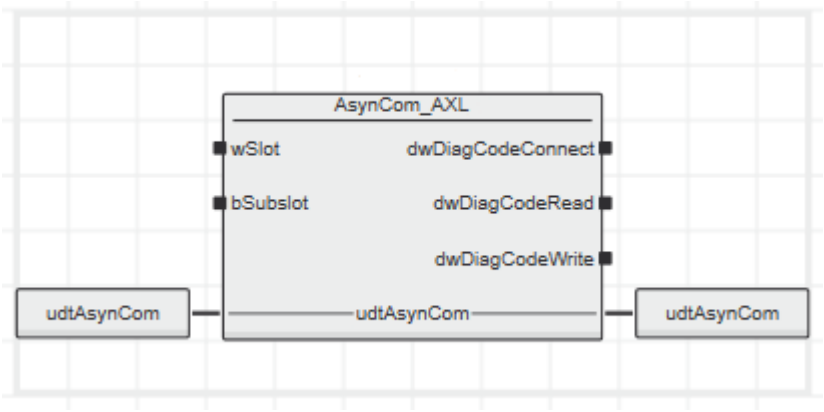
Function block	Description	Version	Supported articles	License
AsynCom_AXL	Function block for asynchronous communication with Axioline devices.	3	AXC F 2152 (2404267)	none
AsynCom_IBS	Function block for asynchronous communication with Inline devices.	1	AXC F 2152 (2404267)	none
AsynCom_PN	Function block for asynchronous communication with Profinet devices.	1	AXC F 2152 (2404267)	none
AsynCom_PN_Get_Idx	The function block searches for the I/O device (Bus coupler) index of the given Node ID in the arrDeviceAsynCom in/output.	1	AXC F 2152 (2404267)	none
AsynCom_F_2_NodeID	The function block searches for the Node ID of the given F-Destination-Address in the arrDeviceAsynCom in/output.	1	AXC F 2152 (2404267)	none

4 AsynCom_AXL

The AsynCom_AXL function block is to be used as a communication node between a system block and a PDI device in an Axioline configuration.

To do so, the udtAsynCom structure of the function block is connected to the system block. In the wSlot variable, the slot number of the device is written as an initial value. In the bus configuration, this value can be found on the left-hand side next to the module designation.

4.1 Function block call



4.2 Input parameters

Name	Type	Description
wSlot	WORD	Module number. The module number can be found in the bus configuration under the Axioline settings of the module. The wSlot parameter is adapted when the block is activated.
bSubslot	BYTE	Module sub number. The module sub number can be found in the bus configuration under the Axioline settings of the module. The bSubslot parameter is adapted when the block is activated.

4.3 Output parameters

Name	Type	Description
dwDiagCodeConnect	DWORD	No function with AXL
dwDiagCodeRead	DWORD	Error code of the PDI_READ firmware block or reading timeout
dwDiagCodeWrite	DWORD	Error code of the PDI_WRITE firmware block or writing timeout

4.4 Inout parameters

Name	Type	Description
udtAsynCom	ASYN_UDT_COM	Data exchange structure for asynchronous communication. Connected to AsynCom* function block.

4.5 Diagnosis

4.5.1 dwDiagCodeConnect

Function block output dwDiagCodeConnect and structure parameter udtAsynCom.udtConnect.dwDiagCodeConnect:

DiagCode	Description
16#00000010	Only udtAsynCom.dwDiagCodeConnect: wSlot function block input without content

4.5.2 dwDiagCodeRead

Function block output dwDiagCodeRead and structure parameter udtAsynCom.udtRead.dwDiagCodeRead:

DiagCode	Description
16#0000C411	Timeout while reading
16#xxxxxxxx	See error messages of the FW function block PDI_READ

4.5.3 dwDiagCodeWrite

Function block output dwDiagCodeWrite and structure parameter udtAsynCom.udtWrite.dwDiagCodeWrite:

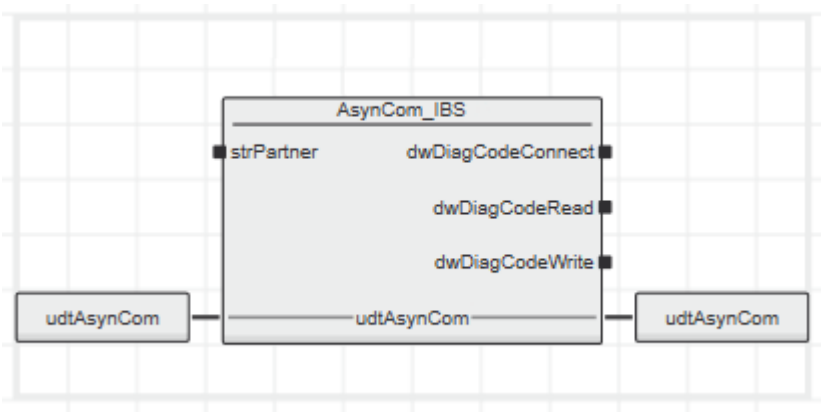
DiagCode	Description
16#0000C412	Timeout while writing
16#xxxxxxxx	See error messages of the FW function block PDI_WRITE

5 AsynCom_IBS

The AsynCom_IBS function block is to be used as a communication node between a system block and a PCP device in an INTERBUS environment.

To do so, the udtAsynCom structure of the function block is connected to the system block. In the strPartner variable, the communication reference value of the device is written as an initial value in the following format: “CRx”. Here, the x represents the respective communication reference. This value can be found in the device description for the respective module, in the INTERBUS settings point.

5.1 Function block call



5.2 Input parameters

Name	Type	Description
strPartner	STRING	Communication reference (CR) of the module. For example “CR2”. The communication reference can be found in the bus configuration for the properties of the INTERBUS device. Do not change strPartner after cold start.

5.3 Output parameters

Name	Type	Description
dwDiagCodeConnect	DWORD	Error code of the PCP_CONNECT firmware block or timeout during connection establishment
dwDiagCodeRead	DWORD	Error code of the PCP_READ firmware block or reading timeout
dwDiagCodeWrite	DWORD	Error code of the PCP_WRITE firmware block or writing timeout

5.4 Inout parameters

Name	Type	Description
udtAsynCom	ASYN_UDT_COM	Data exchange structure for asynchronous communication. Connected to AsynCom* function block.

5.5 Diagnosis

5.5.1 dwDiagCodeConnect

Function block output dwDiagCodeConnect and structure parameter udtAsynCom.udtConnect.dwDiagCodeConnect:

DiagCode	Description
16#00000010	FB input strPartner without content
16#0000C410	Timeout while connecting.
16#xxxxxxxx	See error messages of the FW function block PCP_CONNECT

5.5.2 dwDiagCodeRead

Function block output dwDiagCodeRead and structure parameter udtAsynCom.udtRead.dwDiagCodeRead:

DiagCode	Description
16#0000C411	Timeout while reading.
16#xxxxxxxx	See error messages of the FW function block PCP_READ.

