Function block library

CANbus_11

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 Installation hint

Please copy the library data to your PLCnext Engineer (former: PC Worx Engineer) working library directory.

If you did not specify a different directory during **PLCnext Engineer** installation the default PLCnext Engineer working library directory is

C:\Users\Public\Documents\PLCnext Engineer\Libraries

2 General information

The CANbus library offers function blocks for support of and communication with the CANbus.

3 Change notes

Library version	Library build	PLCnext Engineer version	Change notes	Supported PLCs
11	20211021	2021.0 LTS	IL_CO_NMT: - Now supports Node ID 0 All available examples and example descriptions in documentation are adapted to standard.	AXC F 1152 (1151412) AXC F 2152 (2404267) AXC F 3152 (1069208)
10	20210401	2021.0 LTS	IL_J1939_RD: - New function block IL_J1939_RD_Multi: - New function block IL_J1939_WR: - New function block IL_NMEA_RD: - New function block IL_NMEA_RD_Multi: - New function block IL_NMEA_WR: - New function block	"

9	20200423	2020.0 LTS	CAN_UDT_DATA: - Added last messages and internal information for the mapping AXL_CAN_COMM: - Writes additional information in the communication structure IL_CO_EMCY: - New function block IL_CO_NMT_Guard: - New function block IL_CO_NodeGuard: - New function block IL_CO_NodeInfo: - New function block IL_CO_SDO_RD: - New function block IL_CO_SDO_WR: - New function block IL_CO_Search: - New function block IL_CO_SYNC: - New function block IL_CO_PDO_RD: - New function block IL_CO_PDO_WR: - New function block CAN_TO_AXL_STRUCT: - New function block CAN_TO_IL_STRUCT: - New function block	"
8	20200206	2020.0 LTS	Released for 2020.0 LTS	AXC F 1152 (1151412) AXC F 2152 (2404267)
8	20191001	2019.0 LTS 2019.3 2019.6 2019.9	Adapted to 2019.9	AXC F 2152 (2404267)
7	20190917	2019.0 LTS 2019.3 2019.6	IL_CAN_COMM: new. Converted from PC Worx 6. IL_CO_NMT: new. Converted from PC Worx 6. IL_CO_RD_WR: new. Converted from PC Worx 6.	"

6	20190722	2019.0 LTS 2019.3 2019.6	Adapted to 2019.6	"
5	20190219	2019.0 LTS	CANbus_5: • Adapted to PLCnext Engineer 2019.0 LTS	"
4	20181001	7.2.3	Adapted to PLCnext Engineer 7.3	"
3	20180625	7.2.2	AXL_CAN_COMM: • Bug fix for "index out of range" error AXL_CAN_Para: • Bug fix for "conflict in case of simultaneous access to AsynCom function blocks" error AXL_CAN_Para11: • Bug fix for "conflict in case of simultaneous access to AsynCom function blocks" error AXL_CAN_Para29: • Bug fix for "conflict in case of simultaneous access to AsynCom function blocks" error	"
2	-	7.2.2	No need for PCWEngineerAdaption anymore	"
1	-	7.2.2	Converted from PC Worx 6	"

New version number: Functional changes of at least one function block, incompatibilities (e.g. change of library format)

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

4 Function blocks

Function block	Description	Version	Supported articles	License
AXL_CAN_COMM	Driver for AXL F CAN Module	4	AXL F IF CAN 1H (2702668)	none
AXL_CAN_Para	Function block for parameterization of the AXL F CAN Module	4	AXL F IF CAN 1H (2702668)	none
AXL_CAN_Para11	Function block for parameterization of the AXL F CAN Module. For filter values in case of usage of 11 bit CAN identifier	3	AXL F IF CAN 1H (2702668)	none
AXL_CAN_Para29	Function block for parameterization of the AXL F CAN Module. For filter values in case of usage of 29 bit CAN identifier.	3	AXL F IF CAN 1H (2702668)	none
IL_CAN_COMM	Driver	1	IB IL CAN-MA-PAC (2700196)	none
IL_CO_EMCY	This function block is waiting for an emergency message. Additional information regarding the emergency message can be obtained from the outputs.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_NMT_Guard	Function block for changing operating mode of a node.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_NMT	Function block for determination and configuration operating mode of the CANopen node.	2	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_NodeGuard	Function block for displaying the current operating mode of a node.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_NodeInfo	Function block for reading information from a node about Hardware version and Software version of the module, name of the module or serial number of the module.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_PDO_RD	Function block for receiving PDO messages (e.g., 180 or 700).	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none

IL_CO_PDO_WR	Function block for sending PDO messages.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_RD_WR	Function block for setting the objects (indexes, subindexes) of a CANopen node.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_SDO_RD	Function block for reading contents of an index.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_SDO_WR	Function block for assigning a new value to an index.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_Search	Searches for available nodes in a CANopen network and displays their device names and node IDs.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_CO_SYNC	Function block for sending a COB-ID 80 synchronization message.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_NMEA_RD_Multi	Function block for targeted reading of the data of a packet from a multi-packet message.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_NMEA_RD	Function block for reading the current values from the array of a parameter group.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none

IL_NMEA_WR	Function block that can make up to 8 bytes of data in one node in an NMEA network available to a node in another NMEA network by entering a CAN ID.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_J1939_RD_Multi	Function block for reading the current data of a packet (parameter group) from a multi-packet message.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_J1939_READ	Function block for reading the current data of a packet (parameter group) from a standard message.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
IL_J1939_WRITE	Function block for transmitting data to a node in a J1939 network.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
CAN_TO_AXL_STRUCT	Function block for mapping data from the CN_udt_RxTx structure to the CAN_UDT_DATA structure.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none
CAN_TO_IL_STRUCT	Function block for mapping data from the CAN_UDT_DATA structure to the CN_udt_RxTx structure.	1	IB IL CAN-MA-PAC (2700196) AXL F IF CAN 1H (2702668)	none

5 AXL_CAN_COMM

With this function block it is possible to use the AXL F IF CAN module. The function block is able to write control bits in the process data and read out status bits. Additionally it is possible to write messages in the output process data and in the send buffer of the module as well as reading received messages from the input process data.

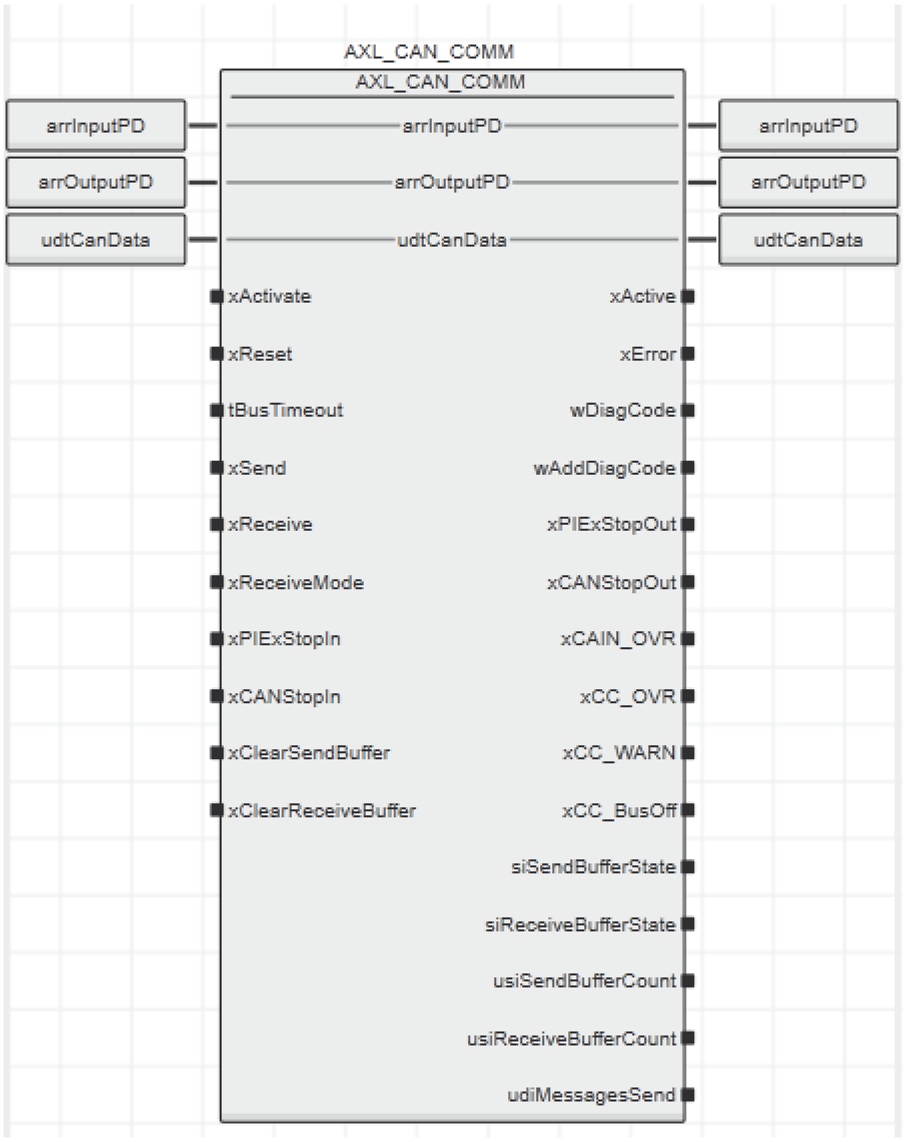
Both functions (receiving and sending) can be used with the same instance of the function block. By this the possibility of handling the data is limited.

If the data should be processed between receiving and sending, two instances have to be used.

The first instance should be used for receiving and the second instance for sending.

This function block is used to send and receive CAN messages. With the communication structure udtCanData it is possible to add messages to the send array or read out the messages in the receive array.

5.1 Function block call



5.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xReset	BOOL	Rising edge: Resets the function block.
tBusTimeout	TIME	The time the function block waits for the acknowledgment after sending a message. Initial value: 500ms.
xSend	BOOL	TRUE: Messages in the send-array will be transmitted in the send-buffer of the module.
xReceive	BOOL	TRUE: Messages will be transmitted from the receive-buffer of the module in the receive-array.
xReceiveMode	BOOL	TRUE: Receive-mode 1 is active. Received messages with the same ID will be stacked in the receive array with udiSequence showing how often it was received. FALSE: Receive-mode 2 is active. Received messages will be placed in the next free field in the receive-array.
xPIExStopIn	BOOL	Sets the status bit PIExStop.
xCanStopIn	BOOL	Sets the status bit CANStop.
xClearSendBuffer	BOOL	Deletes every message in the send-buffer of the module.
xClearReceiveBuffer	BOOL	Deletes every message in the receive buffer of the module.

5.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
wAddDiagCode	WORD	Additional diagnosis code. Refer to diagnostic table.
xPIExStopOut	BOOL	TRUE: Data-transfer between CAN and module stopped. FALSE: Normal operation mode.
xCanStopOut	BOOL	TRUE: CAN controller stopped. FALSE: Normal operation mode.
xCAIN_OVR	BOOL	TRUE: Receive-buffer overrun. Messages received after overrun were discarded. FALSE: No impact.
xCC_OVR	BOOL	TRUE: CAN controller overrun. A message was lost in the module. FALSE: No impact.
xCC_WARN	BOOL	TRUE: CAN controller warning. Controller reached warning level. FALSE: No impact.
xCC_BusOff	BOOL	TRUE: CAN controller bus off. The controller is in BusOff state. FALSE: No impact.
siSendBufferState	SINT	State of the send buffer. • 0: empty. • 1: 0% - 33% filled. • 2: 33% - 67% filled. • 3: 67% or more filled.
siReceiveBufferState	SINT	State of the receive buffer. • 0: empty. • 1: 0% - 33% filled. • 2: 33% - 67% filled. • 3: 67% or more filled.
usiSendBufferCount	USINT	Number of messages currently in the send buffer of the module.
usiReceiveBufferCount	USINT	Number of messages currently in the receive buffer of the module.
udiMessageSend	UDINT	Number of messages transmitted from the send array of the function block to the send buffer of the module.

5.4 Inout parameters

Name	Type	Description
arrInputPD	CAN_ARR_B_0_63	Input process data.
arrOutputPD	CAN_ARR_B_0_63	Output process data.
udtCanData	CAN_UDT_DATA	Communication structure for one or two instances of the function block.

5.5 Diagnosis

wDiagCode	wAddDiagCode	Description
16#0000	16#0000	Function block is deactivated.
16#8000		Function block is in regular operation
.	16#0000	No execution.
.	16#0001	Send mode executed.
.	16#0002	Receive mode executed.
16#C414	16#0001	Sending timeout.
16#C420	16#0001	Receive array full.
16#C900	16#0001	Index out of range. Please check the input process data.

6 AXL_CAN_Para

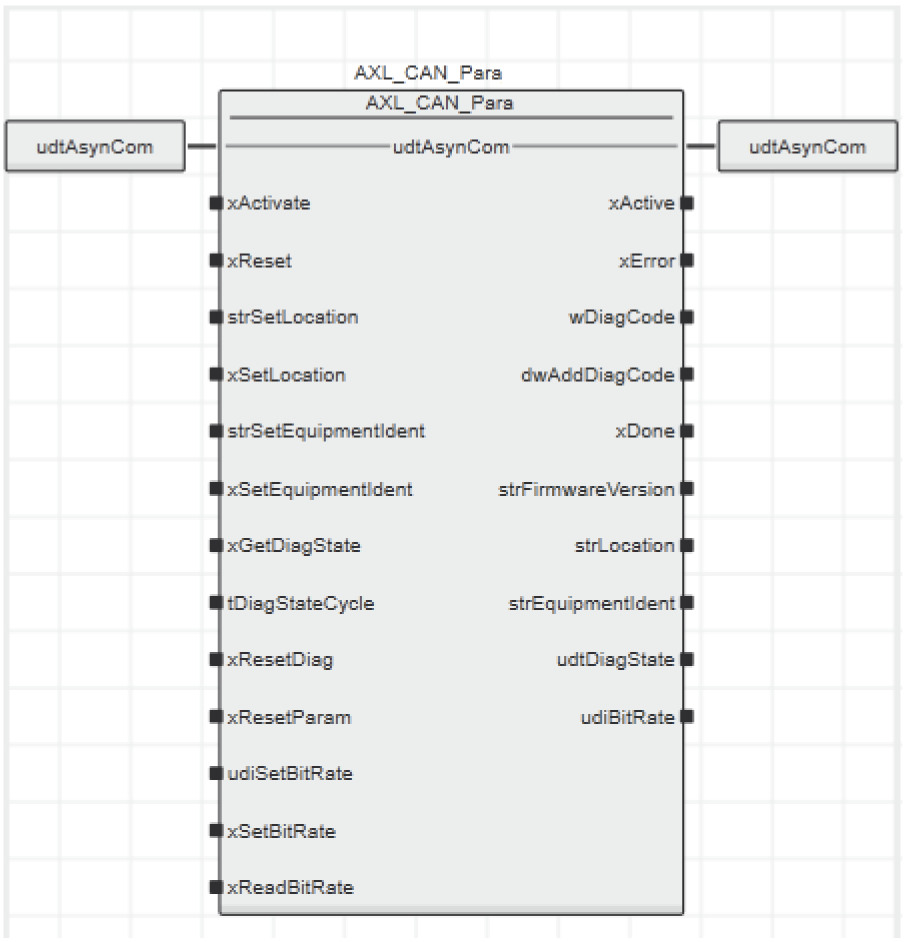
This function block allows user to set/read following parameters on AXL F IF CAN module:

- Location (set/read)
- Equipment identifier (set/read)
- Diagnosis state (only read once or cyclically)
- Bit rate (set/read)

Following functions can be performed:

- Reset diagnosis state
- Reset all parameters to default values

6.1 Function block call



6.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xReset	BOOL	Rising edge: Resets the function block.
strSetLocation	STRING	Location string to be set.
xSetLocation	BOOL	Execute setting location.
strSetEquipmentIdent	STRING	Equipment identifier string to be set.
xGetDiagState	BOOL	Get diagnosis status.
tDiagStateCycle	TIME	Interval for reading of diagnosis status (0 = no update).
xResetDiag	BOOL	Reset diagnosis.
xResetParam	BOOL	Set default values for all parameters.
udiSetBitRate	UDINT	Value for bit rate, which is set, when xSetBitRate is TRUE.
xSetBitRate	BOOL	Execute setting bit rate.
xReadBitRate	BOOL	Read bit rate.

6.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
wAddDiagCode	WORD	Additional diagnosis code. Refer to diagnostic table.
xDone	WORD	Command has been executed successfully (TRUE for 1 cycle).
strFirmwareVersion	STRING	Firmware version of module.
strLocation	STRING	Location of module.
strEquipmentIdent	STRING	Equipment identifier.
srDiagState	UDT_AXL_CAN_DiagState	Structure containing diagnosis status of module.
udiBitRate	UDINT	Current bit rate.

6.4 Inout parameters

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

6.5 Diagnosis

wDiagCode	wAddDiagCode	Description
16#0000	16#0000	Function block is deactivated.
16#8000		Function block is in regular operation
16#C010	16#0000	Error in communication function block!
16#C011	16#0000	Timeout! Communication function block could not be activated!
16#C020	16#0000	Error during reading or number occurred! Communication function block is not available!
16#C025	16#0000	Error! Wrong module type connected!
16#C026	16#0000	Error in communication function block during reading order number occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C027	16#0000	Error during reading order number occurred. Watchdog time exceeded. Communication function block does not response!
16#C030	16#0000	Error during reading firmware version occurred!
16#C035	16#0000	Error in communication function block during reading firmware version occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C036	16#0000	Error during reading firmware version occurred. Watchdog time exceeded. Communication function block does not response!
16#C040	16#0000	Error during reading location occurred! Communication function block is not available!
16#C045	16#0000	Error in communication function block during reading location occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C046	16#0000	Error during reading location occurred. Watchdog time exceeded. Communication function block does not response!
16#C050	16#0000	Error during reading equipment identifier occurred! Communication function block is not available!
16#C055	16#0000	Error in communication function block during reading equipment identifier occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C056	16#0000	Error during reading equipment identifier occurred. Watchdog time exceeded. Communication function block does not response!
16#C060	16#0000	Error during reading bit rate occurred! Communication function block is not available!
16#C065	16#0000	Error in communication function block during reading bit rate occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication!
16#C066	16#0000	Error during reading bit rate occurred. Watchdog time exceeded. Communication function block does not response!
16#C101	16#0000	Error! Invalid bit rate!
16#C200	16#0000	Error during reading bit rate occurred! Communication function block is not available!
16#C210	16#0000	Error in communication function block during reading bit rate occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C211	16#0000	Error during reading bit rate occurred. Watchdog time exceeded. Communication function block does not response!

16#C300	16#0000	Error during reading diagnosis state occurred! Communication function block is not available!
16#C310	16#0000	Error in communication function block during reading diagnosis state occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C311	16#0000	Error during reading diagnosis state occurred. Watchdog time exceeded. Communication function block does not response!
16#C400	16#0000	Error during setting location occurred! Communication function block is not available!
16#C410	16#0000	Error in communication function block during setting location occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C411	16#0000	Error during setting location occurred. Watchdog time exceeded. Communication function block does not response!
16#C430	16#0000	Error during reading location occurred! Communication function block is not available!
16#C440	16#0000	Error in communication function block during reading firmware version occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C441	16#0000	Error during reading location occurred. Watchdog time exceeded. Communication function block does not response!
16#C450	16#0000	Error! Location could not be set! Value to be set and value read back from the module are different!
16#C500	16#0000	Error during setting equipment identifier occurred! Communication function block is not available!
16#C510	16#0000	Error in communication function block during setting equipment identifier occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C511	16#0000	Error during setting equipment identifier occurred. Watchdog time exceeded. Communication function block does not response!
16#C530	16#0000	Error during reading equipment identifier occurred! Communication function block is not available!
16#C540	16#0000	Error in communication function block during reading equipment identifier occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C541	16#0000	Error during reading equipment identifier occurred. Watchdog time exceeded. Communication function block does not response!
16#C550	16#0000	Error! Equipment identifier could not be set! Value to be set and value read back from the module are different!
16#C600	16#0000	Error during resetting diagnosis occurred! Communication function block is not available!
16#C610	16#0000	Error in communication function block during resetting diagnosis occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C611	16#0000	Error during resetting diagnosis occurred. Watchdog time exceeded. Communication function block does not response!
16#C700	16#0000	Error during setting all parameters to default values occurred! Communication function block is not available!
16#C710	16#0000	Error in communication function block during resetting all parameters to default values occurred! dwAddDiagCode = Diagnosis code provided, by function block for asynchronous communication.
16#C711	16#0000	Error during resetting all parameters to default values occurred. Watchdog time exceeded. Communication function block does not response!

16#C730	16#0000	Error during reading location occurred!
16#C740	16#0000	Error in communication function block during reading location occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C741	16#0000	Error during reading location occurred. Watchdog time exceeded. Communication function block does not response!
16#C750	16#0000	Error during reading equipment identifier occurred! Communication function block is not available!
16#C760	16#0000	Error in communication function block during reading equipment identifier occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C761	16#0000	Error during reading equipment identifier occurred. Watchdog time exceeded. Communication function block does not response!
16#C770	16#0000	Error during reading bit rate occurred! Communication function block is not available!
16#C780	16#0000	Error in communication function block during reading bit rate occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C781	16#0000	Error during reading bit rate occurred. Watchdog time exceeded. Communication function block does not response!
16#C800	16#0000	Error during setting bit rate occurred! Communication function block is not available!
16#C810	16#0000	Error in communication function block during setting bit rate occurred! dwAddDiagCode = Diagnosis code provided by function block for asynchronous communication.
16#C811	16#0000	Error during setting bit rate occurred. Watchdog time exceeded. Communication function block does not response!
16#C830	16#0000	Error during reading bit rate occurred! Communication function block is not available!
16#C840	16#0000	Error in communication function block during reading bit rate occurred! dwAddDiagCode = Diagnosis code, provided by function block for asynchronous communication.
16#C841	16#0000	Error during reading bit rate occurred. Watchdog time exceeded. Communication function block does not response!
16#C850	16#0000	Error! Bit rate could not be set! Value to be set and value read back from the module are different!
16#C900	16#0000	Error occured while converting the response to string (BUF_TO_STRING Error).

7 AXL_CAN_Para11

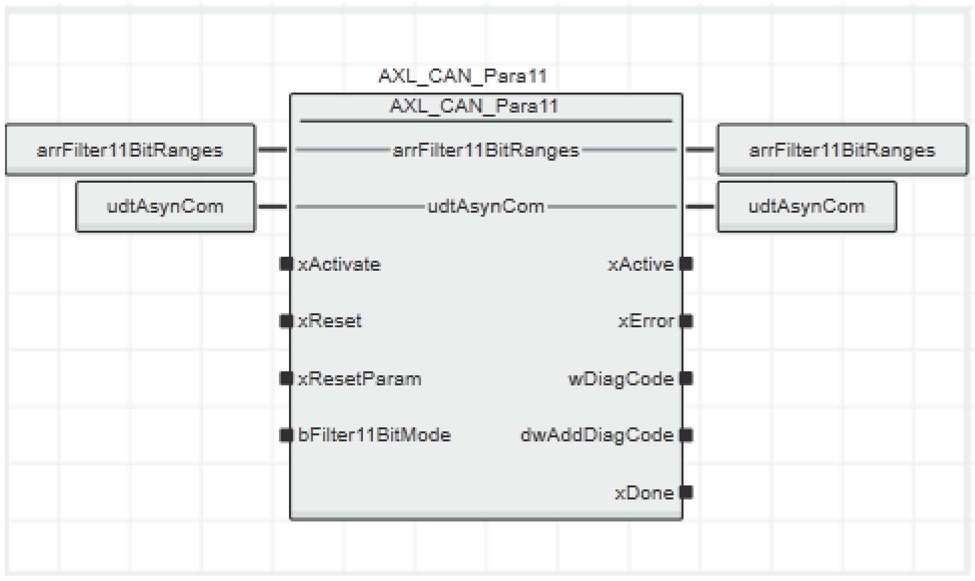
This function block allows user to set/read following parameters on AXL F IF CAN module:

- 11-bit filter mode
- 11-bit filter ranges

Following functions can be performed:

- Reset all parameters to default values

7.1 Function block call



7.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xReset	BOOL	Rising edge: Resets the function block.
xResetParam	BOOL	Set default values for all parameters.
bFilter11BitMode	BOOL	Defines 11-bit filter parameters: • 0 = receive all • 1 = block all; • 2 = filter active (receiving mode); • 3 = filter active (blocking mode).

7.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
wAddDiagCode	WORD	Additional diagnosis code. Refer to diagnostic table.
xDone	BOOL	Command has been executed successfully (TRUE for 1 cycle).

7.4 Inout parameters

Name	Type	Description
arrFilter11BitRanges	AXL_CAN_ARR_11BitFilterRange	Array containing filter parameters for 11-bit IDs.
udtAsynCom	ASYN_UDT_COM	Data exchange structure for asynchronous communication. Connected to AsynCom* function block.

7.5 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C010	Error in communication function block!
16#C011	Timeout! Communication function block could not be activated!
16#C020	Error! Invalid filter mode value!
16#C030	Error! Filter start parameter (From) is out of range! (Max = 2047)
16#C031	Error! Filter end parameter (To) is out of range! (Max = 2047)
16#C032	Error! Invalid filter parameter range!
16#C040	Error during reading or number occurred!
16#C050	Error! Wrong module connected (wrong node ID)!
16#C051	Error in communication function block during reading
16#C052	Error during reading order number occurred.
16#C100	Error during setting filter mode occurred!
16#C110	Error in communication function block during setting filter mode.
16#C111	Error during setting filter mode occurred. Watchdog time exceeded.
16#C120	Error during setting filter parameters occurred!
16#C130	Error in communication function block during setting filter.
16#C131	Error during reading location occurred.
16#C400	Error during setting all parameters to default values occurred!
16#C410	Error in communication function block during resetting all parameters.
16#C411	Error during resetting all parameters to default values occurred.

8 AXL_CAN_Para29

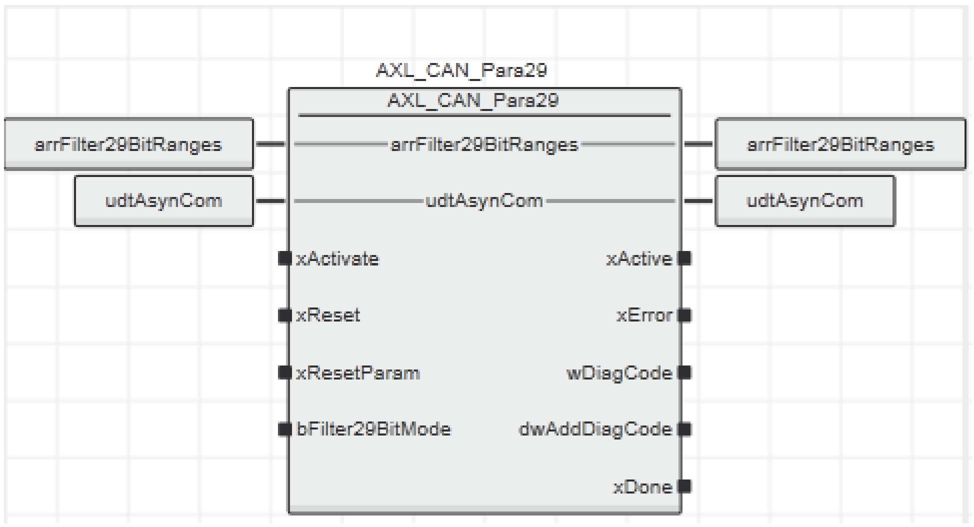
This function block allows user to set/read following parameters on AXL F IF CAN module:

- 29-bit filter mode
- 29-bit filter ranges

Following functions can be performed:

- Reset all parameters to default values

8.1 Function block call



8.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xReset	BOOL	Rising edge: Resets the function block.
xResetParam	BOOL	Set default values for all parameters.
bFilter29BitMode	BOOL	Defines 29-bit filter parameters: • 0 = receive all • 1 = block all; • 2 = filter active (receiving mode); • 3 = filter active (blocking mode).

8.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
wAddDiagCode	WORD	Additional diagnosis code. Refer to diagnostic table.
xDone	BOOL	Command has been executed successfully (TRUE for 1 cycle).

8.4 Inout parameters

Name	Type	Description
arrFilter29BitRanges	AXL_CAN_ARR_29BitFilterRange	Array containing filter parameters for 29-bit IDs.
udtAsynCom	ASYN_UDT_COM	Data exchange structure for asynchronous communication. Connected to AsynCom* function block.

8.5 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C010	Error in communication function block!
16#C011	Timeout! Communication function block could not be activated!
16#C020	Error! Invalid filter mode value!
16#C030	Error! Filter start parameter (From) is out of range! (Max = 536870911)
16#C031	Error! Filter end parameter (To) is out of range! (Max = 536870911)
16#C032	Error! Invalid filter parameter range!
16#C040	Error during reading or number occurred!
16#C050	Error! Wrong module connected (wrong node ID)!
16#C051	Error in communication function block during reading
16#C052	Error during reading order number occurred.
16#C100	Error during setting filter mode occurred!
16#C110	Error in communication function block during setting filter mode.
16#C111	Error during setting filter mode occurred. Watchdog time exceeded.
16#C120	Error during setting filter parameters occurred!
16#C130	Error in communication function block during setting filter.
16#C131	Error during reading location occurred.
16#C400	Error during setting all parameters to default values occurred!
16#C410	Error in communication function block during resetting all parameters.
16#C411	Error during resetting all parameters to default values occurred.

9 IL_CAN_COMM

This block communicates with the IL-CAN master via process data. All incoming messages are copied into an array that is conceived as FIFO. When the array is full, the oldest entry is overwritten.

Depending on whether all messages are confirmed (bConf.X4 = TRUE) or not, either the first found active or all active messages are transmitted. In the case of confirmed messages, a test is performed to determine whether the expected AnswerMessage was received... however, it will not be explicitly displayed or copied.

9.1 Special remarks

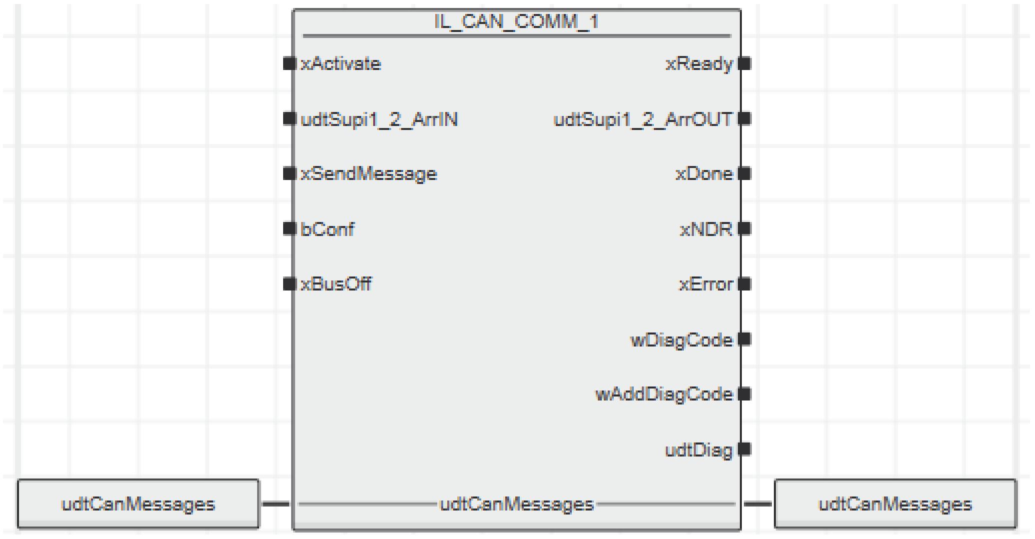
The user must copy the input and output data of both Supis to the corresponding structures.

This allows this block to be used for all 4 length variants (32, 64, 96 and 128 bytes) of the CAN master.

Here, transmitting and receiving is described from the point of view of the CAN master. Therefore Rx is to be received by the CAN master and Tx was transmitted from the CAN master.

If a message was incorrectly specified with a too large length (> 8Byte), the entry (.CanRx[n].iDLC) will be corrected during transmission.

9.2 Function block call



9.3 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
udtSupi1_2_ArrIN	UDT_CN_SUPI_1_2	Array with Supi1-IN and Supi2-IN of the IL-CAN master
xSendMessage	BOOL	Transmit message (is in udtCanMessages.Tx)
bConf	BYTE	Configuration byte is accepted on a positive edge from xActivate. X0 : Autorestart at CAN communication errors. The FB is reinitialized. X1 : Reserved. X2 : Each ID is stored in its own Tx-Array element X3 : DiagData ON/OFF. Fills the CN_udt_Diagnostic_CanNet structure. X4 : Confirmed Messages. All Rx-Messages are answered. (All messages are processed “one after the other”) X5 : Confirm TX-Messages. All Tx-Messages are answered. X6 : Status of DIP_Switch 1 on the module (length code) X7 : Status of DIP_Switch 2 on the module (length code)
xBusOff	BOOL	Restarts the bus if the bus status changes to "CAN Stop". It happens on a rising edge of xBusOff or automatically, if the input is permanently on TRUE.

9.4 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
udtSupi1_2_ArrOUT	UDT_CN_SUPI_1_2	Array with Supi1-OUT and Supi2-OUT of the IL-CAN master
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xNDR	BOOL	For 1 cycle TRUE of a message (or more) was received.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
wAddDiagCode	WORD	Additional diagnosis code. Refer to diagnostic table.
udtDiag	CN_udt_Diagnostic_CanNet	Structure description: iUsedRxBuf : %-value iMaxUsedRxBuf : %-value iTxCycleTime : Roundtrip to the module, only acknowledged messages %-value. iUsedTxBuf : %-value iMaxUsedTxBuf : %-value iRxCycleTimeCur : Actual Roundtrip to the module, only acknowledged messages iRxCycleTimeMax : Maximal Roundtrip to the module, only acknowledged messages

9.5 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

9.5.1 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

9.6 Diagnosis

wDiagCode	wAddDiagCode	Description
16#0000	16#0000	The block is not in operation.
16#8000	16#0000	The block is running without errors.
16#80xx	16#0000	CAN Info Bits
16#0000	16#0001	LPRXOVR -> Overrun LP RxQueue
16#0000	16#0002	COVR -> CAN overrun
16#0000	16#0004	BOFF -> CAN bus off
16#0000	16#0008	ESET -> CAN error-status-bit set
16#0000	16#0010	ERESET -> CAN error-status-bit reset
16#0000	16#0020	LPTXOVR -> Overrun LP TxQueue
16#0000	16#0040	HPRXOVR -> Overrun HP RxQueue
16#0000	16#0080	HPTXOVR -> Overrun HP TxQueue
16#C001	16#0000	TimeOut: no confirmation.
16#C101	16#0000	TimeOut: no confirmation, but automatic restart.
16#C200	16#0000	• Process data has not been assigned. • The physical connection between the module and the CANopen-Network is interrupted. • Terminal points of the IB_IL_CAN_MA were not correctly connected. • The baud rate of the module does not correspond to the baud rate of the CANopen-Network. • The IB IL CAN MA module is defective.

Remark

A reset of **xActivate** should be the response to all error codes with 16#Cxxx.

10 IL_CO*

CANopen is a communication protocol that is mainly used in automation technology and for networking within complex devices. CANopen is based on CAN and along with J1939 and DeviceNet it is the most-used protocol in Europe.