5.5.3 dwDiagCodeWrite

Function block output dwDiagCodeWrite and structure parameter udtAsynCom.udtWrite.dwDiagCodeWrite:

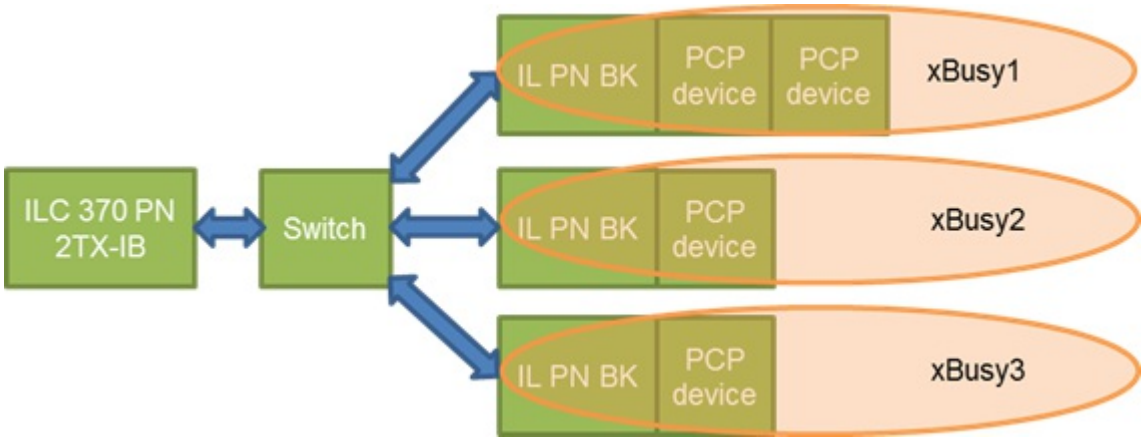
DiagCode	Description
16#0000C412	Timeout while writing.
16#xxxxxxxx	See error messages of the FW function block PCP_WRITE.

6 AsynCom_PN

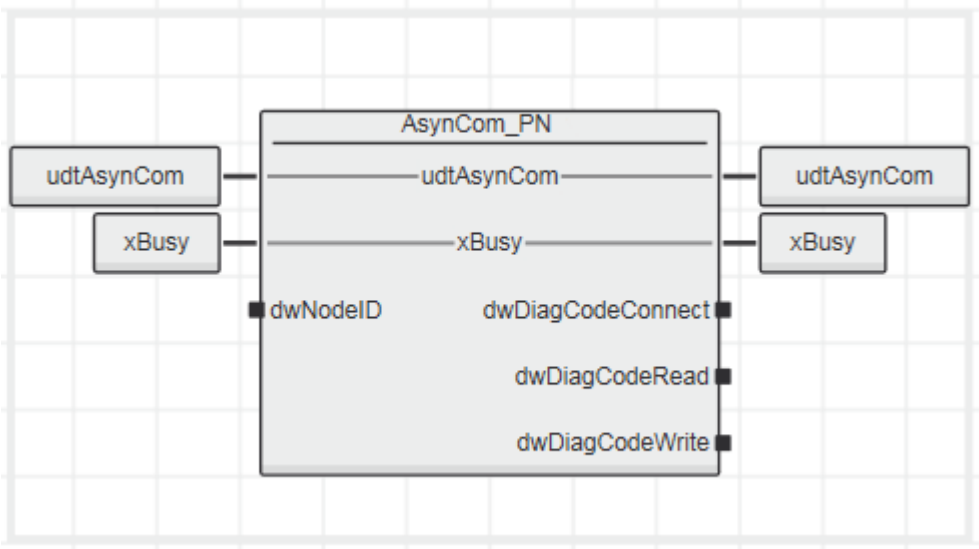
The AsynCom_PN function block is to be used as a communication node between a system block and a PCP device in a Profinet environment.

To do so, the udtAsynCom structure of the function block is connected to the system block. In the dwNodeID variable, the value of the node ID of the device is written as an initial value, with which the system block should communicate. This value can be found in the device description for the respective module, in the INTERBUS settings point.

The xBusy input and output has to be wired with one variable. If several AsynCom_PN blocks address the modules on a PN bus coupler, make sure that the same variable is used for these AsynCom_PN blocks. This prevents that several acyclic requests are sent to a bus coupler at the same time.



6.1 Function block call



6.2 Input parameters

Name	Type	Description
dwNodeID	DWORD	Node ID of the module. See module settings in the bus configuration. The Node ID is adopted when the block is activated.

6.3 Output parameters

Name	Type	Description
dwDiagCodeConnect	DWORD	No function at PN
dwDiagCodeRead	DWORD	Error code of the RDREC firmware module or reading timeout
dwDiagCodeWrite	DWORD	Error code of the WRREC firmware module or writing timeout

6.4 Inout parameters

Name	Type	Description
udtAsynCom	ASYN_UDT_COM	Data exchange structure for asynchronous communication. Connected to AsynCom* function block.
xBusy	BOOL	The variable is TRUE, as long as a communication block sends or receives data.

6.5 Diagnosis

6.5.1 dwDiagCodeConnect

Function block output dwDiagCodeConnect and structure parameter udtAsynCom.udtConnect.dwDiagCodeConnect:

DiagCode	Description
16#00000010	Only udtAsynCom.dwDiagCodeConnect: Node ID function block input without content

6.5.2 dwDiagCodeRead

Function block output dwDiagCodeRead and structure parameter udtAsynCom.udtRead.dwDiagCodeRead:

DiagCode	Description
16#0000C411	Timeout while writing
16#xxxxxxxx	See error messages of the FW function block RDREC

6.5.3 dwDiagCodeWrite

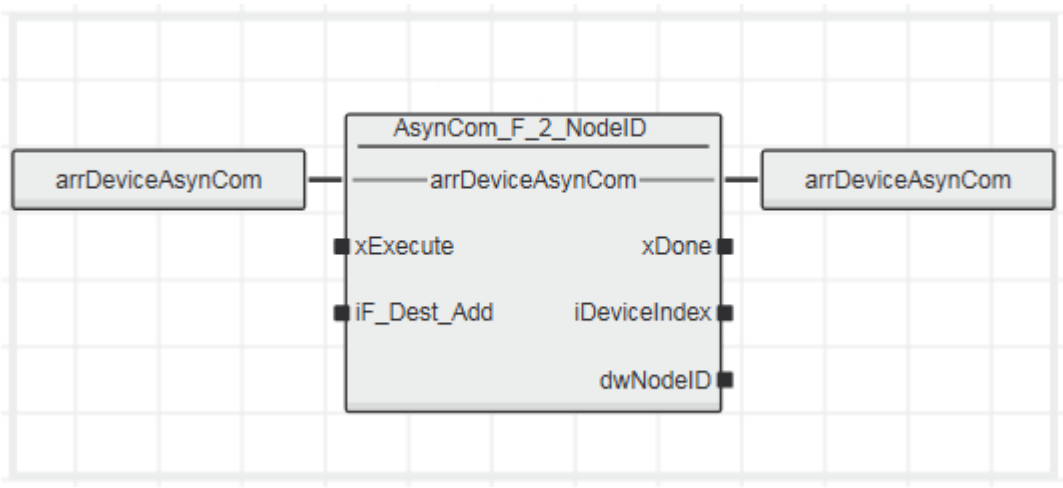
Function block output dwDiagCodeWrite and structure parameter udtAsynCom.udtWrite.dwDiagCodeWrite:

DiagCode	Description
16#0000C412	Timeout while writing
16#xxxxxxxx	See error messages of the FW function block WRREC

7 AsynCom_F_2_NodeID

The function block searches for the Node ID of the given F-Destination-Address in the arrDeviceAsynCom in/output.