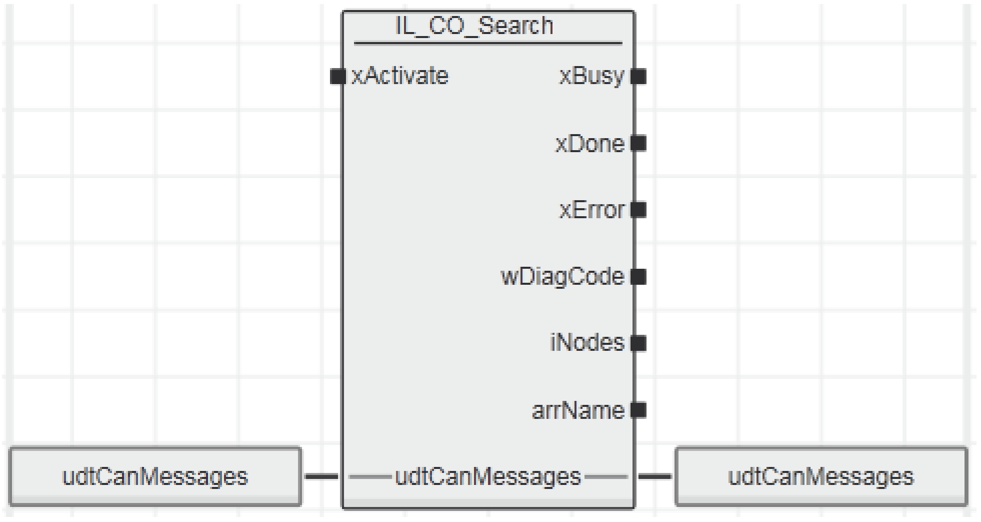
Various CANopen blocks are described in this documentation that enable us to:

a. Search for existing nodes in a CANopen network and display them together with names and IDs.
b. Determine the operating mode of a CANopen node as well as set one or all existing nodes to a certain CANopen operating mode.
c. Set objects (indexes/subindexes) of a CANopen node or request the current status of an object and some other options.

10.1 IL_CO_Search

This block can be used to search for existing nodes, the node ID, and the name in the CANopen network. These parameters are then displayed in the arrName structure.

10.1.1 Function block call



10.1.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.

10.1.3 Output parameters

Name	Type	Description
xBusy	BOOL	TRUE: The block is busy with the service execution.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
iNodes	INT	Number of detected nodes.
arrName	BYTE	16 string arrays.

10.1.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.1.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.1.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.

10.1.7 Startup instructions

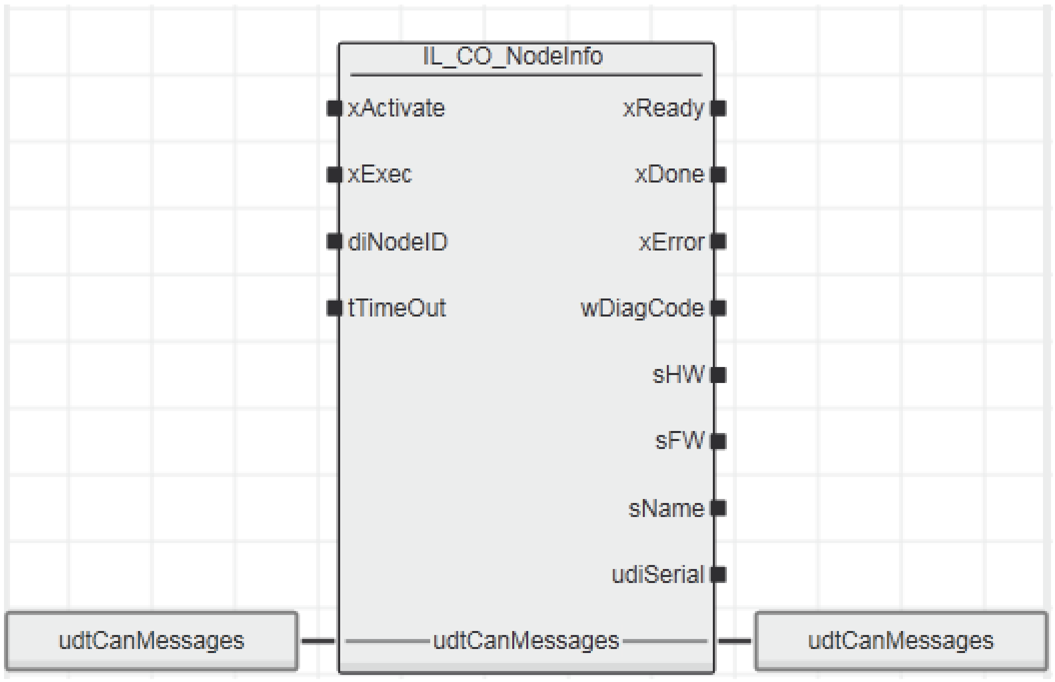
After activating the block, xBusy immediately switches to TRUE. When the search is completed, xBusy is reset and xDone is set. The code 8000 appears at the wDiagCode output. This code remains at the wDiagCode output until the search for CANopen nodes is completed. It will take approximately one minute until the xDone output is set and xBusy is reset. The wDiagCode value changes from 8000 to 0000. If xDone changes to TRUE, the iNodes output indicates the number of detected nodes. The detected nodes (max. 16) are saved at the arrName output together with their node IDs and names.

10.2 IL_CO_NodeInfo

The following information can be read from a node using this block:

- Hardware version of the module.
- Software version of the module.
- Name of the module.
- Serial number of the module.

10.2.1 Function block call



10.2.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xExec	BOOL	If no error occurs, information retrieval can be started.
diNodeID	DINT	Node ID of the node in the CANopen network.
tTimeOut	DINT	Time interval for data transmission.

10.2.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
sHW	STRING	Hardware version of the node.
sSW	STRING	Software version of the node
sName	STRING	Name of the node.
udiSerial	UDINT	Serial number of the node.

10.2.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.2.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.2.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C300	If the node does not respond.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	tTimeOut is set to zero.

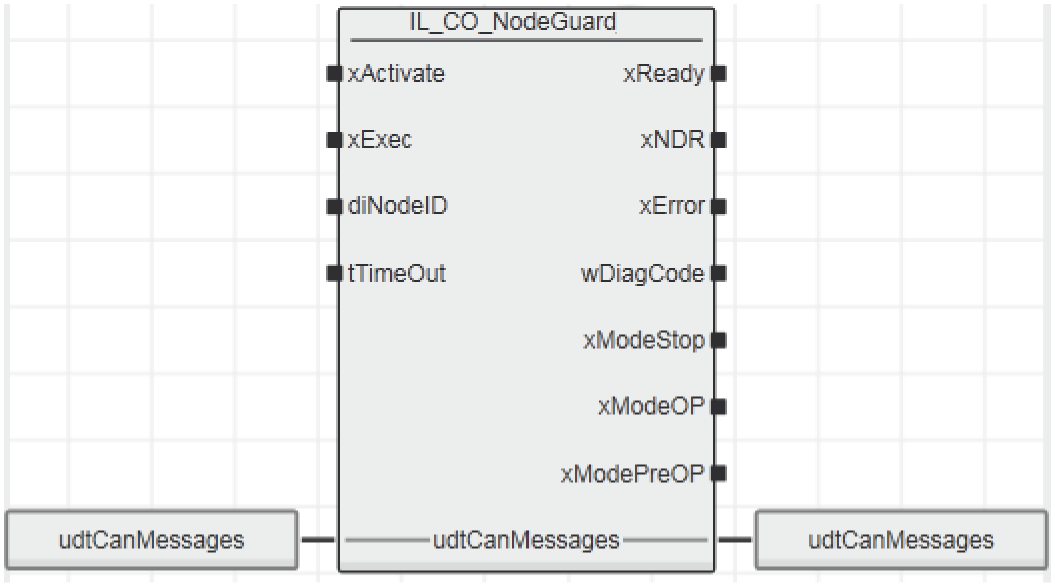
10.2.7 Startup instructions

If no error is present and the xReady output is set, the xExec input can be set. xReady is then reset. As soon as the requested data has arrived, xDone is set and the data is displayed at the output. Please note that xExec should only be set once xActivate has been set.

10.3 IL_CO_NodeGuard

The current operating mode of a node is displayed using this block. The determined operating mode becomes valid as soon as the xNDR output is set.

10.3.1 Function block call



10.3.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xExec	BOOL	If no error is present, a request can be sent by setting this input.
diNodeID	DINT	Node ID of the node in the CANopen network.
tTimeOut	TIME	Waiting time for a response from the CANopen node before an error is output.

10.3.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xNDR	BOOL	TRUE: information has been retrieved and is available at the output.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
xModeStop	BOOL	The CO node is in the STOP state.
xModeOP	BOOL	The CO node is in the OPERATIONAL state.
xModePreOP	BOOL	The CO node is in the PreOPERATIONAL state.

10.3.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.3.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.3.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	TimeOut is equal to zero.

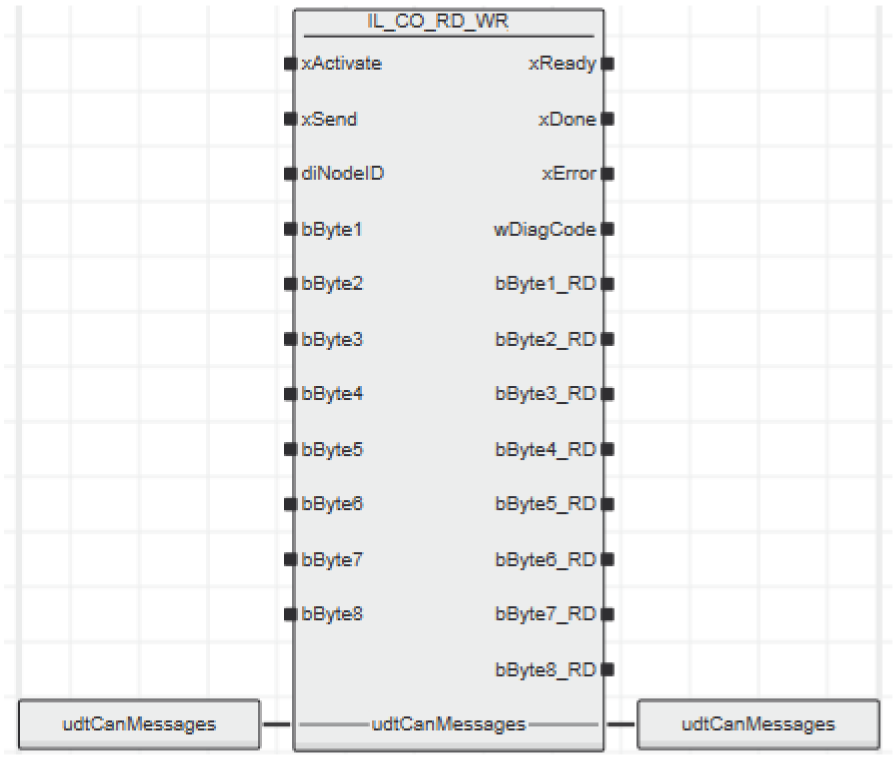
10.3.7 Startup instructions

If no error is present, xReady is set and wDiagCode indicates the value 16#8000. Only now can xSend be activated, xReady is then reset and the operating mode of the node is indicated. The validity of the indicated operating mode begins with the setting of the xNDR output. Please note that the operating mode of the node is not updated automatically. To determine the current operating mode, xSend must first be reset and then set again. Please note that xExec should only be set once xActivate has been set.

10.4 IL_CO_RD_WR

An SDO command is sent using this block and the validity of the command is displayed directly at the output.

10.4.1 Function block call



10.4.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xSend	BOOL	If no error is present, a command can be sent by setting this input.
diNodeID	DINT	Node ID of the node in the CANopen network.
bByte1	BYTE	CANopen command.
bByte2	BYTE	Index (low byte).
bByte3	BYTE	Index (high byte).
bByte4	BYTE	Subindex.
bByte5-8	BYTE	Data.

10.4.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
bByte1_RD	BYTE	Byte regarding the validity of the command.
bByte2_RD	BYTE	Index (low byte).
bByte3_RD	BYTE	Index (high byte).
bByte4_RD	BYTE	Subindex.
bByte5_RD-bByte8_RD	BYTE	Data.

10.4.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.4.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.4.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C300	If the node does not respond.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C303	Index = 0.
16#C306	Invalid CANopen command in Byte1.
16#C311	Incorrect index or subindex specified.

10.4.7 Startup instructions

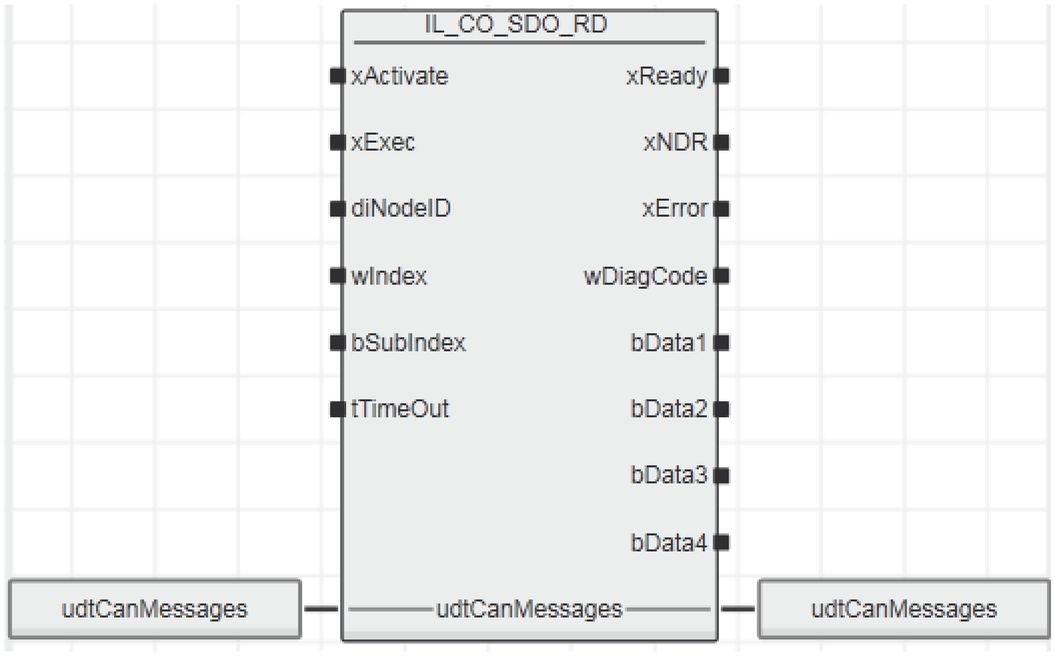
If all entries are correct, xReady is set and wDiagCode indicates the value 16#8000. Only now can xSend be activated. xReady is then reset and as soon as a positive (bByte1_RD = 60) or negative response (bByte1_RD = 80) is returned by the node, xDone is set. In the event of a negative response, it must be checked whether the index and subindex are correct.

If the node is defective or switched off, five seconds must pass after setting xSend and error message 16#C300 is then output. The advantage of this block compared to the IL_CO_RD and IL_CO_WR blocks is that the result of a request is displayed directly at the output. Please note that xSend should only be set once xActivate has been set.

10.5 IL_CO_SDO_RD

The contents of an index are read using this block.

10.5.1 Function block call



10.5.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xExec	BOOL	If no error occurs, information retrieval can be started.
diNodeID	DINT	Node ID of the node in the CANopen network.
wIndex	WORD	Index.
wSubIndex	WORD	Subindex.
tTimeOut	DINT	Waiting time for a response from the CANopen node before an error is output.

10.5.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xNDR	BOOL	TRUE: information has been retrieved and is available at the output.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
bData1-4	BYTE	Data.

10.5.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.5.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.5.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C300	If the node does not respond.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C303	Index = 0.
16#C305	TimeOut is equal to zero.
16#C311	Incorrect index or subindex specified.

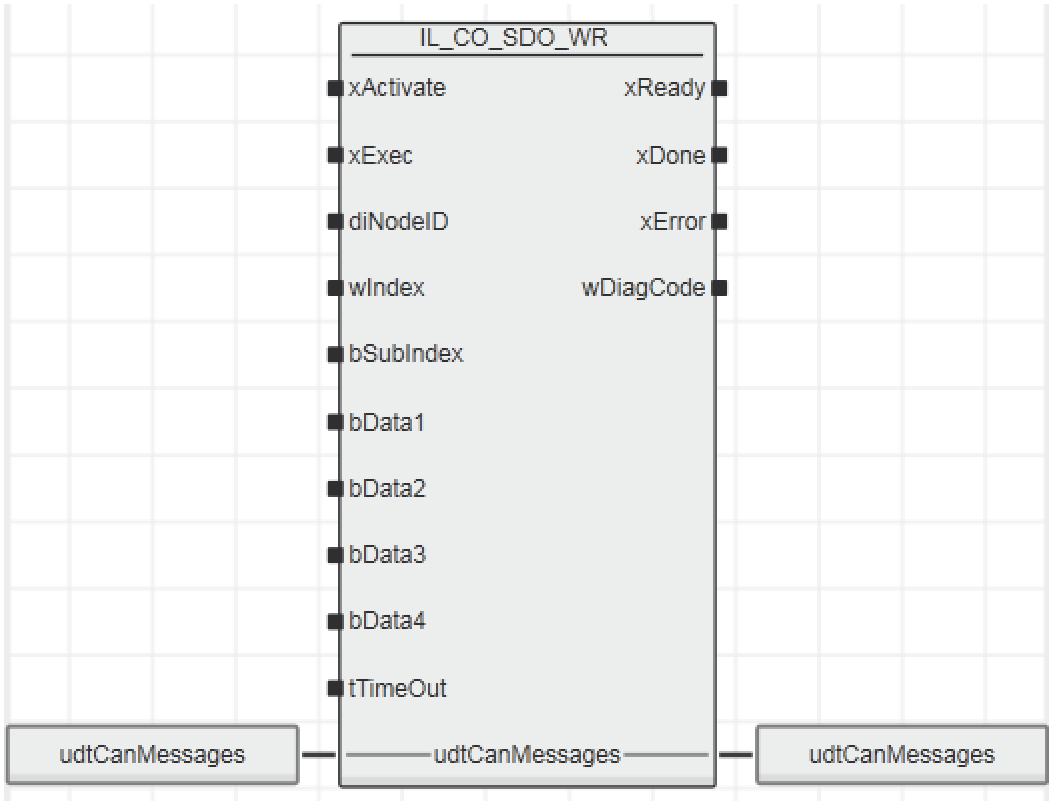
10.5.7 Startup instructions

If the entries are correct, the xReady output is set and wDiagCode indicates the value 16#8000. Only now can the xSend input be activated. xReady is then reset and xDone is set for exactly one cycle. As long as xExec is set, xReady remains reset. For a new request, xExec must first be reset and then set again. If the index or subindex at the input is incorrect, error code 16#C311 is displayed. If the node does not respond, error message 16#C300 appears as usual. Please note that xExec should only be set once xActivate has been set.