7.1 Function block call



7.2 Input parameters

Name	Type	Description
xExecute	BOOL	Execute the function block at a rising edge.
iF_Dest_Add	INT	F-Destination address.

7.3 Output parameters

Name	Type	Description
xDone	BOOL	Function block executed
iDeviceIndex	INT	I/O Device (Bus coupler)-Index.
dwNodeID	DWORD	Node ID of the PROFINET slot.

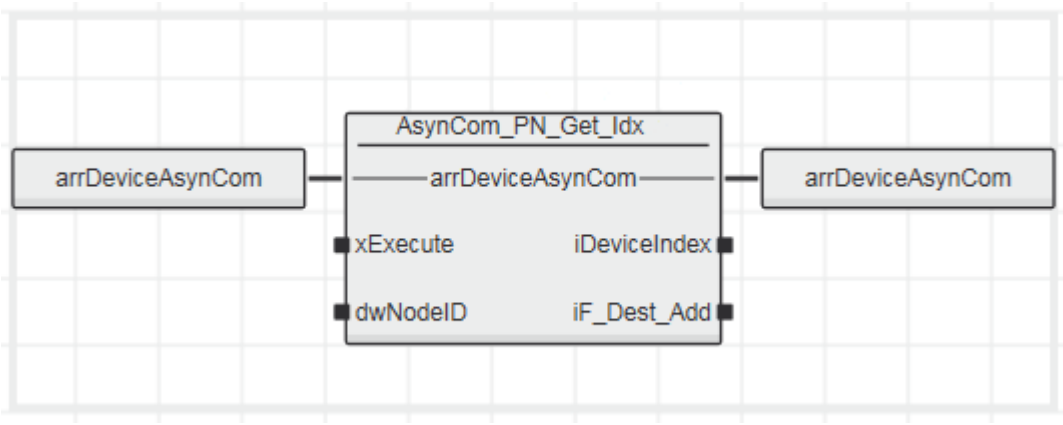
7.4 Inout parameters

Name	Type	Description
arrDeviceAsynCom	ASYN_ARR_DEVICES_PN_0_255	Array of structure containing information about all PROFINET devices.

8 AsynCom_PN_Get_Idx

The function block searchs for the I/O Device (bus coupler) index of the given Node ID in the arrDeviceAsynCom in/output. The block searches also for the iF-Dest_Address, if the Node ID has one.

8.1 Function block call



8.2 Input parameters

Name	Type	Description
xExecute	BOOL	Execute the function block at a rising edge.
dwNodeID	DWORD	Node ID of the PROFINET slot.

8.3 Output parameters

Name	Type	Description
iDeviceIndex	INT	I/O Device (bus coupler)-Index
iF_Dest_Add	INT	F-Destination address 0: no F-Destination address is assigned for this Node ID

8.4 Inout parameters

Name	Type	Description
arrDeviceAsynCom	ASYN_ARR_DEVICES_PN_0_255	Array of structure containing information about all PROFINET devices.

9 Examples

For the usage instruction of the AsynCom libraby please find the following examples:

- AsynCom_9_EXA_1027843
- AsynCom_9_EXA_2702373_PN
- AsynCom_9_EXA_Execute
- AsynCom_9_EXA_Activate

These examples are located in the “Examples” folder of the unzipped msi file of the library.

They describe how function blocks with xActivate and xExecute have to handle the udtAsynCom when using AsynCom for communication.

9.1 Example AsynCom_9_EXA_1027843

This example shows how to use AsynCom with AXL F IOL8 2H (1027843) as IO link master and AXL E IOL DO8 M12 6P (2702659) as 1. IO link device in an axioline local bus system

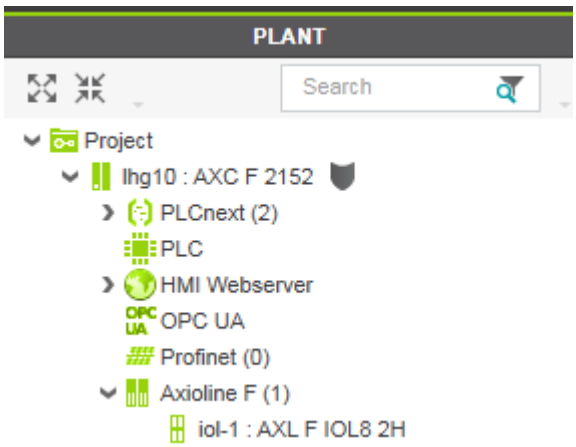
This example is also suitable for AXL SE IOL4 (1088132).

9.1.1 Usage

- Create system as descibed in the bus structure of this project
- Connect AXL E IOL DO8 M12 6P - 2702659 as 1. IO link device to AXL_F_IOL8_2H
- Compile and download to your PLC
- Set Main.xStart = TRUE
- Demo ends with OK if Main.iState = 32000

9.1.2 Bus structure

For this example, the following hardware is used:



9.1.3 Example machine

The example has a function block named ExampleMachine. It is connected to the device under test. ExampleMachine contains a state machine that shows which variables have to be set and to be checked in which order.

```
IF (xStart = TRUE) THEN
    CASE iState OF
    0: (* Init *)
        bSubslot := BYTE#1; (* 1. IO link device = AXL E IOL DO8 M12 6P - 2702659 *)
        (* Init udtAsynCom struct *)
        udtAsynCom.xActivate            := FALSE;
        udtAsynCom.xAutoConfirm         := FALSE;
        udtAsynCom.xEnableErrorMsg      := FALSE;

        udtAsynCom.udtRead.xReq         := FALSE;
        udtAsynCom.udtRead.iMlen        := 0;
        udtAsynCom.udtRead.wIndex       := WORD#0;
        udtAsynCom.udtRead.wSubIndex    := WORD#0;

        udtAsynCom.udtWrite.xReq        := FALSE;
        udtAsynCom.udtWrite.iLen        := 0;
        udtAsynCom.udtWrite.wIndex      := WORD#0;
        udtAsynCom.udtWrite.wSubIndex   := WORD#0;
        iState := 10;

    10: (* Activate *)
        udtAsynCom.xActivate := TRUE;
        IF (
            udtAsynCom.xActive = TRUE
            AND udtAsynCom.xError = FALSE
        ) THEN
            iState := 20;
        END_IF;

    20:
        (*
        ** Read wIndex = WORD#16#0013 = Order number
        ** from 1. IO link device = AXL E IOL DO8 M12 6P - 2702659
        *)
        udtAsynCom.udtRead.wIndex       := WORD#16#0013; (* Order number *)
        udtAsynCom.udtRead.wSubIndex    := WORD#0;
        udtAsynCom.udtRead.iMLen        := 8;
        udtAsynCom.udtRead.xReq         := TRUE;
        IF (
            udtAsynCom.udtRead.xDone = TRUE
            (* Order number data are read to receive buffer *)
            AND udtAsynCom.udtRead.arrData[1] = BYTE#16#32 (* '2' *)
            AND udtAsynCom.udtRead.arrData[2] = BYTE#16#37 (* '7' *)
            AND udtAsynCom.udtRead.arrData[3] = BYTE#16#30 (* '0' *)
            AND udtAsynCom.udtRead.arrData[4] = BYTE#16#32 (* '2' *)
            AND udtAsynCom.udtRead.arrData[5] = BYTE#16#36 (* '6' *)
            AND udtAsynCom.udtRead.arrData[6] = BYTE#16#35 (* '5' *)
            AND udtAsynCom.udtRead.arrData[7] = BYTE#16#39 (* '9' *)
        ) THEN
            udtAsynCom.udtRead.xReq := FALSE;
            iState := 30;
        END_IF;

    30:
        (*
        ** Write wIndex = WORD#16#0101 = Reset code
        ** to 1. IO link device = AXL E IOL DO8 M12 6P - 2702659
        *)
        udtAsynCom.udtWrite.wIndex      := WORD#16#0101; (* Reset code *)
        udtAsynCom.udtWrite.wSubIndex   := WORD#0;
        udtAsynCom.udtWrite.iLen        := 2; (* 2 Byte *)
        udtAsynCom.udtWrite.arrData[1]  := BYTE#16#00; (* Reset code value *)
        udtAsynCom.udtWrite.arrData[2]  := BYTE#16#01; (* Reset code value *)
        udtAsynCom.udtWrite.xReq        := TRUE;
```

```
        IF (
            udtAsynCom.udtWrite.xDone = TRUE
        ) THEN
            iState := 40;
        END_IF;

    40:
        (*
        ** Read back wIndex = WORD#16#0101 = Reset code
        ** from 1. IO link device = AXL E IOL DO8 M12 6P - 2702659
        *)
        udtAsynCom.udtRead.wIndex       := WORD#16#0101; (* Reset code *)
        udtAsynCom.udtRead.wSubIndex    := WORD#0;
        udtAsynCom.udtRead.iMLen        := 2;
        udtAsynCom.udtRead.xReq         := TRUE;
        IF (
            udtAsynCom.udtRead.xDone = TRUE
            (* Order number data are read to receive buffer *)
            AND udtAsynCom.udtRead.arrData[1] = BYTE#16#00 (* Reset code *)
            AND udtAsynCom.udtRead.arrData[2] = BYTE#16#01 (* Reset code *)
        ) THEN
            udtAsynCom.udtRead.xReq := FALSE;
            iState := 32000;
        END_IF;

    32000: (* End of demo *)
        iState := 32000;
    END_CASE;
END_IF;
```

9.2 Example AsynCom_9_EXA_2702373_PN

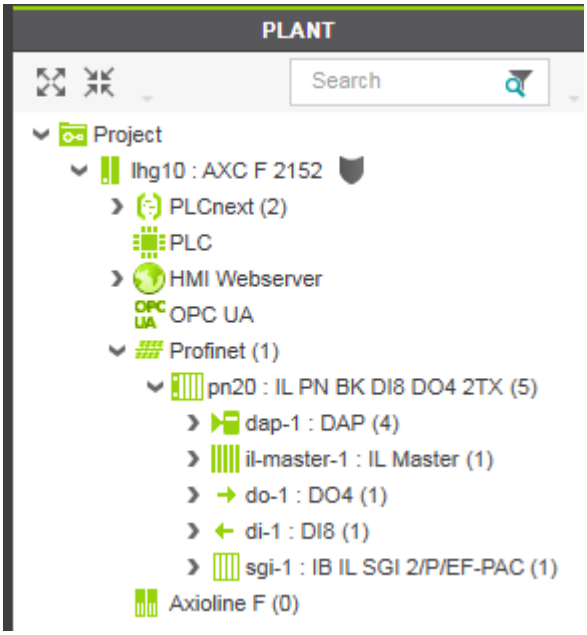
This example shows how to use “Used nominal characteristic value” on IB IL SGI 2/P/EF-PAC (2702373) in a PROFINET system

9.2.1 Usage

- Create system as described in the bus structure of this project
- Compile and download to your PLC
- Set Main.xStart = TRUE
- Demo ends with OK if Main.iState = 32000

9.2.2 Bus structure

For this example, the following hardware is used:



9.2.3 Example machine

The example has a function block named ExampleMachine. It is connected to the device under test. ExampleMachine contains a state machine that shows which variables have to be set and to be checked in which order.

```
IF(xStart = TRUE) THEN
    CASE iState OF
    0: (* Init *)
        iState := 100;

    (*
    ** STEP 1: Set "Used nominal characteristic value" with PCP
    *)
    100: (* Init AsynCom *)
        udtAsynCom.xActivate := TRUE;
        IF (udtAsynCom.xActive = TRUE) THEN
            iState := 110;
        END_IF;
```

```
110:
    (*
    ** AsynCom is activated
    ** Write Index. Refer to IB IL SGI 2/P/EF-PAC - 2702373 data sheet
    *)
    udtAsynCom.udtWrite.wIndex := WORD#16#A2; (* Used characteristic value *)
    udtAsynCom.udtWrite.wSubIndex := WORD#16#0000;
    udtAsynCom.udtWrite.iLen := 4; (* 2 * 2 = 4 Byte *)
    (* Channel 1: 10#7000 = 16#1B58 = 0.7 mV / V *)
    udtAsynCom.udtWrite.arrData[1] := BYTE#16#1B;
    udtAsynCom.udtWrite.arrData[2] := BYTE#16#58;
    (* Channel 2: 10#6000 = 16#1770 = 0.6 mV / V *)
    udtAsynCom.udtWrite.arrData[3] := BYTE#16#17;
    udtAsynCom.udtWrite.arrData[4] := BYTE#16#70;
    udtAsynCom.udtWrite.xReq := TRUE;
    IF (
        udtAsynCom.udtWrite.xDone = TRUE
        AND udtAsynCom.udtWrite.xError = FALSE
    ) THEN
        udtAsynCom.udtWrite.xReq := FALSE;
        iState := 200;
    END_IF;

(*
** STEP 2: Enable "Used nominal characteristic value" with process data (PD)
*)
200:
    wOut0 := WORD#16#4000;
    (* Channel 1 *)
    wOut1 := WORD#2#0000_0000_0000_1110;
    (* Channel 2 *)
    wOut2 := WORD#2#0000_0000_0000_1110;
    (*
    **              0
    **               0
    **                0 = N/O contact control via process data
    **                 0
    **                   0 = Advanced open circuit detection
    **                    000 = Mean-value
    **                        0000 = Conversion time
    **                             1110 = Specification via "Used nominal
    **                                    characteristic value" object
    *)
    IF (
        wIn0 = wOut0
        AND wIn1 = wOut1
        AND wIn2 = wOut2
    ) THEN
        iState := 210;
    END_IF;

210: (* Switch output process data to read values *)
    wOut0 := WORD#16#0000;
    wOut1 := WORD#16#0000;
    wOut2 := WORD#16#0000;
    iState := 300;

(*
** STEP 3: Read back "Used nominal characteristic value" with PCP for check
*)
300:
    udtAsynCom.udtRead.wIndex := WORD#16#00A2;
    udtAsynCom.udtRead.wSubIndex := WORD#16#0000;
    udtAsynCom.udtRead.iMlen := 4;
```

```
        udtAsynCom.udtRead.xReq := TRUE;
        IF (
            udtAsynCom.udtRead.xDone = TRUE
            AND udtAsynCom.udtRead.xError = FALSE
            (* Channel 1 *)
            AND udtAsynCom.udtRead.arrData[1] = BYTE#16#1B
            AND udtAsynCom.udtRead.arrData[2] = BYTE#16#58
            (* Channel 2 *)
            AND udtAsynCom.udtRead.arrData[3] = BYTE#16#17
            AND udtAsynCom.udtRead.arrData[4] = BYTE#16#70
        ) THEN
            iState := 32000;
        END_IF;

    32000: (* End of demo *)
        iState := 32000;

    END_CASE;
END_IF;
```

9.3 Example Activate

9.3.1 Bus structure

For this example, the following hardware is used:

- AXC F 2152 (2404267)
- AXL AI8 ME (2688187)

When using the AsynCom library for asynchronous communication in a function block with the xActivate input, we have to check if the function block is active after we activate it. For this we check the state of the structure variable udtAsynCom.xActive.