10.6 IL_CO_SDO_WR

A new value is assigned to an index using this block.

10.6.1 Function block call



10.6.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xExec	BOOL	If no error occurs, information retrieval can be started.
diNodeID	DINT	Node ID of the node in the CANopen network.
wIndex	WORD	Index.
bSubIndex	BYTE	subindex.
bData1-4	BYTE	Data.
tTimeOut	DINT	Waiting time for a response from the CANopen node before an error is output.

10.6.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.

10.6.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.6.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.6.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at !IB_IL_CAN_MA have been connected incorrectly. • The baud rate of the !IB_IL_CAN_MA module does not correspond to the network. • The !IB_IL_CAN_MA module is defective.
16#C300	If the node does not respond.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C303	Index = 0.
16#C305	TimeOut is equal to zero.
16#C311	Incorrect index or subindex specified.

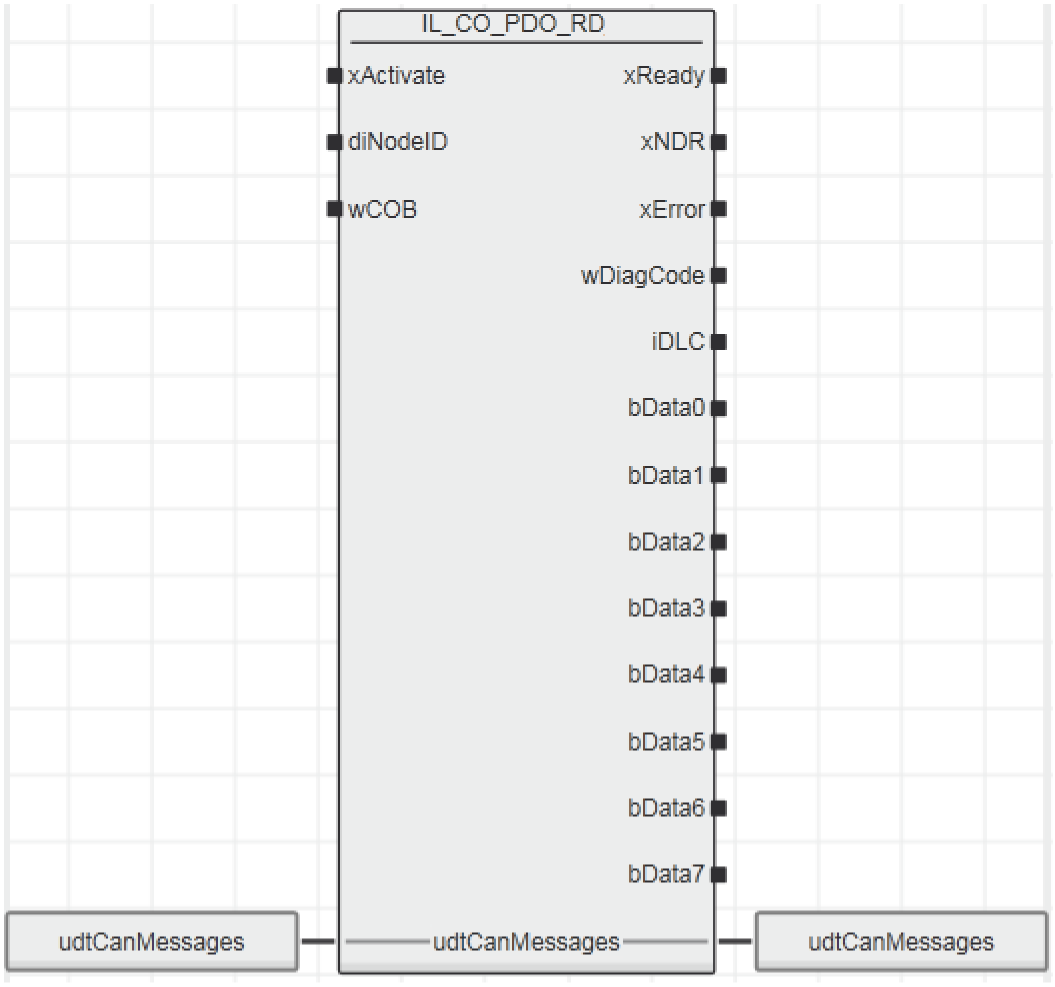
10.6.7 Startup instructions

If the entries are correct, the xReady output is set and wDiagCode indicates the value 16#8000. Only now can the xSend input be activated. xReady is then reset and xDone is set for exactly one cycle. As long as xExec is set, xReady remains reset. For a new request, xExec must first be reset and then set again. If the index or subindex at the input is incorrect, error code 16#C311 is displayed. If the node does not respond, error message 16#C300 appears as usual. Please note that xExec should only be set once xActivate has been set.

10.7 IL_CO_PDO_RD

The block waits for a PDO message (e.g., 180 or 700). As soon as a corresponding message is present, the contents of the message are displayed.

10.7.1 Function block call



10.7.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
diNodeID	DINT	Node ID of the node in the CANopen network.
wCOB	WORD	COB message designation, e.g., 180 or 700.

10.7.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xNDR	BOOL	TRUE: information has been retrieved and is available at the output.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
iDLC	INT	Amount of valid data in bytes.
bDate0-7	BYTE	Data.

10.7.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.7.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.7.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C307	COB message designation is equal to zero.

10.7.7 Startup instructions

Activating the block first checks whether the IL_CAN_COMM block is in the READY state. If no error message is present, the block is waiting for a PDO message. As soon as a message appears whose COB ID is the result of the sum of the diNode and wCOB inputs, the contents of the message are displayed and the xNDR output is simultaneously set for one INTERBUS cycle.

Please note that:

- Only messages that have not yet been retrieved can be displayed (i.e., xUsed of the message in the array is still TRUE).
- The data in the bData0 - bData7 outputs in the cycle have absolute validity in that xNDR is set to TRUE.

10.7.8 FAQs

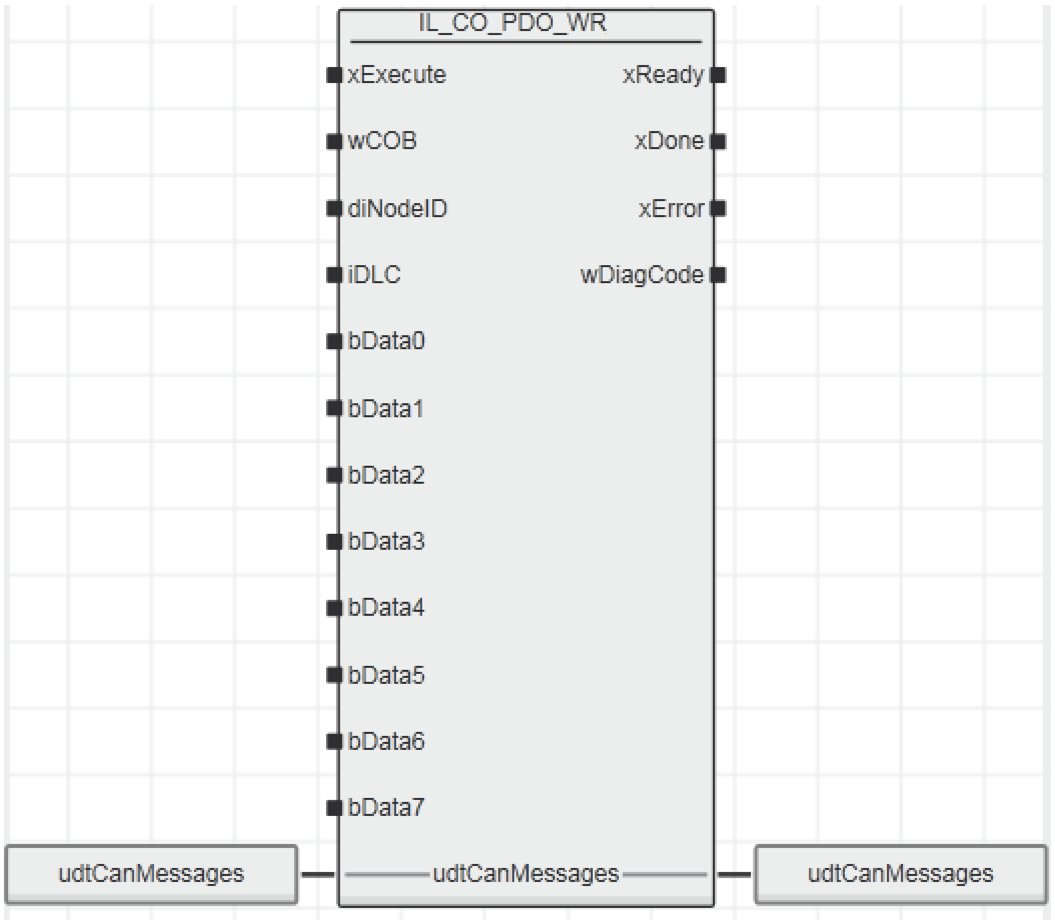
10.7.9 How is the !IB_IL_CAN_MA module configured?

The configuration of the terminal (e.g., setting the correct baud rate) is described on page 10 of the terminal data sheet.

10.8 IL_CO_PDO_WR

The block waits for the input xExecute. When a rising edge is detected, the block writes all inputs in a empty message space of the send array and sends the PDO to the CAN bus.

10.8.1 Function block call



10.8.2 Input parameters

Name	Type	Description
xExecute	BOOL	Rising edge: Executes the function block.
wCOB	WORD	Communication object identifier.
diNodeID	DINT	Node ID of the node in the CANopen network.
iDLC	INT	Number of used databytes.
bData0	BYTE	Databyte 1.
bData1	BYTE	Databyte 2.
bData2	BYTE	Databyte 3.
bData3	BYTE	Databyte 4.
bData4	BYTE	Databyte 5.
bData5	BYTE	Databyte 6.
bData6	BYTE	Databyte 7.
bData7	BYTE	Databyte 8.

10.8.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
iDLC	INT	Amount of valid data in bytes.
bDate0-7	BYTE	Data.

10.8.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.8.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.8.6 Diagnosis

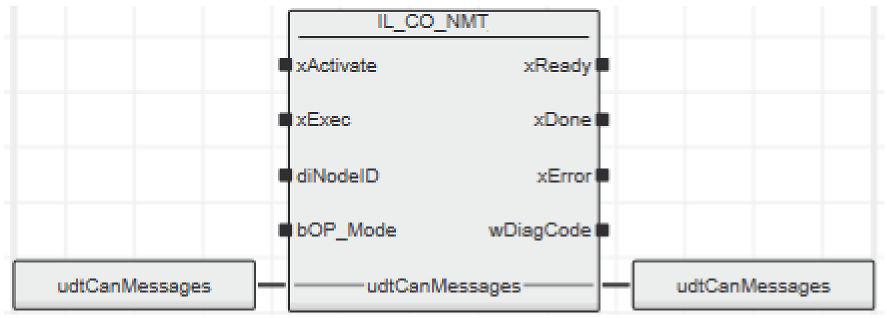
DiagCode	Description
16#C100	Function block has no error.
16#C100	xSend is not set at the *_CAN_COMM.
16#C200	The *_CAN_COMM is not ready.
16#C301	diNodeID is 0.
16#C307	wCOB is 0.
16#C420	Send-array is full.

10.9 IL_CO_NMT

The CANopen node can be set to one of the following operating modes using this block:

- 1 = Operational
- 2 = Preoperational
- 3 = Stop
- 4 = ResetNode
- 5 = ResetCommunication

10.9.1 Function block call



10.9.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
xExec	BOOL	Executes the function block.
diNodeID	DINT	Node ID of the device where the operation mode should be changed. Node ID 0 is for all nodes.
bOP_Mode	BYTE	The operation mode that should be set. 1=Operational 2=Preoperational 3=Stop 4=ResetNode 5=ResetCommunication

10.9.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.

10.9.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.9.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.9.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID < DINT#0 or diNodeID > DINT#127.
16#C302	bOp_Mode = byte#0 or bOP_Mode > byte#5.

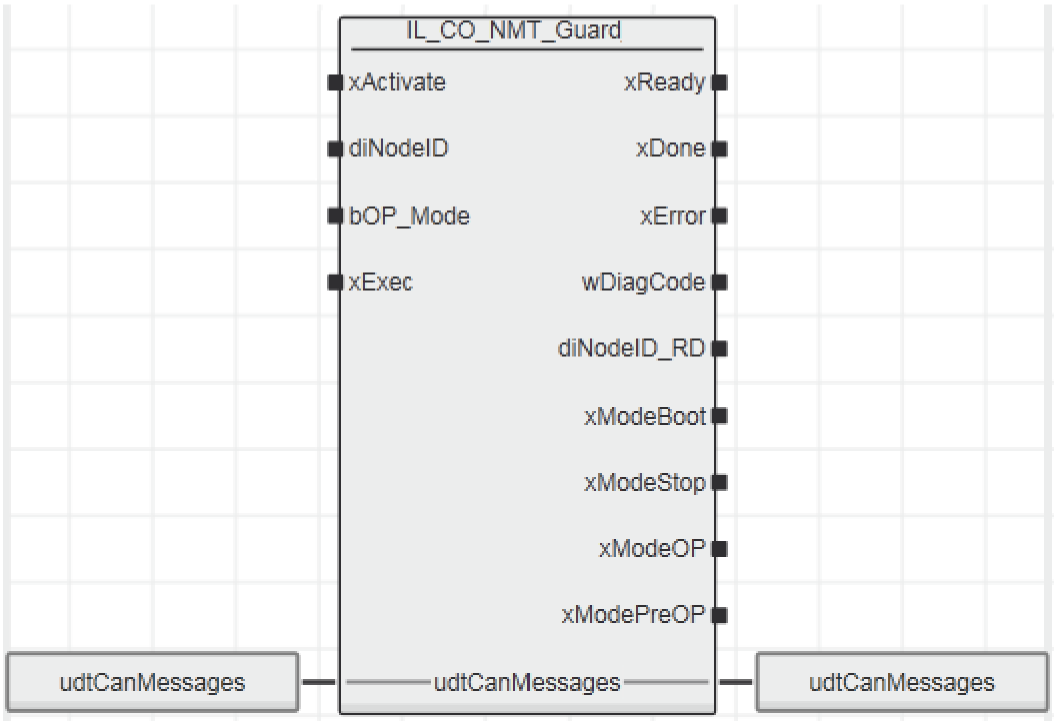
10.9.7 Startup instructions

Activating the block checks whether the IL_CAN_COMM block is ready to operate. If no error message is present and xExec is activated, a SYNC message is sent. The first byte of the SYNC message contains the number of sent SYNC messages (max. 255). Please note that xExec should only be set once xActivate has been set.

10.10 IL_CO_NMT_Guard

The operating mode of a node is changed using the block and the current operating mode is then displayed.

10.10.1 Function block call



10.10.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
diNodeID	DINT	Node ID of the node in the CANopen network.
bOP_Mode	BYTE	Operating mode.
xExec	BOOL	If no error is present, a request can be sent by setting this input.

10.10.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
diNodeID_RD	DINT	CO node ID.
xModeBoot	BOOL	The CO node is in the BOOT state.
xModeStop	BOOL	The CO node is in the STOP state.
xModeOP	BOOL	The CO node is in the OPERATIONAL state.
xModePreOP	BOOL	The CO node is in the PreOPERATIONAL state.

10.10.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.10.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.10.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C300	If the node does not respond.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C302	Incorrect operating mode (0 >= bOP_Mode > 5).

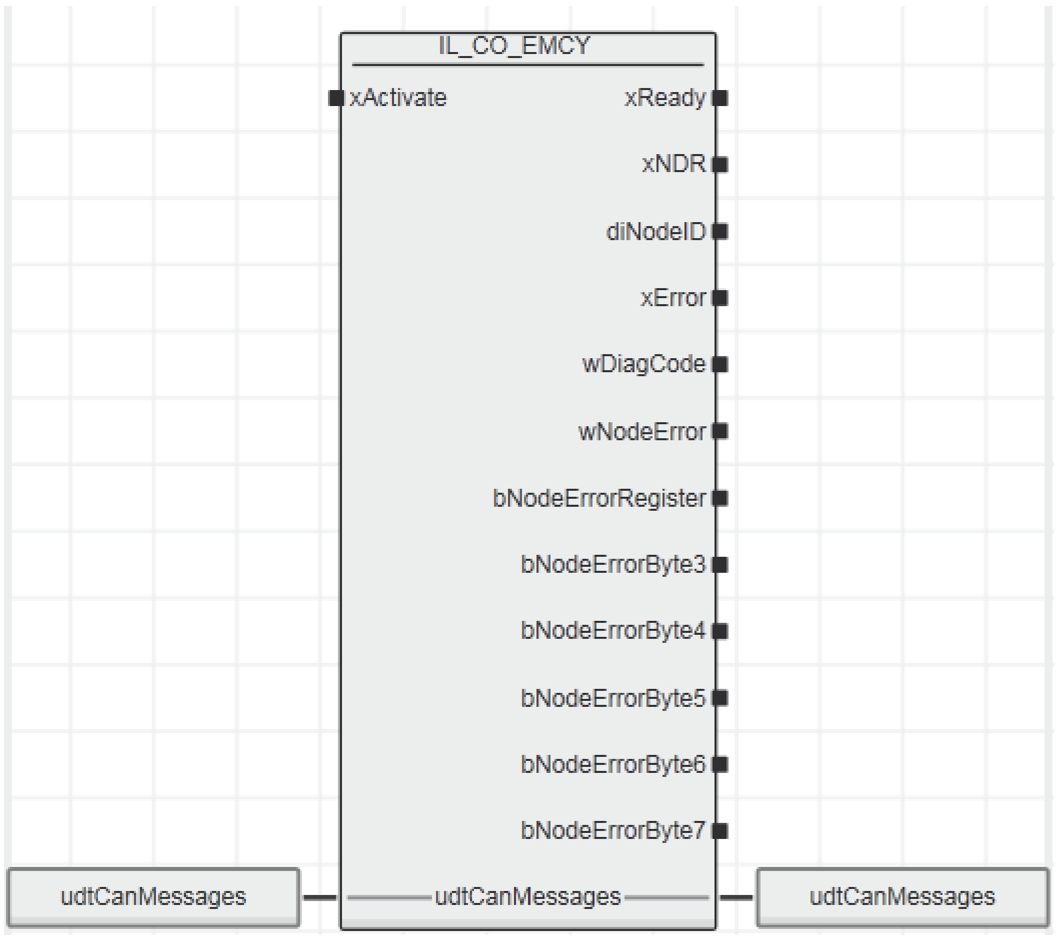
10.10.7 Startup instructions

Activating the block first checks whether the IL_CAN_COMM block is in the READY state. If no error message is present, a request to change the operating mode of a node can be sent by setting xExec. During processing 16#8100 is displayed as wDiagCode. When processing is complete, 16#8000 is output as wDiagCode and the xDone output is set. If, for whatever reason, no response is received from the node within 5 seconds, error message 16#C300 appears. The ID of the node used to determine the operating mode is output by the diNodeID_RD output. The current operating mode of the node is indicated by the xModeBoot, xModeStop, xModeOP, and xModePreOP outputs.