It is important that xActivate and xActive have the same state, else the function block is still in activation or deactivation phase. So when xActivate and xActive are TRUE, we can take the udtAsynCom.xOccupied and set our local variable xOccupiedOwner to TRUE so we know that we own the structure for that moment. If both xActivate and xActive are FALSE, we set xActivate to TRUE, take the udtAsynCom.xOccupied and set our local variable xOccupiedOwner to TRUE. Then we wait for xActive = TRUE.

If the state of both variables is different, we just wait until they are both FALSE or TRUE.

```
IF xActivate THEN

    udtDiag.wDiagCode := WORD#16#8000;
    udtDiag.wAddDiagCode := WORD#16#0000;

    IF udtAsynCom.xActivate AND udtAsynCom.xActive THEN

        xActive := TRUE;

        IF udtAsynCom.xOccupied AND NOT xOccupiedOwner THEN

            udtDiag.wDiagCode := WORD#16#8500;
            udtDiag.wAddDiagCode := WORD#16#0001;

        ELSIF (NOT udtAsynCom.xOccupied) OR (udtAsynCom.xOccupied AND xOccupiedOwn

            udtAsynCom.xOccupied := TRUE;
            xOccupiedOwner := TRUE;
            udtDiag.wDiagCode := WORD#16#8000;
            udtDiag.wAddDiagCode := WORD#16#0000;
            udtDiag.iState := 10;

        END_IF;

    ELSIF (NOT udtAsynCom.xActivate) AND (NOT udtAsynCom.xActive) THEN

        udtAsynCom.xActivate := TRUE;
        udtAsynCom.xOccupied := TRUE;
        xOccupiedOwner := TRUE;

    END_IF;

ELSE

    udtDiag.wDiagCode := WORD#16#0000;
    udtDiag.wAddDiagCode := WORD#16#0000;
    xActive := FALSE;

END_IF;
```

Then we can set the index, the subindex and the length of the object we want to read. After that, we have to set the xReq to TRUE.

```
udtAsynCom.udtRead.wIndex := WORD#16#0082;
udtAsynCom.udtRead.wSubIndex := WORD#16#0001;
udtAsynCom.udtRead.iMLen := 8;

udtAsynCom.udtRead.xReq := TRUE;
```

Then we have to wait till the AsynCom function block tells us that the reading was successfull or that an error occured. So we wait for the xDone or the xError to become TRUE.

```
IF udtAsynCom.udtRead.xDone THEN

    udtAsynCom.udtRead.wIndex := WORD#16#0000;
    udtAsynCom.udtRead.wSubIndex := WORD#16#0000;
    udtAsynCom.udtRead.iMLen := 0;

    udtAsynCom.udtRead.xReq := FALSE;

    udtAsynCom.xOccupied := FALSE;
    xOccupiedOwner := FALSE;

    (**)
    rValue := IEEE754_TO_REAL_1(
    udtAsynCom.udtRead.arrData[1],
    udtAsynCom.udtRead.arrData[2],
    udtAsynCom.udtRead.arrData[3],
    udtAsynCom.udtRead.arrData[4]);

    xDone := TRUE;

    xTON_Activate := TRUE;

    udtDiag.iState := 40;

ELSIF udtAsynCom.xError THEN

    udtAsynCom.udtRead.wIndex := WORD#16#0000;
    udtAsynCom.udtRead.wSubIndex := WORD#16#0000;
    udtAsynCom.udtRead.iLen := 0;

    udtAsynCom.udtRead.xReq := FALSE;

    xError := TRUE;

    udtDiag.wDiagCode := WORD#16#C510;
    udtDiag.wAddDiagCode := WORD#16#0001;

    udtDiag.iState := 999;

END_IF;
```

When xDone is TRUE, we can do whatever we need with the data in udtRead and set the xOccupiedOwner and the xOccupied in the udtAsynCom structure to FALSE because we do not need it anymore. We do not need to deactivate the AsynCom function block.

When xError is TRUE, we go in a case that is only for the error handling. There we stay and do nothing until the xActivate of our function block becomes FALSE. Then we set xActivate of the struct to FALSE and wait until xActive is FALSE too. Then we release the xOccupied and the xOccupiedOwner and go back in our state 0.

```
    xDone := FALSE;

    IF xTON_Finished THEN

        xTON_Activate := FALSE;
        udtDiag.iState := 0;

    END_IF;

    IF NOT xActivate THEN

        xActive := FALSE;
        udtDiag.wDiagCode := WORD#16#0000;
        udtDiag.wAddDiagCode := WORD#16#0000;
        udtDiag.iState := 0;

    END_IF;

999 : (**)
    IF NOT xActivate THEN

        udtAsynCom.xActivate := FALSE;
        udtDiag.iState := 1000;

    END_IF;

1000 : (**)
    IF NOT udtAsynCom.xActive THEN

        udtAsynCom.udtTest.udtWrite.xError := FALSE;
        udtAsynCom.udtTest.udtRead.xError := FALSE;

        udtAsynCom.xOccupied := FALSE;
        xOccupiedOwner := FALSE;
        xError := FALSE;
        xActive := FALSE;
        udtDiag.wDiagCode := WORD#16#0000;
        udtDiag.wAddDiagCode := WORD#16#0000;

        udtDiag.iState := 0;

    END_IF;
```

9.4 Example Execute

9.4.1 Bus structure

For this example, the following hardware is used:

- AXC F 2152 (2404267)
- AXL AI8 ME (2688187)

When using the AsynCom library for asynchronous communication in a function block with the xExecute input, we have to check if the function block is active after we activate it. For this we check the state of the structure variable udtAsynCom.xActive.

It is important that xActivate and xActive have the same state, else the function block is still in activation or deactivation phase. So when xActivate and xActive are TRUE, we can take the udtAsynCom.xOccupied and set our local variable xOccupiedOwner to TRUE so we know that we own the structure for that moment. If both xActivate and xActive are FALSE, we set xActivate to TRUE, take the udtAsynCom.xOccupied and set our local variable xOccupiedOwner to TRUE. Then we wait for xActive = TRUE.

If the state of both variables is different, we just wait until they are both FALSE or TRUE.

```
    IF xExecute AND xReady THEN
        xReady := FALSE;
        udtDiag.wDiagCode := WORD#16#8000;
        udtDiag.wAddDiagCode := WORD#16#0000;
        strReadLocation := '';
        udtDiag.iState := 10;
    END_IF;

10 : (**)
    IF udtAsynCom.xActivate AND udtAsynCom.xActive THEN

        IF udtAsynCom.xOccupied AND NOT xOccupiedOwner THEN

            udtDiag.wDiagCode := WORD#16#8500;
            udtDiag.wAddDiagCode := WORD#16#0001;

        ELSIF (NOT udtAsynCom.xOccupied) OR (udtAsynCom.xOccupied AND xOccupiedOwne

            udtAsynCom.xOccupied := TRUE;
            xOccupiedOwner := TRUE;
            udtDiag.wDiagCode := WORD#16#8000;
            udtDiag.wAddDiagCode := WORD#16#0000;
            udtDiag.iState := 20;

        END_IF;

    ELSIF (NOT udtAsynCom.xActivate) AND (NOT udtAsynCom.xActive) THEN

        udtAsynCom.xActivate := TRUE;
        udtAsynCom.xOccupied := TRUE;
        xOccupiedOwner := TRUE;

    END_IF;
```

Then we can set the index, the subindex ,the length and the data of the object we want to write. After that, we have to set the xReq to TRUE.

```
udtAsynCom.udtWrite.wIndex := WORD#16#0014;
udtAsynCom.udtWrite.wSubIndex := WORD#16#0000;
udtAsynCom.udtWrite.iLen := LEN(strLocation);

xRes := STRING_COPY(strLocation,udtAsynCom.udtWrite.arrData,UDINT#0, TO_UDINT(udtA

udtAsynCom.udtWrite.xReq := TRUE;
```

Then we have to wait till the AsynCom function block tells us that the writing was successfull or that an error occured. So we wait for the xDone or the xError to become TRUE.

```
IF udtAsynCom.udtWrite.xDone THEN

    udtAsynCom.udtWrite.wIndex := WORD#16#0000;
    udtAsynCom.udtWrite.wSubIndex := WORD#16#0000;
    udtAsynCom.udtWrite.iLen := 0;

    udtAsynCom.udtWrite.xReq := FALSE;

    udtDiag.iState := 40;

ELSIF udtAsynCom.xError THEN

    udtAsynCom.udtWrite.wIndex := WORD#16#0000;
    udtAsynCom.udtWrite.wSubIndex := WORD#16#0000;
    udtAsynCom.udtWrite.iLen := 0;

    udtAsynCom.udtWrite.xReq := FALSE;

    xError := TRUE;

    udtDiag.wDiagCode := WORD#16#C510;
    udtDiag.wAddDiagCode := WORD#16#0001;

    udtDiag.iState := 999;

END_IF;
```

When xDone is TRUE, we can read the same index to check if our writing was really successfull. Then we can set the xOccupiedOwner and the xOccupied in the udtAsynCom structure to FALSE because we do not need it anymore. We do not need to deactivate the AsynCom function block.

```
IF udtAsynCom.udtRead.xDone THEN

    udtAsynCom.udtRead.wIndex := WORD#16#0000;
    udtAsynCom.udtRead.wSubIndex := WORD#16#0000;
    udtAsynCom.udtRead.iMLen := 0;

    udtAsynCom.udtRead.xReq := FALSE;

    udtAsynCom.xOccupied := FALSE;
    xOccupiedOwner := FALSE;
```

When xError is TRUE, we go in a case that is only for the error handling. There we stay and do nothing until the xExecute of our function block becomes FALSE. Then we set xActivate of the struct to FALSE and wait until xActive is FALSE too. Then we release the xOccupied and the xOccupiedOwner and go back in our state 0.

```
        999 : (**)
            IF NOT xExecute THEN

                udtAsynCom.xActivate := FALSE;
                udtDiag.iState := 1000;

            END_IF;

        1000 : (**)
            IF NOT udtAsynCom.xActive THEN

                udtAsynCom.udtTest.udtWrite.xError := FALSE;
                udtAsynCom.udtTest.udtRead.xError := FALSE;

                udtAsynCom.xOccupied := FALSE;
                xOccupiedOwner := FALSE;
                xError := FALSE;
                xReady := TRUE;
                udtDiag.wDiagCode := WORD#16#0000;
                udtDiag.wAddDiagCode := WORD#16#0000;

                udtDiag.iState := 0;

            END IF;
```

10 Appendix

10.1 Diag codes of used firmware function blocks

10.1.1 PDI_READ

for PLCnext Engineer

ERROR = TRUE

STATUS[0]	STATUS[1]	Meaning
16#09B0	16#000C	The variable connected to RD_1 is invalid (no array or invalid array type).
16#09B0	16#000B	The array connected to RD_1 is too small to save the requested receive data.
16#09B0	16#000E	Timeout. No response to the sent PDI READ request received.
16#09B0	16#000F	An internal error has occurred.

When receiving a negative confirmation as response to a PDI_READ request, the Axioline module directly copies the received error code (Error_Code and Add_Info) to STATUS[0] or STATUS[1]. These error codes are module-specific. For a description see the respective module documentation.

10.1.2 PDI_WRITE

for PLCnext Engineer

ERROR = TRUE

STATUS[0]	STATUS[1]	Meaning
16#09B0	16#000A	The variable connected to SD_1 is invalid (no array or invalid array type).
16#09B0	16#0009	Invalid value at DATA_CNT input. The value is either greater than the array connected to SD_1, greater than the maximum allowed length (245 bytes) or equal to zero.
16#09B0	16#000E	Timeout. No response to the sent PDI WRITE request received.
16#09B0	16#000F	An internal error has occurred.

When receiving a negative confirmation as response to a PDI_WRITE request, the Axioline module directly copies the received error code (Error_Code and Add_Info) to STATUS[0] or STATUS[1]. These error codes are module-specific. For a description see the respective module documentation.

10.1.3 RDREC

for PLCnext Engineer

Error code (hex)	Meaning
16#0000	No error occurred.
16#F001	Too many instances used.
16#F002	Error during initialization of the function block.
16#F003	Invalid ID.
16#F004	Invalid HANDLE/ID.
16#F005	Resources conflict.
16#F006	A function block internal task could not be generated.
16#F007	Too many instances used.
16#F008	Invalid type of a parameter.
16#F009	Invalid parameter value.
16#F00A	Unallowed parameter.
16#F00B	Invalid length specified.
16#F00C	ID could not be created (too many IDs).
16#F00D	No entry found that matches the specified ID.
16#F00F	No further entries found.
16#F010	Entry in use.
16#F011	Alarm acknowledgement could not be done.
16#F012	Error reading the AR parameters (1st time).
16#F013	Negative acknowledgement received for the execution of a PROFINET service.
16#F014	Invalid length for parameter LEN/MLEN or/and RECORD data record too short.
16#F015	The service used to read the RECORD data record could not be run.
16#F016	The service used to write the RECORD data record could not be run.
16#F017	Service acknowledgement not received.
16#F018	Invalid INDEX used to access the RECORD data record of the IO device, for example, INDEX greater than 16#7FFF.
16#F019	Unknown command code.
16#F01A	Error starting the Application Relation (AR).
16#F01B	Error stopping the Application Relation (AR).
16#F01C	Notification of stopped Application Relation (AR) failed.
16#F01D	Setting the "Drive BF" flag failed.
16#F01E	Error reading the AR parameters (2nd time).