10.11 IL_CO_EMCY

This block is waiting for an emergency message.
As soon as a corresponding message is present, xNDR is set for one cycle. Additional information regarding the emergency message can be obtained from the outputs.

10.11.1 Function block call



10.11.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.

10.11.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xNDR	BOOL	TRUE: information has been retrieved and is available at the output.
diNodeID	DINT	Node ID of the node in the CANopen network.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
wNodeError	WORD	Alarm error code of the node.
bNodeErrorRegister	BYTE	Error register of the node.
bNodeErrorByte3-8	BYTE	Additional error information.

10.11.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.11.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.11.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.

10.11.7 Startup instructions

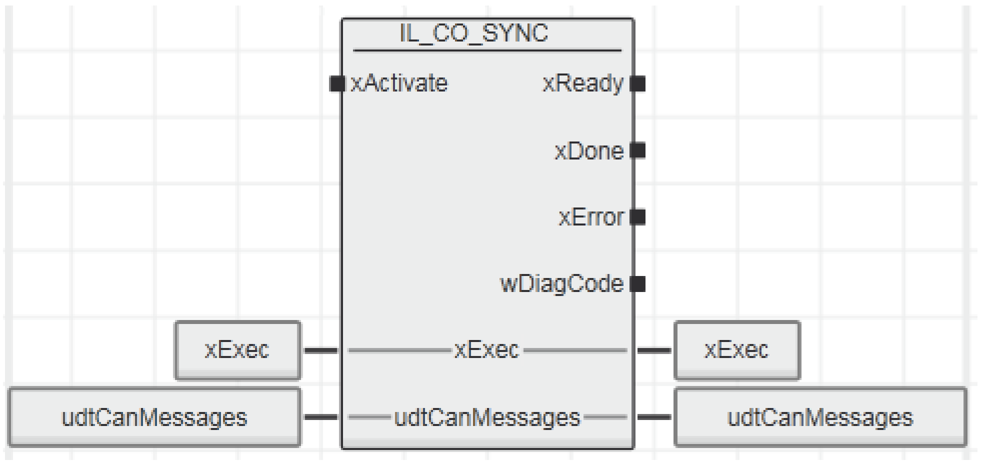
Please note that this information is only valid for one cycle where xNDR has been set. If the IL_CAN_COMM block is not yet in the READY state, this state is indicated at the output by wDiagCode = 16#C200.

Important:
If emergency messages are already present in the array prior to activation, the last message is shown at the output.

10.12 IL_CO_SYNC

A COB-ID 80 synchronization message is sent using this block.

10.12.1 Function block call



10.12.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.

10.12.3 Output parameters

Name	Type	Description
xReady	BOOL	FALSE: The function block is executing services. TRUE: The function block is ready to execute services.
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.

10.12.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for CanMessages

10.12.5 Struct CN_udt_CanMessage

Name	Type	Description
xUsed	BOOL	TRUE Receive (Tx) -> Indicates the received message. Transmit (Rx) -> Indicates the message to be transmitted. FALSE Messages are neither sent nor received.
diID	DINT	Number of the CAN-ID parameter group.
iDLC	INT	Message length when Tx incl. RTR and ID, when Rx only user data length.
udiSequence	UDINT	Number of times a parameter group occurred in a function block.
usiFrameFormat	USINT	0 = Standard, 1 = Extended
usiFrameType	USINT	0 = D (Data), 1 = R (RTR)
arrData	CN_ARR_B_1_8	Eight bytes data

10.12.6 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.

10.12.7 Startup instructions

Activating the block checks whether the IL_CAN_COMM block is ready to operate. If no error message is present and xExec is activated, a SYNC message is sent. The first byte of the SYNC message contains the number of sent SYNC messages (max. 255). Please note that xExec should only be set once xActivate has been set.

11 IL_J1939*

A CAN (Controller Area Network) is used to transmit electronic signals and information in cars and small industrial applications.

However, CAN does not meet all the requirements for trucks, busses, agricultural machinery, and building machinery.

That is the reason why the SAE organization “Society of Automotive Engineers” has developed standards for the design and use of control devices that transmit electronic signals as well as control information from one vehicle element (component) to another. This gave rise to the J1939 protocol based on the physical layer of CAN. The most important features of J1939 are the use of parameter groups, transport protocol functions and network management.

For this function block group, the following is recommended:

- Please use the latest *_CAN_COMM communication block.
- The bConf input on the IL_CAN_COMM block should be assigned value BYTE#16#C1.
- To increase the performance of the controller, use the *_CAN_COMM block in an event task with sequential preprocessing and the protocol blocks in a idle task, for example.
- I/O update should be done using the task the *_CAN_COMM block processes.

Parameter groups

For example, a parameter group can be the motor temperature that contains the cooler temperature, the oil temperature and the fuel temperature. The structure of a parameter group is illustrated below.

PGN 65262	Engine temperature
Transmission rate	1 sec
Data length	8 bytes
Data page	0
PDU format (PF)	254
PDU specific (PS)	238
Default priority	6
PG number	65262 (16#FEEE)

Data description:

BYTE	Description	SPN
1	Engine coolant temperature.	110
2	Fuel temperature.	174
3,4	Engine oil temperature.	175
5,6	Turbocharger oil temperature.	176
7	Engine intercooler temperature.	52
8	Engine intercooler thermostat Opening.	1134

Messages

The J1939 protocol allows data to be transmitted whose length is longer than eight bytes and a maximum of 1785 bytes. The data length in a CAN frame is always eight bytes irrespective of whether an 11-bit or 29-bit identifier is used. If data with a data length longer than eight bytes is to be transmitted, the data is divided into multiple messages with a data length of eight bytes each. This option is referred to as “Transport Protocol Functions”. The first byte of each message is reserved for the sequential number of the message.
Before transmitting data with a length longer than eight bytes, the control device transmits a special information message (BAM = Broadcast Announce Message) to all devices in the network first.

Network management

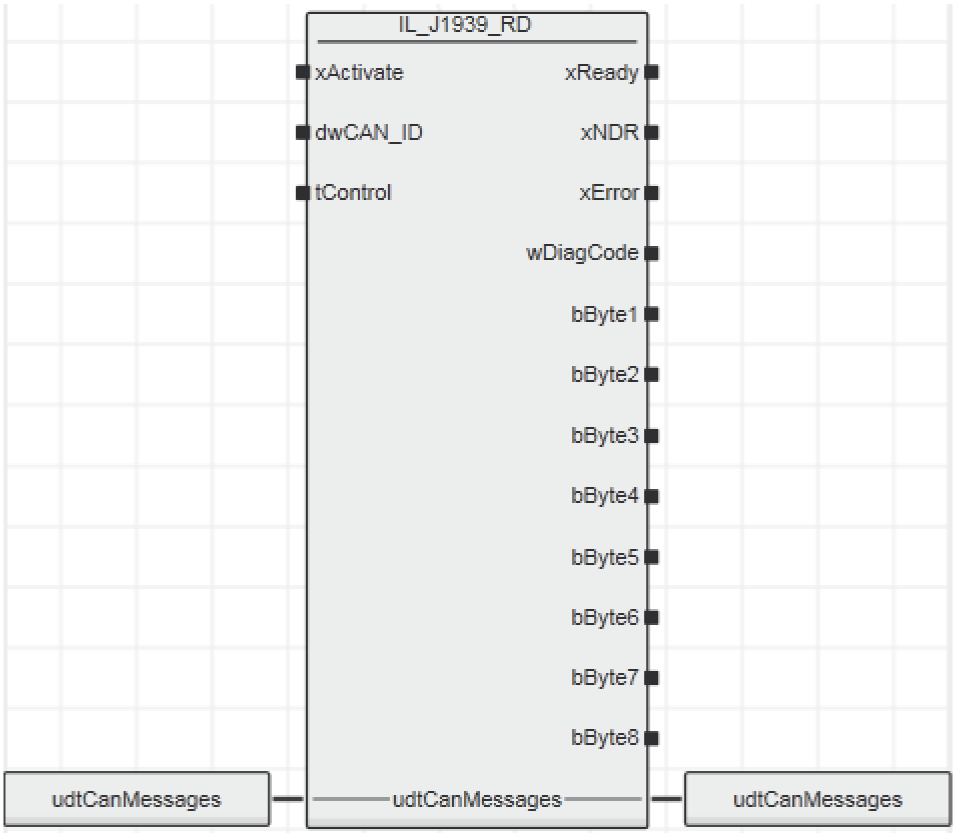
In addition to the BAM message, the network management under J1939 presents “Address Claiming Process” in the first instance. In this way, each control device notifies its address to all other control devices within the J1939 network after startup.

11.1 IL_J1939_RD

With the IL_J1939_RD function block the parameter group number as well as eight bytes of data are transmitted in a frame. This block reads the current data of a parameter group and displays them at the output.

This block implies that the connection via the IL_CAN_COMM communication block is established and therefore data exists.

11.1.1 Function block call



11.1.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
dwCAN_ID	DWORD	CAN ID.
tControl	TIME	Interval, in which the connection and the CAN ID are checked. This time must be longer than the period that is set in the J1939 device for the relevant CAN ID.

11.1.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xNDR	BOOL	TRUE: if a message is received.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
bByte1 - bByte8	BYTE	Data of the parameter group.

11.1.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for FIFO of messages.

11.1.5 Diagnosis

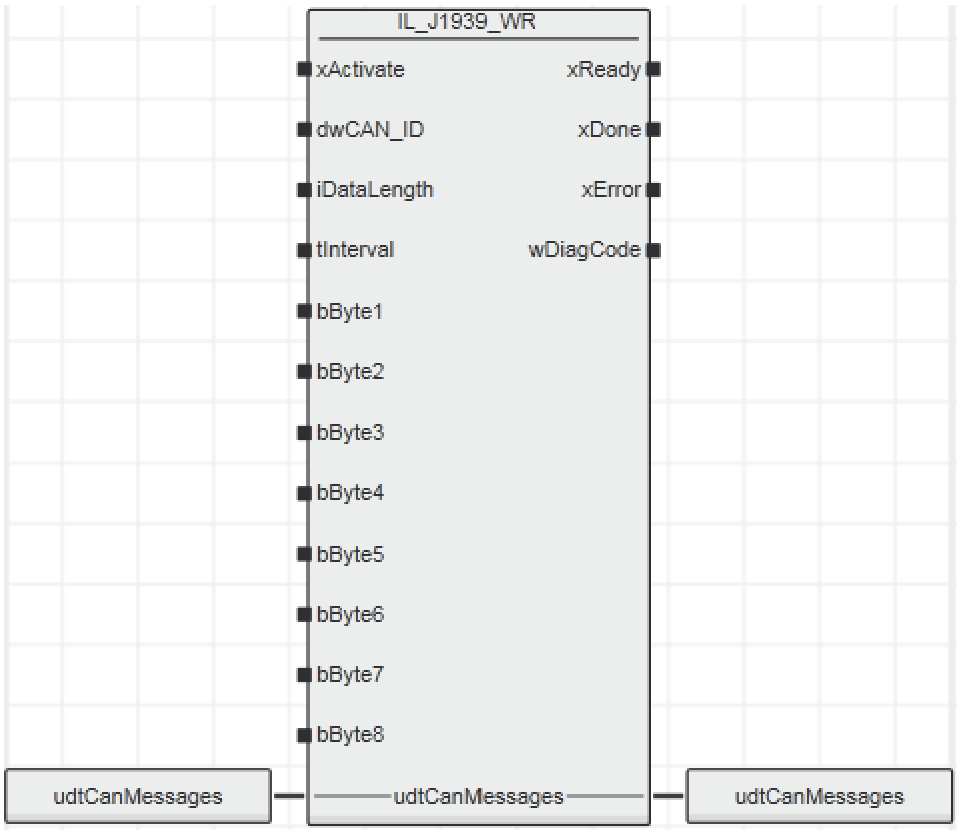
DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	Check time is equal to zero.
16#C310	CAN ID is not available. An interruption may have occurred. The block first waits for a certain check time and then the error message is output.

11.2 IL_J1939_WR

With the IL_J1939_WR function block the CAN ID as well as eight bytes of data are transmitted in a 29-bit frame. In a J1939 network, the block can make up to eight bytes of data available to another node in the network by entering a CAN ID.

This block implies that the connection via the IL_CAN_COMM communication block is established and therefore data exists.

11.2.1 Function block call



11.2.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
dwCAN_ID	DWORD	29-bit CAN identifier.
bDataLength	BYTE	Data length (max. eight bytes).
tInterval	TIME	Time interval for transmission of the data.
bByte1 - bByte8	BYTE	Data to be transmitted.

11.2.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.

11.2.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for FIFO of messages.

11.2.5 Diagnosis

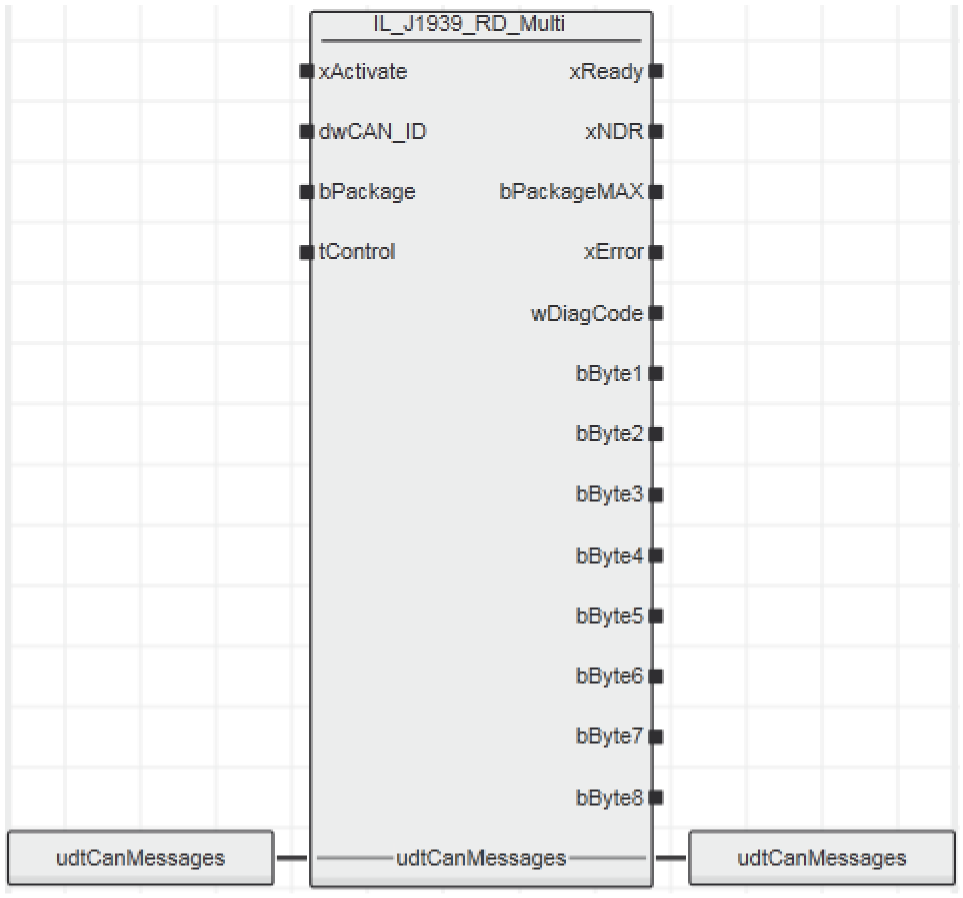
DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	Check time is equal to zero.
16#C313	0 > data length > 8: specified data length exceeds the limits.

11.3 IL_J1939_RD_Multi

With the IL_J1939_RD function block data with a length longer than eight bytes can be transmitted and received in J1939 networks. This block makes it possible to read specific data of a packet from a multi-packet message.

This block implies that the connection via the IL_CAN_COMM communication block is established and therefore data exists.

11.3.1 Function block call



11.3.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
dwCAN_ID	DWORD	BAM CAN ID.
bPackageNo	BYTE	Number of a packet in a multi-packet message.
tControl	TIME	Interval, in which the connection and the CAN ID are checked. This time must be longer than the period that is set in the J1939 device for the relevant CAN ID.

11.3.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xNDR	BOOL	TRUE: if a message is received.
bPackageMAX	BYTE	Maximum number of packets in a multi-packet message.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
bByte1 - bByte8	BYTE	Data of the parameter group.

11.3.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for FIFO of messages.

11.3.5 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	Check time is equal to zero.
16#C308	The entered CAN ID is not a BAM CAN ID.
16#C309	Packet number is equal to zero.
16#C310	CAN ID is not available. An interruption may have occurred. The block first waits for a certain check time and then the error message is output.
16#C319	Packet number does not exist.

12 IL_NMEA*

A CAN (Controller Area Network) is used to transmit electronic signals and information in cars and small industrial applications.

However, CAN does not meet all the requirements for trucks, busses, agricultural machinery, and building machinery.

That is the reason why the SAE organization "Society of Automotive Engineers" has developed standards for the design and use of control devices that transmit electronic signals as well as control information from one vehicle element (component) to another. This gave rise to the J1939 protocol based on the physical layer of CAN. The most important features of J1939 are the use of parameter groups, transport protocol functions and network management. The NMEA 2000 protocol is based on the J1939 protocol and works similar.

For this function block group, the following is recommended:

- Please use the latest *_CAN_COMM communication block.
- The bConf input on the IL_CAN_COMM block should be assigned value BYTE#16#C1.
- To increase the performance of the controller, use the *_CAN_COMM block in an event task with sequential preprocessing and the protocol blocks in a idle task, for example.
- I/O update should be done using the task the *_CAN_COMM block processes.

Messages

The NMEA 2000 protocol allows for sending messages with different data lengths.

- Single packet with a data length of up to 8 bytes.
- Multi-packet with a data length of up to 1785 bytes.
- Quick packet with a data length of up to 223 bytes.

The data length in a CAN frame is always eight bytes irrespective of whether an 11-bit or 29-bit identifier is used. If data with a data length longer than eight bytes is to be transmitted, the data is divided into multiple messages with a data length of eight bytes each. This option is referred to as "Transport Protocol Functions". The first byte of each message is reserved for the sequential number of the message. Before transmitting data with a length longer than eight bytes, the control device transmits a special information message (BAM = Broadcast Announce Message) to all devices in the network first.

Network management

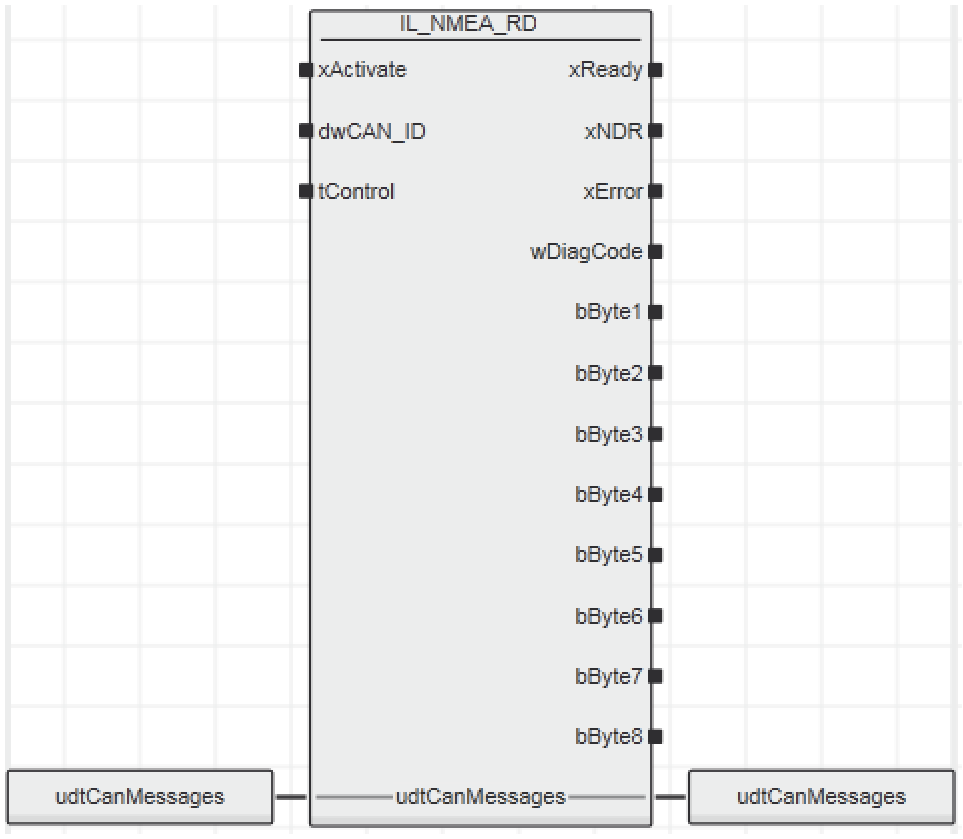
In addition to the BAM message, the network management under NMEA and J1939 presents "Address Claiming Process" in the first instance. In this way, each control device notifies its address to all other control devices within the NMEA network after startup.

12.1 IL_NMEA_RD

With the IL_NMEA_RD function block the parameter group number as well as eight bytes of data are transmitted in a frame. This block reads the current data of a parameter group and displays them at the output.

This block implies that the connection via the *_CAN_COMM communication block is established and therefore data exists.

12.1.1 Function block call



12.1.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
dwCAN_ID	DWORD	CAN ID.
tControl	TIME	Interval, in which the connection and the CAN ID are checked. This time must be longer than the period that is set in the NMEA device for the relevant CAN ID.

12.1.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xNDR	BOOL	TRUE: if a message is received.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
bByte1 - bByte8	BYTE	Data of the parameter group.

12.1.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for FIFO of messages.

12.1.5 Diagnosis

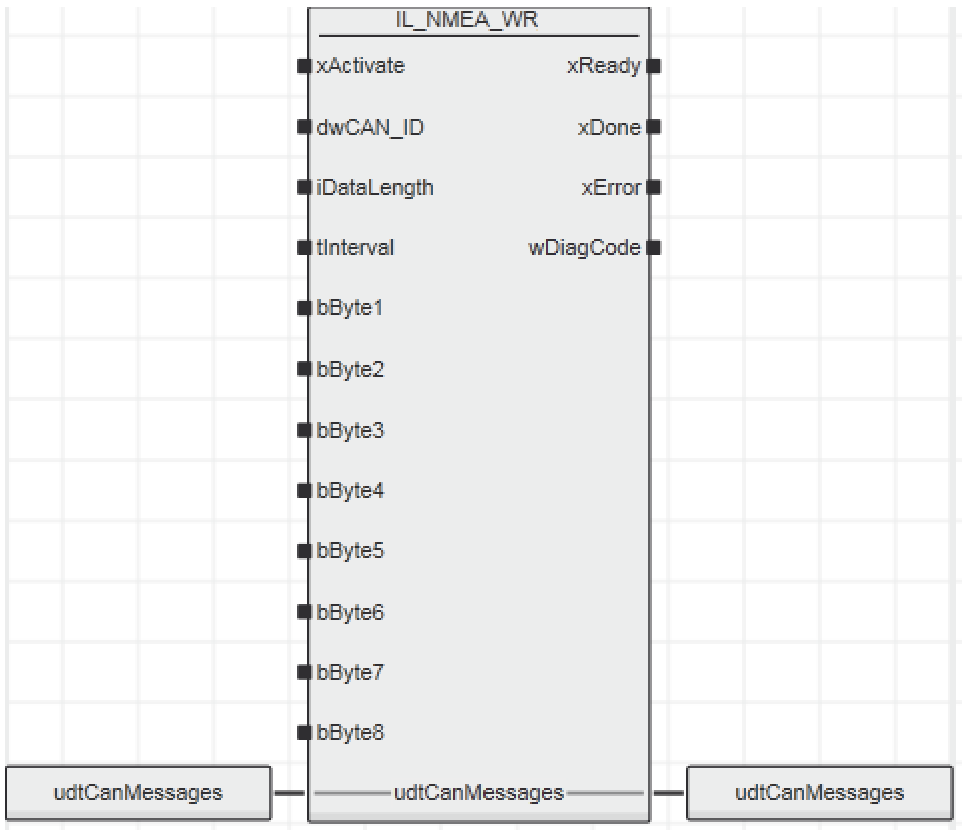
DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	Check time is equal to zero.
16#C310	CAN ID is not available. An interruption may have occurred. The block first waits for a certain check time and then the error message is output.

12.2 IL_NMEA_WR

With the IL_NMEA_WR function block the CAN ID as well as eight bytes of data are transmitted in a 29-bit frame. In a NMEA network, the block can make up to eight bytes of data available to another node in the network by entering a CAN ID.

This block implies that the connection via the *_CAN_COMM communication block is established and therefore data exists.

12.2.1 Function block call



12.2.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
dwCAN_ID	DWORD	29-bit CAN identifier.
bDataLength	BYTE	Data length (max. eight bytes).
tInterval	TIME	Time interval for transmission of the data.
bByte1 - bByte8	BYTE	Data to be transmitted.

12.2.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xDone	BOOL	TRUE: The request was sent and the response was successfully received from the communication partner. The parameter is TRUE for only one cycle.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.

12.2.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for FIFO of messages.

12.2.5 Diagnosis

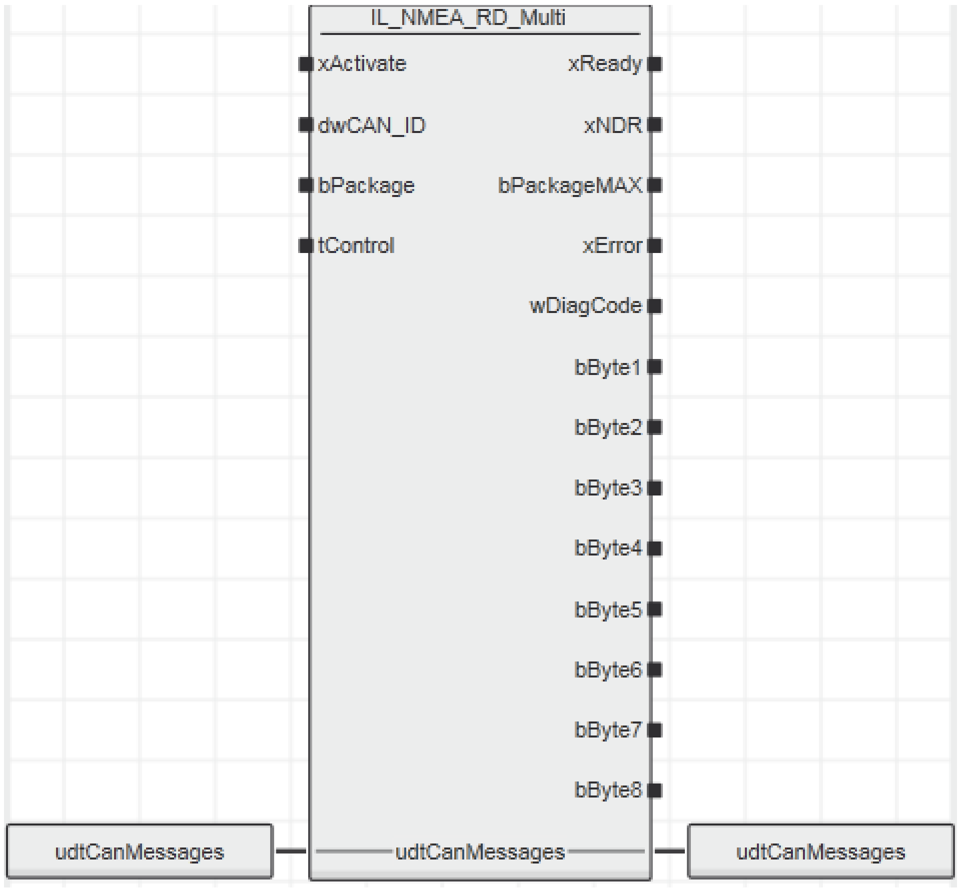
DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C100	The xSend block input is not set.
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	Check time is equal to zero.
16#C313	0 > data length > 8: specified data length exceeds the limits.

12.3 IL_NMEA_RD_Multi

With the IL_NMEA_RD function block data with a length longer than eight bytes can be transmitted and received in NMEA networks. This block makes it possible to read specific data of a packet from a multi-packet message.

This block implies that the connection via the *_CAN_COMM communication block is established and therefore data exists.

12.3.1 Function block call



12.3.2 Input parameters

Name	Type	Description
xActivate	BOOL	Rising edge: Activates the function block. FALSE: Deactivates the function block.
dwCAN_ID	DWORD	BAM CAN ID.
bPackageNo	BYTE	Number of a packet in a multi-packet message.
tControl	TIME	Interval, in which the connection and the CAN ID are checked. This time must be longer than the period that is set in the NMEA device for the relevant CAN ID.

12.3.3 Output parameters

Name	Type	Description
xActive	BOOL	FALSE: Function block is not active. TRUE: Function block is active. Do not start any further action unless xActive is TRUE after activation!
xNDR	BOOL	TRUE: if a message is received.
bPackageMAX	BYTE	Maximum number of packets in a multi-packet message.
xError	BOOL	TRUE: An error has occurred. For more details refer to wDiagCode and wAddDiagCode.
wDiagCode	WORD	Diagnosis code. Refer to diagnostic table.
bByte1 - bByte8	BYTE	Data of the parameter group.

12.3.4 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_CanMessage	Struct for FIFO of messages.

12.3.5 Diagnosis

DiagCode	Description
16#0000	Function block is deactivated
16#8000	Function block is in regular operation
16#C200	The xReady block output is FALSE, because: • xActivate has not been set. • Process data has not been assigned. • Terminal points at IB IL CAN-MA-PAC (2700196) have been connected incorrectly. • The baud rate of the IB IL CAN-MA-PAC (2700196) module does not correspond to the network. • The IB IL CAN-MA-PAC (2700196) module is defective.
16#C301	diNodeID outside the range 0 <= diNode-Id < 128.
16#C305	Check time is equal to zero.
16#C308	The entered CAN ID is not a BAM CAN ID.
16#C309	Packet number is equal to zero.
16#C310	CAN ID is not available. An interruption may have occurred. The block first waits for a certain check time and then the error message is output.
16#C319	Packet number does not exist.

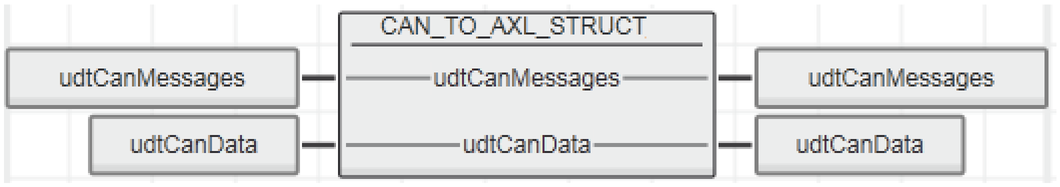
13 Mapping

The following two function blocks are neccessary for the use of IL_CO* function blocks with the AXL_CAN_COMM.

13.1 CAN_TO_AXL_STRUCT

This function block copies all data from the CN_udt_RxTx structure to the CAN_UDT_DATA structure.

13.1.1 Function block call



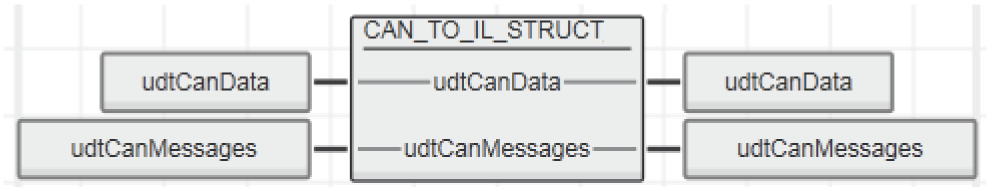
13.1.2 Inout parameters

Name	Type	Description
udtCanMessages	CN_udt_RxTx	Communication structure for IL_CAN_COMM.
udtCanData	CAN_UDT_DATA	Communication structure for AXL_CAN_COMM.

13.2 CAN_TO_IL_STRUCT

This function block copies all data from the CAN_UDT_DATA structure to the CN_udt_RxTx structure.

13.2.1 Function block call



13.2.2 Inout parameters

Name	Type	Description
udtCanData	CAN_UDT_DATA	Communication structure for AXL_CAN_COMM.
udtCanMessages	CN_udt_RxTx	Communication structure for IL_CAN_COMM.

14 Startup examples

For the startup instruction of the CANbus library please find the following examples:

- CAN_*_EXA_AXL.pcwex
- CAN_*_EXA_IL.pcwex
- CAN_*_EXA_AXL_CO.pcwex
- CAN_*_EXA_AXL_J1939.pcwex
- CAN_*_EXA_AXL_NMEA.pcwex

These examples are packed in the zipped Examples folder of the library.

They describe the use of the CANbus function blocks.

14.1 Example 1: CAN_*_EXA_AXL

14.1.1 Plant

For this example, the following hardware is used:

- AXC F 2152 (2404267)
- AXL F IF CAN 1H (2702668)

14.1.2 Example description

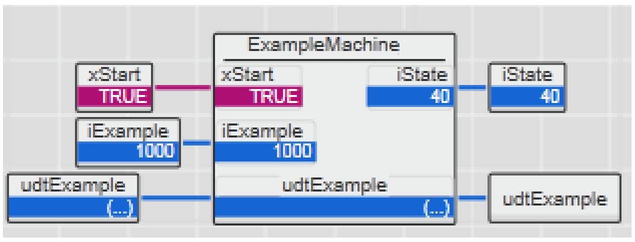
In this example project, we find the function block ExampleMachine. This function block contains a state machine for each example with a detailed description about what we have to do to use the function block correctly.

The following examples can be executed:

iExample	Codesheet	Description
1000	E_1000	Sending and receiving with one instance of the AXL_CAN_COMM.
2000	E_2000	Sending and receiving with two instances of the AXL_CAN_COMM.
3000	E_3000	Change the parametrization of the AXL F IF CAN 1H (2702668) with the AXL_CAN_Para function block.
4000	E_4000	Set filters of the AXL F IF CAN 1H (2702668) with the AXL_CAN_Para11 and AXL_CAN_Para29 function block.

14.1.2.1 Example machine

We can open the function block and read the step by step description or simply select our desired example by setting iExample to the belonging value and setting xStart to TRUE.