10.1.4 WRREC

for PLCnext Engineer

Error code (hex)	Meaning
16#0000	No error occurred.
16#F001	Too many instances used.
16#F002	Error during initialization of the function block.
16#F003	Invalid ID.
16#F004	Invalid HANDLE/ID.
16#F005	Resources conflict.
16#F006	A function block internal task could not be generated.
16#F007	Too many instances used.
16#F008	Invalid type of a parameter.
16#F009	Invalid parameter value.
16#F00A	Unallowed parameter.
16#F00B	Invalid length specified.
16#F00C	ID could not be created (too many IDs).
16#F00D	No entry found that matches the specified ID.
16#F00F	No further entries found.
16#F010	Entry in use.
16#F011	Alarm acknowledgement could not be done.
16#F012	Error reading the AR parameters (1st time).
16#F013	Negative acknowledgement received for the execution of a PROFINET service.
16#F014	Invalid length for parameter LEN/MLEN or/and RECORD data record too short.
16#F015	The service used to read the RECORD data record could not be run.
16#F016	The service used to write the RECORD data record could not be run.
16#F017	Service acknowledgement not received.
16#F018	Invalid INDEX used to access the RECORD data record of the IO device, for example, INDEX greater than 16#7FFF.
16#F019	Unknown command code.
16#F01A	Error starting the Application Relation (AR).
16#F01B	Error stopping the Application Relation (AR).
16#F01C	Notification of stopped Application Relation (AR) failed.
16#F01D	Setting the "Drive BF" flag failed.
16#F01E	Error reading the AR parameters (2nd time).

10.1.5 FILE_OPEN

ERRORID	Description
0	No error information available.
2	The maximum number of files is already opened.
4	The file is already opened.
5	File is write protected or access denied.
6	File name not defined.

10.1.6 FILE_SEEK

ERRORID	Description
0	No error information available.
1	Invalid file handle.
13	Invalid positioning mode or position specified is before the beginning of the file.
24	Position could not be set.

10.1.7 FILE_READ

ERRORID	Description
0	No error information available.
1	Invalid file handle.
10	End of data reached.
12	The number of characters to be read is greater than the data buffer.
22	No data could be read.

10.1.8 FILE_CLOSE

ERRORID	Description
0	No error information available.
1	Invalid file handle.
20	File could not be closed.