14.1.2.2 State machine: E_1000

```
(* E_1000 This example shows how we can send and receive messages
 with one instance of the AXL_CAN_COMM function block *)
IF udtExample.iExample  = 1000 THEN

    CASE udtExample.iState OF

        0: (* Activate *)
            udtExample.udtAXL_CAN_COMM_X1.xActivate     := TRUE;
            udtExample.udtAXL_CAN_COMM_X1.tBusTimeout   := t#1s;

            udtExample.iState   := 10;

        10: (* Check the function block is active *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode  = WORD#16#0000
            THEN
                (* Activate the send and receive mode *)
                udtExample.udtAXL_CAN_COMM_X1.xSend         := TRUE;
                udtExample.udtAXL_CAN_COMM_X1.xReceive      := TRUE;
                (* Messages with the same ID will be stacked *)
                udtExample.udtAXL_CAN_COMM_X1.xReceiveMode  := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* Check if the wanted mode is active *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode  = WORD#16#0002
                (* Since the send mode is executed first in the code, we only see
                that the receive mode is active in the wAddDiagCode *)
            THEN
                (* Preparing the message to send *)
                udtExample.udtCanData.arrMessagesSend[0].diID       := DINT#123;
                udtExample.udtCanData.arrMessagesSend[0].iDLC       := 8;
                udtExample.udtCanData.arrMessagesSend[0].arrData[1] := BYTE#16#01;
                udtExample.udtCanData.arrMessagesSend[0].arrData[2] := BYTE#16#23;
                udtExample.udtCanData.arrMessagesSend[0].arrData[3] := BYTE#16#45;
                udtExample.udtCanData.arrMessagesSend[0].arrData[4] := BYTE#16#67;
                udtExample.udtCanData.arrMessagesSend[0].arrData[5] := BYTE#16#89;
                udtExample.udtCanData.arrMessagesSend[0].arrData[6] := BYTE#16#AB;
                udtExample.udtCanData.arrMessagesSend[0].arrData[7] := BYTE#16#CD;
                udtExample.udtCanData.arrMessagesSend[0].arrData[8] := BYTE#16#EF;

                (* When the message is ready, we set xUsed so the message is send to
                the bus *)
                udtExample.udtCanData.arrMessagesSend[0].xUsed  := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* Check if the message was send *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive           = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError            = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode         = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode      = WORD#16#0002 AND
                udtExample.udtAXL_CAN_COMM_X1.udiMessagesSend   = UDINT#1
            THEN
                (* In our example, we have a CAN device that mirrors all received messages
                with the ID increased by 1, so now we check if we receive the right
                message *)
                udtExample.iState   := 40;
            END_IF;

        40: (* Wait for received message *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive                       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError                        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode                     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode                  = WORD#16#0002 AND
                udtExample.udtAXL_CAN_COMM_X1.udiMessagesSend               = UDINT#1 AND
                udtExample.udtCanData.arrMessagesReceive[0].xUsed           = TRUE AND
                udtExample.udtCanData.arrMessagesReceive[0].diID            = DINT#124 AND
                udtExample.udtCanData.arrMessagesReceive[0].iDLC            = 8 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[1]      = BYTE#16#01 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[2]      = BYTE#16#23 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[3]      = BYTE#16#45 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[4]      = BYTE#16#67 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[5]      = BYTE#16#89 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[6]      = BYTE#16#AB AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[7]      = BYTE#16#CD AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[8]      = BYTE#16#EF AND
                udtExample.udtCanData.arrMessagesReceive[0].udiSequence     = UDINT#1
            THEN
                (* If we want to send the same message again, we can simply set
                xUsed to TRUE again *)
                udtExample.udtCanData.arrMessagesSend[0].xUsed  := TRUE;

                udtExample.iState   := 50;
            END_IF;

        50: (* Check if the message was send again *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive           = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError            = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode         = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode      = WORD#16#0002 AND
                udtExample.udtAXL_CAN_COMM_X1.udiMessagesSend   = UDINT#2
            THEN
                (* Now we can wait again to receive the mirrored message *)
                udtExample.iState   := 60;
            END_IF;

        60: (* Wait for received message *)
            (* Because we set xReceiveMode to TRUE we will now see that the received
            message with an already received ID will be stacked, so now we have the
            udiSequence on TRUE *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive                       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError                        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode                     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode                  = WORD#16#0002 AND
                udtExample.udtAXL_CAN_COMM_X1.udiMessagesSend               = UDINT#2   AND
                udtExample.udtCanData.arrMessagesReceive[0].xUsed           = TRUE AND
                udtExample.udtCanData.arrMessagesReceive[0].diID            = DINT#124 AND
                udtExample.udtCanData.arrMessagesReceive[0].iDLC            = 8 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[1]      = BYTE#16#01 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[2]      = BYTE#16#23 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[3]      = BYTE#16#45 AND
```

```
                udtExample.udtCanData.arrMessagesReceive[0].arrData[4]  = BYTE#16#67 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[5]  = BYTE#16#89 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[6]  = BYTE#16#AB AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[7]  = BYTE#16#CD AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[8]  = BYTE#16#EF AND
                udtExample.udtCanData.arrMessagesReceive[0].udiSequence = UDINT#2
            THEN
                (* If everything is fine, we can just deactivate the function block again *)
                udtExample.udtCanData.arrMessagesSend[0].diID       := DINT#0;
                udtExample.udtCanData.arrMessagesSend[0].iDLC       := 0;
                udtExample.udtCanData.arrMessagesSend[0].arrData[1] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[2] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[3] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[4] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[5] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[6] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[7] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[8] := BYTE#16#00;

                udtExample.udtCanData.arrMessagesReceive[0].diID        := DINT#0;
                udtExample.udtCanData.arrMessagesReceive[0].udiSequence := UDINT#0;
                udtExample.udtCanData.arrMessagesReceive[0].iDLC        := 0;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[1]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[2]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[3]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[4]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[5]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[6]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[7]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[8]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].xUsed       := FALSE;

                udtExample.udtAXL_CAN_COMM_X1.xActivate     := FALSE;
                udtExample.udtAXL_CAN_COMM_X1.tBusTimeout   := t#0s;
                udtExample.udtAXL_CAN_COMM_X1.xSend         := FALSE;
                udtExample.udtAXL_CAN_COMM_X1.xReceive      := FALSE;
                udtExample.udtAXL_CAN_COMM_X1.xReceiveMode  := FALSE;

                udtExample.iState   := 70;
            END_IF;

        70: (* Wait until the function block is deactivated *)
            IF udtExample.udtAXL_CAN_COMM_X1.xActive    = FALSE THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Example finished *)
            udtExample.iExample := 32000;
            udtExample.iState   := 0;

    END_CASE;

END_IF;
```

14.1.2.3 State machine: E_2000

```
(* E_2000 This example shows how we can send and receive messages
 with two instances of the AXL_CAN_COMM function block *)
IF udtExample.iExample  = 2000 THEN

    CASE udtExample.iState OF

        0: (* Activate *)
            udtExample.udtAXL_CAN_COMM_X1.xActivate     := TRUE;
            udtExample.udtAXL_CAN_COMM_X2.xActivate     := TRUE;
            udtExample.udtAXL_CAN_COMM_X1.tBusTimeout   := t#1s;
            udtExample.udtAXL_CAN_COMM_X2.tBusTimeout   := t#1s;

            udtExample.iState := 10;

        10: (* Check the function blocks are active *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode  = WORD#16#0000 AND
                udtExample.udtAXL_CAN_COMM_X2.xActive       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X2.xError        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X2.wDiagCode     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X2.wAddDiagCode  = WORD#16#0000
            THEN
                (* Activate the send and receive mode. For maximum performance, we
                should call the receiving instance first. That way, we can receive
                data, process the data and send an answer in the same cycle.
                But for this exmaple, it is not necessary. *)
                udtExample.udtAXL_CAN_COMM_X1.xReceive      := TRUE;
                (* Messages with the same ID will be stacked *)
                udtExample.udtAXL_CAN_COMM_X1.xReceiveMode  := TRUE;
                udtExample.udtAXL_CAN_COMM_X2.xSend         := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* Check if the wanted mode is active *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode  = WORD#16#0002 AND
                udtExample.udtAXL_CAN_COMM_X2.xActive       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X2.xError        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X2.wDiagCode     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X2.wAddDiagCode  = WORD#16#0001
            THEN
                (* Preparing the message to send *)
                udtExample.udtCanData.arrMessagesSend[0].diID       := DINT#123;
                udtExample.udtCanData.arrMessagesSend[0].iDLC       := 8;
                udtExample.udtCanData.arrMessagesSend[0].arrData[1] := BYTE#16#01;
                udtExample.udtCanData.arrMessagesSend[0].arrData[2] := BYTE#16#23;
                udtExample.udtCanData.arrMessagesSend[0].arrData[3] := BYTE#16#45;
                udtExample.udtCanData.arrMessagesSend[0].arrData[4] := BYTE#16#67;
                udtExample.udtCanData.arrMessagesSend[0].arrData[5] := BYTE#16#89;
                udtExample.udtCanData.arrMessagesSend[0].arrData[6] := BYTE#16#AB;
                udtExample.udtCanData.arrMessagesSend[0].arrData[7] := BYTE#16#CD;
                udtExample.udtCanData.arrMessagesSend[0].arrData[8] := BYTE#16#EF;

                (* When the message is ready, we set xUsed so the AXL_CAN_COMM sends
                it to the bus *)
                udtExample.udtCanData.arrMessagesSend[0].xUsed  := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* Check if the message was send *)
            IF
                udtExample.udtAXL_CAN_COMM_X2.xActive           = TRUE AND
                udtExample.udtAXL_CAN_COMM_X2.xError            = FALSE AND
                udtExample.udtAXL_CAN_COMM_X2.wDiagCode         = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X2.wAddDiagCode      = WORD#16#0001 AND
                udtExample.udtAXL_CAN_COMM_X2.udiMessagesSend   = UDINT#1
            THEN
                (* In our example, we have a CAN device that mirrors all received messages
                with the ID increased by 1, so now we check if we receive the right
                message *)
                udtExample.iState   := 40;
            END_IF;

        40: (* Wait for received message *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive                       = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError                        = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode                     = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode                  = WORD#16#0002 AND
                (* Not the sending instance *)
                udtExample.udtAXL_CAN_COMM_X1.udiMessagesSend               = UDINT#0   AND
                udtExample.udtCanData.arrMessagesReceive[0].xUsed       = TRUE AND
                udtExample.udtCanData.arrMessagesReceive[0].diID        = DINT#124 AND
                udtExample.udtCanData.arrMessagesReceive[0].iDLC        = 8 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[1]  = BYTE#16#01 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[2]  = BYTE#16#23 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[3]  = BYTE#16#45 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[4]  = BYTE#16#67 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[5]  = BYTE#16#89 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[6]  = BYTE#16#AB AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[7]  = BYTE#16#CD AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[8]  = BYTE#16#EF AND
                udtExample.udtCanData.arrMessagesReceive[0].udiSequence = UDINT#1
            THEN
                (* If we want to send the same message again, we can simply set
                xUsed to TRUE again *)
                udtExample.udtCanData.arrMessagesSend[0].xUsed  := TRUE;

                udtExample.iState   := 50;
            END_IF;

        50: (* Check if the message was send again *)
            IF
                udtExample.udtAXL_CAN_COMM_X2.xActive           = TRUE AND
                udtExample.udtAXL_CAN_COMM_X2.xError            = FALSE AND
                udtExample.udtAXL_CAN_COMM_X2.wDiagCode         = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X2.wAddDiagCode      = WORD#16#0001 AND
                udtExample.udtAXL_CAN_COMM_X2.udiMessagesSend   = UDINT#2
            THEN
                (* Now we can wait again to receive the mirrored message *)
                udtExample.iState   := 60;
            END_IF;

        60: (* Wait for received message *)
            (* Because we set xReceiveMode to TRUE we will now see that the received
            message with an already received ID will be stacked, so now we have the
            udiSequence on TRUE *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive                    = TRUE AND
                udtExample.udtAXL_CAN_COMM_X1.xError                     = FALSE AND
                udtExample.udtAXL_CAN_COMM_X1.wDiagCode                  = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM_X1.wAddDiagCode               = WORD#16#0002 AND
                (* Not the sending instance *)
                udtExample.udtAXL_CAN_COMM_X1.udiMessagesSend            = UDINT#0 AND
                udtExample.udtCanData.arrMessagesReceive[0].xUsed        = TRUE AND
                udtExample.udtCanData.arrMessagesReceive[0].diID         = DINT#124 AND
                udtExample.udtCanData.arrMessagesReceive[0].iDLC         = 8 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[1]   = BYTE#16#01 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[2]   = BYTE#16#23 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[3]   = BYTE#16#45 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[4]   = BYTE#16#67 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[5]   = BYTE#16#89 AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[6]   = BYTE#16#AB AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[7]   = BYTE#16#CD AND
                udtExample.udtCanData.arrMessagesReceive[0].arrData[8]   = BYTE#16#EF AND
                udtExample.udtCanData.arrMessagesReceive[0].udiSequence = UDINT#2
            THEN
                (* If everything is fine, we can just deactivate the function block again *)
                udtExample.udtCanData.arrMessagesSend[0].diID       := DINT#0;
                udtExample.udtCanData.arrMessagesSend[0].iDLC       := 0;
                udtExample.udtCanData.arrMessagesSend[0].arrData[1] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[2] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[3] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[4] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[5] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[6] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[7] := BYTE#16#00;
                udtExample.udtCanData.arrMessagesSend[0].arrData[8] := BYTE#16#00;

                udtExample.udtCanData.arrMessagesReceive[0].diID        := DINT#0;
                udtExample.udtCanData.arrMessagesReceive[0].udiSequence := UDINT#0;
                udtExample.udtCanData.arrMessagesReceive[0].iDLC        := 0;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[1]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[2]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[3]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[4]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[5]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[6]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[7]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[8]  := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].xUsed       := FALSE;

                udtExample.udtAXL_CAN_COMM_X1.xActivate     := FALSE;
                udtExample.udtAXL_CAN_COMM_X1.xReceive      := FALSE;
                udtExample.udtAXL_CAN_COMM_X1.xReceiveMode  := FALSE;

                udtExample.udtAXL_CAN_COMM_X2.xActivate     := FALSE;
                udtExample.udtAXL_CAN_COMM_X2.xSend         := FALSE;

                udtExample.udtAXL_CAN_COMM_X1.tBusTimeout   := t#0s;
                udtExample.udtAXL_CAN_COMM_X2.tBusTimeout   := t#0s;

                udtExample.iState   := 70;
            END_IF;

        70: (* Wait until the function blocks are deactivated *)
            IF
                udtExample.udtAXL_CAN_COMM_X1.xActive   = FALSE AND
                udtExample.udtAXL_CAN_COMM_X2.xActive   = FALSE
            THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Example finished *)
            udtExample.iExample := 32000;
            udtExample.iState   := 0;

    END_CASE;

END_IF;
```

14.1.2.4 State machine: E_3000

```
(* E_3000 This example shows how we can change the parametrization of the
AXL F IF CAN module with the AXL_CAN_Para function block *)
IF udtExample.iExample  = 3000 THEN

    CASE udtExample.iState OF

        0: (* Activate *)
            (* Set AsynCom slot first *)
            udtExample.udtAsynComAXL.wSlot  := WORD#16#0001;

            udtExample.udtAXL_CAN_Para.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Check the function block is active *)
            IF
                udtExample.udtAXL_CAN_Para.xActive          = TRUE AND
                udtExample.udtAXL_CAN_Para.xError           = FALSE AND
                udtExample.udtAXL_CAN_Para.wDiagCode        = WORD#16#8000 AND
                udtExample.udtAXL_CAN_Para.dwAddDiagCode    = DWORD#16#00000000
            THEN
                (* Now we can read out any information we want, in this example
                we are reading the bitrate *)
                udtExample.udtAXL_CAN_Para.xReadBitRate := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait until the execution is done *)
            IF
                udtExample.udtAXL_CAN_Para.xActive          = TRUE AND
                udtExample.udtAXL_CAN_Para.xError           = FALSE AND
                udtExample.udtAXL_CAN_Para.wDiagCode        = WORD#16#8000 AND
                udtExample.udtAXL_CAN_Para.dwAddDiagCode    = DWORD#16#00000000 AND
                udtExample.udtAXL_CAN_Para.xDone            = TRUE
            THEN
                (* Now we can access the output of the value that we executed *)
                udiTemp                                 := udtExample.udtAXL_CAN_Para.udiBitRate
                udtExample.udtAXL_CAN_Para.xReadBitRate := FALSE;

                udtExample.iState   := 30;
            END_IF;

        30: (* We can also set different parameters. In this example, we want to set
            the location *)
            udtExample.udtAXL_CAN_Para.strSetLocation   := 'ExampleLocation';
            udtExample.udtAXL_CAN_Para.xSetLocation     := TRUE;

            udtExample.iState   := 40;

        40: (* Wait until the execution is done *)
            IF
                udtExample.udtAXL_CAN_Para.xActive          = TRUE AND
                udtExample.udtAXL_CAN_Para.xError           = FALSE AND
                udtExample.udtAXL_CAN_Para.wDiagCode        = WORD#16#8000 AND
                udtExample.udtAXL_CAN_Para.dwAddDiagCode    = DWORD#16#00000000 AND
                udtExample.udtAXL_CAN_Para.xDone            = TRUE
            THEN
                (* Now we can access the output of the value that we executed *)
                strTemp                                 := udtExample.udtAXL_CAN_Para.strLocatic
                udtExample.udtAXL_CAN_Para.xSetLocation := FALSE;
                udtExample.iState    := 50;
            END_IF;

        50: (* Now we can deactivate the function block *)
            udtExample.udtAXL_CAN_Para.xActivate    := FALSE;

            udtExample.iState    := 60;

        60: (* Wait until the function block is deactivated *)
            IF udtExample.udtAXL_CAN_COMM_X1.xActive    = FALSE THEN
                udtExample.iState    := 1000;
            END_IF;

        1000: (* Example finished *)
            udtExample.iExample := 32000;
            udtExample.iState    := 0;

    END_CASE;

END_IF;
```