10.2 Data types

The control of the AsynCom function blocks is mainly done via the udtAsynCom structure of type ASYN_UDT_COM and via the inputs of the blocks. The structure is explained in more detail below.

```
TYPE
    (* Array for the data to be send and read *)
    ASYN_ARR_DATA_B_1_512   : ARRAY[1..512] OF BYTE;

    (* Array for pcp communication *)
    ASYN_ARR_ERR_W_1_7      : ARRAY [1..7] OF WORD;
        (* ADD_ERROR of the PCP_connect *)
    ASYN_ARR_ERR_W_1_2      : ARRAY [1..2] OF WORD;
        (* ADD_ERROR of the PCP_READ and PCP_WRITE *)
    ASYN_ARR_STATUS_W_1_2   : ARRAY [1..2] OF WORD;
        (* STATUS of PDI_READ & PDI_WRITE *)
    ASYN_ARR_PCP_CMD_W_1_2  : ARRAY [1..2] OF WORD;
        (* PCP comand *)

    (* I/O parameter of firmware functions as struct *)
    ASYN_UDT_RDREC : STRUCT
        (* Inputs *)
        xReq        : BOOL;
        dwID        : DWORD;
        iIndex      : INT;
        iMLEN       : INT;
        (* Outputs *)
        xValid      : BOOL;
        xError      : BOOL;
        xBusy       : BOOL;
        iLEN        : INT;
        dwStatus    : DWORD;
    END_STRUCT;

    ASYN_UDT_WRREC : STRUCT
        (* Inputs *)
        xReq        : BOOL;
        dwID        : DWORD;
        iIndex      : INT;
        iLEN        : INT;
        (* Outputs *)
        xDone       : BOOL;
        xBusy       : BOOL;
        xError      : BOOL;
        dwStatus    : DWORD;
    END_STRUCT;

    ASYN_UDT_PDI_READ : STRUCT
        (* Inputs *)
        xReq        : BOOL;
        wSlot       : WORD;
        bSubslot    : BYTE;
        wIndex      : WORD;
        bSubIndex   : BYTE;
        (* Outputs *)
        bDataCNT    : BYTE;
        xNDR        : BOOL;
        xError      : BOOL;
        (* Inouts *)
        arrStatus   : ASYN_ARR_STATUS_W_1_2;
    END_STRUCT;

    ASYN_UDT_PDI_WRITE : STRUCT
```

```
    (* Inputs *)
    xReq        : BOOL;
    wSlot       : WORD;
    bSubslot    : BYTE;
    wIndex      : WORD;
    bSubIndex   : BYTE;
    bDataCNT    : BYTE;
    (* Outputs *)
    xDone       : BOOL;
    xError      : BOOL;
    (* INOUT *)
    arrStatus   : ASYN_ARR_STATUS_W_1_2;
END_STRUCT;

ASYN_UDT_PCP_CONNECT : STRUCT
    (* Inputs *)
    xEN_C       : BOOL;
    (* Outputs *)
    xValid      : BOOL;
    iID         : INT;
    xError      : BOOL;
    (* Inouts *)
    arrAddError : ASYN_ARR_ERR_W_1_7;
    strPartner  : STRING;
END_STRUCT;

ASYN_UDT_PCP_READ : STRUCT
    (* Inputs *)
    xReq        : BOOL;
    iID         : INT;
    (* Outputs *)
    xNDR        : BOOL;
    xError      : BOOL;
    iDataCnt    : INT;
    (* Inouts *)
    arrVar_1    : ASYN_ARR_PCP_CMD_W_1_2;
    arrAddError : ASYN_ARR_ERR_W_1_2;
END_STRUCT;

ASYN_UDT_PCP_WRITE : STRUCT
    (* Inputs *)
    xReq        : BOOL;
    iID         : INT;
    iDataCnt    : INT;
    (* Outputs *)
    xDone       : BOOL;
    xError      : BOOL;
    (* Inouts *)
    arrVar_1    : ASYN_ARR_PCP_CMD_W_1_2;
    arrAddError : ASYN_ARR_ERR_W_1_2;
END_STRUCT;

(* Definition of the diagnostic structure *)
ASYN_UDT_DIAG : STRUCT
    udtPDI_Read     : ASYN_UDT_PDI_READ;
    udtPDI_Write    : ASYN_UDT_PDI_WRITE;
    udtPCP_Connect  : ASYN_UDT_PCP_CONNECT;
    udtPCP_Read     : ASYN_UDT_PCP_READ;
    udtPCP_Write    : ASYN_UDT_PCP_WRITE;
    udtRDREC        : ASYN_UDT_RDREC;
    udtWRREC        : ASYN_UDT_WRREC;
END_STRUCT;

(* Definition of the test-structure to force FW FB outputs *)
```

```
ASYN_UDT_TEST_CONNECT : STRUCT
    xError      : BOOL; (* TRUE: force ERROR to TRUE *)
    xValid      : BOOL; (* TRUE: force VALID to FALSE *)
END_STRUCT;

ASYN_UDT_TEST_READ : STRUCT
    xError      : BOOL; (* TRUE: force ERROR to TRUE *)
    xValid      : BOOL; (* TRUE: force VALID to FALSE *)
    xNDR        : BOOL; (* TRUE: force NDR to FALSE *)
    dwStatus    : DWORD; (* dwStatus <> 0: force STATUS to value *)
END_STRUCT;

ASYN_UDT_TEST_WRITE : STRUCT
    xDone       : BOOL; (* TRUE: force DONE to FALSE *)
    xError      : BOOL; (* TRUE: force ERROR to TRUE *)
    dwStatus    : DWORD; (* dwStatus <> 0: force STATUS to value *)
END_STRUCT;

ASYN_UDT_TEST : STRUCT
    udtConnect  : ASYN_UDT_TEST_CONNECT;
    udtRead     : ASYN_UDT_TEST_READ;
    udtWrite    : ASYN_UDT_TEST_WRITE;
END_STRUCT;

(* Definition of the AsynCom exchange struct *)
(* Diagnostic from Connect *)
ASYN_UDT_CONNECT : STRUCT
    xError              : BOOL; (* TRUE: Connect error *)
    dwDiagCodeConnect   : DWORD;(* DiagCodeConnect *)
END_STRUCT;

(* Set of data for writing asynchron data *)
ASYN_UDT_WRITE : STRUCT
    xReq            : BOOL; (* TRUE: Trigger to send out the request *)
    xDone           : BOOL; (* TRUE: request has been processed *)
    xError          : BOOL; (* TRUE: Write error *)
    dwDiagCodeWrite : DWORD;(* DiagCodeWrite *)
    xBusy           : BOOL; (* TRUE: request is in progress *)
    wIndex          : WORD; (* Requested index *)
    wSubIndex       : WORD; (* Requested subindex (for profinet set to 0) *)
    iLen            : INT;  (* Number of bytes to be send *)
    arrData         : ASYN_ARR_DATA_B_1_512; (* User data to be send *)
END_STRUCT;

(* Set of data for reading asynchron data *)
ASYN_UDT_READ : STRUCT
    xReq            : BOOL; (* TRUE: trigger to send out the request *)
    xDone           : BOOL; (* TRUE: Request has been processed *)
    xBusy           : BOOL; (* TRUE: request is in progress *)
    xError          : BOOL; (* TRUE: read error *)
    dwDiagCodeRead  : DWORD;(* DiagCodeRead *)
    wIndex          : WORD; (* Requested index *)
    wSubIndex       : WORD; (* Requested subindex (for profinet set to 0) *)
    iLen            : INT;  (* Number of bytes which has been read *)
    iMlen           : INT;  (* Max. length of bytes *)
    arrData         : ASYN_ARR_DATA_B_1_512;
        (* User data which has been read *)
END_STRUCT;

(* set of data for the exchange with the asynchron communication block *)
ASYN_UDT_COM : STRUCT
        (* State of the internal statemachine state *)
    xActivate           : BOOL;
    xOccupied           : BOOL; (* TRUE: AsynCom FB is occupied *)
```

```
        (* Activates the function block *)
    xActive             : BOOL;
        (* TRUE: Function block is ready to operate;
        connection for example pcp has been established *)
    xError              : BOOL; (* TRUE: Internal error *)
    xAutoConfirm        : BOOL; (* TRUE: Internal restart in failure run *)
    xEnableErrorMsg     : BOOL;
        (* TRUE: Transmit error messages as DiagCodes *)
    xNewErrorMsg        : BOOL;
        (* TRUE: New error message is avalible for one cycle *)
    bBusSystem          : BYTE;
        (* 0: IL PROFINET 2: IL INTERBUS 3: AXL PROFINET*)
    iMaxRetriesPerMinute : INT;
        (* Maximum read or write retries per minute before error *)
    tTimeout            : TIME; (* Statemachine timeout time *)
    dwNodeID            : DWORD;(* ID of the Device *)
    dwDiagCodeConnect   : DWORD;(* DiagCodeConnect *)
    udtConnect          : ASYN_UDT_CONNECT;
    udtRead             : ASYN_UDT_READ;
    udtWrite            : ASYN_UDT_WRITE;
    udtTest             : ASYN_UDT_TEST;    (* Forcing of FW functions *)
    udtDiag             : ASYN_UDT_DIAG;    (* Diagnostic FW functions *)
    iState              : INT;
    iDummy              : INT;
END_STRUCT;

(* For AsynCom_PN_Info *)

(* Buffer for reading the file by block ASYN_NodeIDs_PN.
Adjust the dimension of this field together with variable
diMaxReadBuffer in sheet AsynCom_NodeIDs_PN / IPPNIO *)
ASYN_ARR_B_0_1000       : ARRAY [1..1000] OF BYTE;
(* String for the profinet device name *)
ASYN_STRING_255         : STRING(255);
(* List of node ids of one pnio device
(array index is equal to slot number) *)
ASYN_ARR_DEVICE_NODES_DW_0_63   : ARRAY [0..63] OF DWORD;
ASYN_ARR_DEVICE_NODES_I_0_63    : ARRAY [0..63] OF INT;

(* Structure for one device with list of node ids
and busy signal for asynchron communication *)
ASYN_UDT_DEVICE_PN : STRUCT
    strStatName         : ASYN_STRING_255;
    xDeviceAvailable    : BOOL;
        (* TRUE: PNIO Device is available in the bus structure *)
    iSlotNoMax          : INT;  (* Highest slot number in use *)
    arrNodeIDs          : ASYN_ARR_DEVICE_NODES_DW_0_63;
        (* List of node ids (slot number is equal to field index) *)
    arrFDestAddr        : ASYN_ARR_DEVICE_NODES_I_0_63;
        (*List of F destination addresses
        (slot number is equal to field index)*)
    xBusy               : BOOL;
        (* Busy signal for asynchron communication to this device *)
        iDummy          : INT;
END_STRUCT;

(* List of node ids of one pnio device
(array index is equal to slot number) *)
ASYN_ARR_DEVICES_PN_0_255       : ARRAY [0..255] OF ASYN_UDT_DEVICE_PN;

(* Structures for file I/O diag *)
ASYN_UDT_FILE_OPEN : STRUCT
    (* Inputs *)
    xExecute    : BOOL;
```

```
        (* Outputs *)
        xDone       : BOOL;
        xError      : BOOL;
        uiErrorID   : UINT;
    END_STRUCT;

    ASYN_UDT_FILE_SEEK : STRUCT
        (* Inputs *)
        xExecute    : BOOL;
        diPosition  : DINT;
        uiMode      : uINT;
        (* Outputs *)
        xDone       : BOOL;
        xError      : BOOL;
        uiErrorID   : UINT;
        iDummy      : INT;
    END_STRUCT;

    ASYN_UDT_FILE_READ : STRUCT
        (* Inputs *)
        xExecute        : BOOL;
        udiMaxLength    : UDINT;
        (* Outputs *)
        xDone           : BOOL;
        udiLengthRead   : UDINT;
        xError          : BOOL;
        uiErrorID       : UINT;
    END_STRUCT;

    ASYN_UDT_FILE_CLOSE : STRUCT
        (* Inputs *)
        xExecute    : BOOL;
        (* Outputs *)
        xDone       : BOOL;
        xError      : BOOL;
        uiErrorID   : UINT;
    END_STRUCT;

    ASYN_UDT_PN_INFO_DIAG : STRUCT
        iState      : INT;
        udtOpen     : ASYN_UDT_FILE_OPEN;
        udtSeek     : ASYN_UDT_FILE_SEEK;
        udtRead     : ASYN_UDT_FILE_READ;
        udtClose    : ASYN_UDT_FILE_CLOSE;
        iDummy      : INT;
    END_STRUCT;

END_TYPE
```

11 Support

For technical support please contact your local PHOENIX CONTACT agency

at https://www.phoenixcontact.com

Owner:

PHOENIX CONTACT Electronics GmbH
Business Unit Automation Systems
System Services
Library Services