14.1.2.5 State machine: E_4000

```
(* E_4000 This example shows how we can set the filters of the AXL F IF CAN
module with the AXL_CAN_Para11 and AXL_CAN_Para29 function block *)
IF udtExample.iExample  = 4000 THEN

    CASE udtExample.iState OF

        0: (* First we want to set the filters for the 11-bit identifier
            so we set everything at the AXL_CAN_Para11.
            We can set the filters we need. In this example, we dont want
            to receive messages in the ID range of 100 to 500 *)
            udtExample.udtAXL_CAN_Para11.arrFilter11BitRanges[0].uiFrom := UINT#100;
            udtExample.udtAXL_CAN_Para11.arrFilter11BitRanges[0].uiTo   := UINT#500;

            (* Now we determine when this filter range should do. We want to
            block it so we set the bFilterMode to 16#01. *)
            udtExample.udtAXL_CAN_Para11.bFilter11BitMode   := BYTE#16#03;

            (* Set AsynCom slot *)
            udtExample.udtAsynComAXL.wSlot  := WORD#16#0001;

            (* Now we can acitvate the function block and set the filters. *)
            udtExample.udtAXL_CAN_Para11.xActivate  := TRUE;

            udtExample.iState   := 10;

        10: (* Check if the function block is active and no error occures and wait
            for xDone *)
            IF
                udtExample.udtAXL_CAN_Para11.xActive        = TRUE AND
                udtExample.udtAXL_CAN_Para11.xError         = FALSE AND
                udtExample.udtAXL_CAN_Para11.xDone          = TRUE AND
                udtExample.udtAXL_CAN_Para11.wDiagCode      = WORD#16#8000 AND
                udtExample.udtAXL_CAN_Para11.dwAddDiagCode  = DWORD#16#000000000
            THEN
                (* Now the filters are set and we can continue with the 29-bit
                filters *)
                udtExample.iState   := 20;
            END_IF;

        20: (* Set parameter for AXL_CAN_Para29 *)
            (* Here we want to block messages in the ID range of 3000 to 5000 *)
            udtExample.udtAXL_CAN_Para29.arrFilter29BitRanges[0].udiFrom    := UDINT#3000;
            udtExample.udtAXL_CAN_Para29.arrFilter29BitRanges[0].udiTo      := UDINT#5000;

            (* Now we determine when this filter range should do. We want to
            block it so we set the bFilterMode to 16#01. *)
            udtExample.udtAXL_CAN_Para29.bFilter29BitMode   := BYTE#16#03;

            (* Now we can acitvate the function block and set the filters. *)
            udtExample.udtAXL_CAN_Para29.xActivate  := TRUE;

            udtExample.iState   := 30;

        30: (* Check the function block is active and no error occures and wait
            for xDone *)
            IF
                udtExample.udtAXL_CAN_Para29.xActive        = TRUE AND
                udtExample.udtAXL_CAN_Para29.xError         = FALSE AND
                udtExample.udtAXL_CAN_Para29.xDone          = TRUE AND
                udtExample.udtAXL_CAN_Para29.wDiagCode      = WORD#16#8000 AND
                udtExample.udtAXL_CAN_Para29.dwAddDiagCode  = DWORD#16#000000000
            THEN
                (* Now the filters are set and we can deactivate the function blocks *)
                udtExample.udtAXL_CAN_Para11.arrFilter11BitRanges[0].uiFrom := UINT#0;
                udtExample.udtAXL_CAN_Para11.arrFilter11BitRanges[0].uiTo   := UINT#0;
                udtExample.udtAXL_CAN_Para11.bFilter11BitMode               := BYTE#16#00;
                udtExample.udtAXL_CAN_Para11.xActivate                      := FALSE;

                udtExample.udtAXL_CAN_Para29.arrFilter29BitRanges[0].udiFrom    := UDINT#0;
                udtExample.udtAXL_CAN_Para29.arrFilter29BitRanges[0].udiTo      := UDINT#0;
                udtExample.udtAXL_CAN_Para29.bFilter29BitMode                   := BYTE#16#00;
                udtExample.udtAXL_CAN_Para29.xActivate                          := FALSE;

                udtExample.iState   := 40;
            END_IF;

        40: (* Wait until the function blocks are deactivated *)
            IF
                udtExample.udtAXL_CAN_Para11.xActive    = FALSE AND
                udtExample.udtAXL_CAN_Para29.xActive    = FALSE
            THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Example finished *)
            udtExample.iExample := 32000;
            udtExample.iState   := 0;

    END_CASE;

END_IF;
```

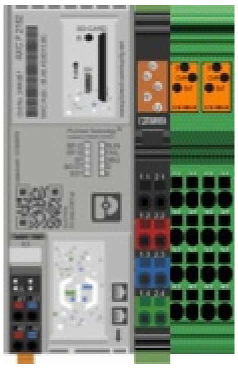
14.2 Example 2: CAN_*_EXA_IL

This example describes the use of the IL_CAN_COMM function block.

14.2.1 Plant

For this example, the following hardware is used:

- AXC F 2152 (2404267)
- AXC F IL ADAPT (1020304)
- IB IL CAN-MA-PAC (2700196)



14.2.2 Example description

The project contains one startup example for the function block. It can be found inside the ExampleMachine function block.

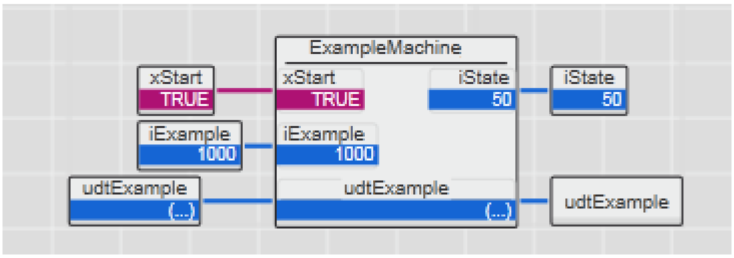
There is a state machines for every step we have to take care of when using the function block.

The following example can be executed :

iExample	Codesheet	Description
1000	E_1000	Receive and send messages with the IL_CAN_COMM.

14.2.2.1 Example machine

We can execute the example by selecting iExample and setting xStart to TRUE.



14.2.2.2 State machine: E_1000

```
(* E_1000 This example shows how the IL_CAN_COMM is used to send
and receive messages *)
IF udtExample.iExample  = 1000 THEN

    CASE udtExample.iState OF

        0: (* Configuration *)
            (* The IL_CAN_COMM function block has to be configured before it can be
            activated. This is done by setting the different bits of the input bConf *)

            (* Autorestart at CAN communication errors *)
            udtExample.udtIL_CAN_COMM.bConf.X0  := TRUE;
            (* Reserved *)
            udtExample.udtIL_CAN_COMM.bConf.X1  := FALSE;
            (* Each ID is stored in its own Tx-Array element *)
            udtExample.udtIL_CAN_COMM.bConf.X2  := TRUE;
            (* DiagData ON/OFF *)
            udtExample.udtIL_CAN_COMM.bConf.X3  := FALSE;
            (* Confirmed Messages *)
            udtExample.udtIL_CAN_COMM.bConf.X4  := FALSE;
            (* Confirm TX-Messages *)
            udtExample.udtIL_CAN_COMM.bConf.X5  := FALSE;
            (* Status of DIP_Switch 1 *)
            udtExample.udtIL_CAN_COMM.bConf.X6  := TRUE;
            (* Status of DIP_Switch 2 *)
            udtExample.udtIL_CAN_COMM.bConf.X7  := TRUE;

            udtExample.iState   := 10;

        10: (* Activation *)
            (* Then we can activate the function block *)
            udtExample.udtIL_CAN_COMM.xActivate := TRUE;

            udtExample.iState   := 20;

        20: (* Check the function block is active *)
            IF
                udtExample.udtIL_CAN_COMM.xReady        = TRUE AND
                udtExample.udtIL_CAN_COMM.xError        = FALSE AND
                udtExample.udtIL_CAN_COMM.wDiagCode     = WORD#16#8000 AND
                udtExample.udtIL_CAN_COMM.wAddDiagCode  = WORD#16#0000
            THEN
                (* Now we can activate the sending mode *)
                udtExample.udtIL_CAN_COMM.xSendMessage  := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* Build a message and send it to the bus *)
            udtExample.udtCanMessages.CanRx[1].diID         := DINT#100;
            udtExample.udtCanMessages.CanRx[1].iDLC         := 8;
            udtExample.udtCanMessages.CanRx[1].arrData[1]   := BYTE#16#01;
            udtExample.udtCanMessages.CanRx[1].arrData[2]   := BYTE#16#23;
            udtExample.udtCanMessages.CanRx[1].arrData[3]   := BYTE#16#45;
            udtExample.udtCanMessages.CanRx[1].arrData[4]   := BYTE#16#67;
            udtExample.udtCanMessages.CanRx[1].arrData[5]   := BYTE#16#89;
            udtExample.udtCanMessages.CanRx[1].arrData[6]   := BYTE#16#AB;
            udtExample.udtCanMessages.CanRx[1].arrData[7]   := BYTE#16#CD;
            udtExample.udtCanMessages.CanRx[1].arrData[8]   := BYTE#16#EF;

            (* Now we set xUsed so the function block knows the message
            is ready to be send *)
            udtExample.udtCanMessages.CanRx[1].xUsed    := TRUE;

            udtExample.iState   := 40;

        40: (* Wait for a confirmation *)
            IF
                udtExample.udtIL_CAN_COMM.xReady        = TRUE AND
                udtExample.udtIL_CAN_COMM.xError        = FALSE AND
                udtExample.udtIL_CAN_COMM.wDiagCode     = WORD#16#8000 AND
                udtExample.udtIL_CAN_COMM.wAddDiagCode  = WORD#16#0000 AND
                udtExample.udtIL_CAN_COMM.xDone         = TRUE
            THEN
                (* Message has been send successfully.
                Since we have a CAN device in the bus that mirrors the messages
                that are received with a ID increased by 1, we are now waiting
                until we received this message *)
                udtExample.iState   := 50;
            END_IF;

        50: (* Wait for a message *)
            IF
                udtExample.udtIL_CAN_COMM.xReady                = TRUE AND
                udtExample.udtIL_CAN_COMM.xError                = FALSE AND
                udtExample.udtIL_CAN_COMM.wDiagCode             = WORD#16#8000 AND
                udtExample.udtIL_CAN_COMM.wAddDiagCode          = WORD#16#0000 AND

                udtExample.udtCanMessages.CanTx[0].xUsed        = TRUE AND
                udtExample.udtCanMessages.CanTx[0].diID         = DINT#101 AND
                udtExample.udtCanMessages.CanTx[0].udiSequence  = UDINT#0 AND
                udtExample.udtCanMessages.CanTx[0].iDLC         = 8 AND
                udtExample.udtCanMessages.CanTx[0].arrData[1]   = BYTE#16#01 AND
                udtExample.udtCanMessages.CanTx[0].arrData[2]   = BYTE#16#23 AND
                udtExample.udtCanMessages.CanTx[0].arrData[3]   = BYTE#16#45 AND
                udtExample.udtCanMessages.CanTx[0].arrData[4]   = BYTE#16#67 AND
                udtExample.udtCanMessages.CanTx[0].arrData[5]   = BYTE#16#89 AND
                udtExample.udtCanMessages.CanTx[0].arrData[6]   = BYTE#16#AB AND
                udtExample.udtCanMessages.CanTx[0].arrData[7]   = BYTE#16#CD AND
                udtExample.udtCanMessages.CanTx[0].arrData[8]   = BYTE#16#EF
            THEN
                (* The message that we expected was received. Now we can send
                the same message again by just setting xUsed in the
                sending array to TRUE again. *)
                udtExample.udtCanMessages.CanRx[1].xUsed    := TRUE;

                udtExample.iState   := 60;
            END_IF;

        60: (* Wait for a confirmation *)
            IF
                udtExample.udtIL_CAN_COMM.xReady        = TRUE AND
                udtExample.udtIL_CAN_COMM.xError        = FALSE AND
                udtExample.udtIL_CAN_COMM.wDiagCode     = WORD#16#8000 AND
                udtExample.udtIL_CAN_COMM.wAddDiagCode  = WORD#16#0000 AND
                udtExample.udtIL_CAN_COMM.xDone         = TRUE
            THEN
                (* Message has been send successfully.
                Now we wait again for the received message.
                The message with the same ID will be stacked, so we will
                check if the udiSequence has now the value of 2. *)
                udtExample.iState   := 70;
            END_IF;

        70: (* Wait for a message *)
```

```
        IF
            udtExample.udtIL_CAN_COMM.xReady                    = TRUE AND
            udtExample.udtIL_CAN_COMM.xError                    = FALSE AND
            udtExample.udtIL_CAN_COMM.wDiagCode                 = WORD#16#8000 AND
            udtExample.udtIL_CAN_COMM.wAddDiagCode              = WORD#16#0000 AND

            udtExample.udtCanMessages.CanTx[0].xUsed            = TRUE AND
            udtExample.udtCanMessages.CanTx[0].diID             = DINT#101 AND
            udtExample.udtCanMessages.CanTx[0].udiSequence      = UDINT#1 AND
            udtExample.udtCanMessages.CanTx[0].iDLC             = 8 AND
            udtExample.udtCanMessages.CanTx[0].arrData[1]       = BYTE#16#01 AND
            udtExample.udtCanMessages.CanTx[0].arrData[2]       = BYTE#16#23 AND
            udtExample.udtCanMessages.CanTx[0].arrData[3]       = BYTE#16#45 AND
            udtExample.udtCanMessages.CanTx[0].arrData[4]       = BYTE#16#67 AND
            udtExample.udtCanMessages.CanTx[0].arrData[5]       = BYTE#16#89 AND
            udtExample.udtCanMessages.CanTx[0].arrData[6]       = BYTE#16#AB AND
            udtExample.udtCanMessages.CanTx[0].arrData[7]       = BYTE#16#CD AND
            udtExample.udtCanMessages.CanTx[0].arrData[8]       = BYTE#16#EF
        THEN
            (* The message that we expected was received.
            Now we can deactivate the function block *)

            udtExample.udtIL_CAN_COMM.xActivate := FALSE;

            udtExample.iState   := 80;
        END_IF;

    80: (* Wait until the function block is deactivated *)
        IF
            udtExample.udtIL_CAN_COMM.xReady        = FALSE AND
            udtExample.udtIL_CAN_COMM.xError        = FALSE AND
            udtExample.udtIL_CAN_COMM.wDiagCode     = WORD#16#0000 AND
            udtExample.udtIL_CAN_COMM.wAddDiagCode  = WORD#16#0000
        THEN
            (* Reset everything we set *)
            udtExample.udtIL_CAN_COMM.bConf.X0                  := FALSE;
            udtExample.udtIL_CAN_COMM.bConf.X1                  := FALSE;
            udtExample.udtIL_CAN_COMM.bConf.X2                  := FALSE;
            udtExample.udtIL_CAN_COMM.bConf.X3                  := FALSE;
            udtExample.udtIL_CAN_COMM.bConf.X4                  := FALSE;
            udtExample.udtIL_CAN_COMM.bConf.X5                  := FALSE;
            udtExample.udtIL_CAN_COMM.bConf.X6                  := FALSE;
            udtExample.udtIL_CAN_COMM.bConf.X7                  := FALSE;
            udtExample.udtIL_CAN_COMM.xSendMessage              := FALSE;

            udtExample.udtCanMessages.CanRx[1].diID             := DINT#0;
            udtExample.udtCanMessages.CanRx[1].iDLC             := 0;
            udtExample.udtCanMessages.CanRx[1].arrData[1]       := BYTE#16#00;
            udtExample.udtCanMessages.CanRx[1].arrData[2]       := BYTE#16#00;
            udtExample.udtCanMessages.CanRx[1].arrData[3]       := BYTE#16#00;
            udtExample.udtCanMessages.CanRx[1].arrData[4]       := BYTE#16#00;
            udtExample.udtCanMessages.CanRx[1].arrData[5]       := BYTE#16#00;
            udtExample.udtCanMessages.CanRx[1].arrData[6]       := BYTE#16#00;
            udtExample.udtCanMessages.CanRx[1].arrData[7]       := BYTE#16#00;
            udtExample.udtCanMessages.CanRx[1].arrData[8]       := BYTE#16#00;

            udtExample.udtCanMessages.CanTx[0].xUsed            := FALSE;
            udtExample.udtCanMessages.CanTx[0].diID             := DINT#0;
            udtExample.udtCanMessages.CanTx[0].udiSequence      := UDINT#0;
            udtExample.udtCanMessages.CanTx[0].iDLC             := 0;
            udtExample.udtCanMessages.CanTx[0].arrData[1]       := BYTE#16#00;
            udtExample.udtCanMessages.CanTx[0].arrData[2]       := BYTE#16#00;
            udtExample.udtCanMessages.CanTx[0].arrData[3]       := BYTE#16#00;
            udtExample.udtCanMessages.CanTx[0].arrData[4]       := BYTE#16#00;
```

```
                udtExample.udtCanMessages.CanTx[0].arrData[5]   := BYTE#16#00;
                udtExample.udtCanMessages.CanTx[0].arrData[6]   := BYTE#16#00;
                udtExample.udtCanMessages.CanTx[0].arrData[7]   := BYTE#16#00;
                udtExample.udtCanMessages.CanTx[0].arrData[8]   := BYTE#16#00;

                udtExample.iState   := 1000;
            END_IF;

        1000: (* Example finished *)
            udtExample.iExample := 32000;
            udtExample.iState   := 0;

    END_CASE;

END_IF;
```

14.3 Example 3: CAN_*_EXA_AXL_CO

This example describes the use of the IL_CO function blocks with the AXL_CAN_COMM and the AXL F IF CAN module.

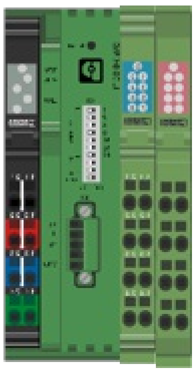
14.3.1 Plant

For this example, the following hardware is used:

- AXC F 2152 (2404267)
- AXL F IF CAN 1H (2702668)

Also, the following external hardware is used.

- IL CO BK-PAC (2702230)
- IB IL 24 DI8/HD-PAC (2700173)
- IB IL 24 DO8/HD-PAC (2700172)



14.3.2 Example description

The CANopen bus coupler is in the same CANbus and has the node ID of 2.

The project shows one startup example for each function block. They can be found inside the Example_Machine function block. There are state machines for every step we have to take care of when using one block.

For starting the example, we have to go in every code sheet and adjust the setup of the function blocks for our settings.

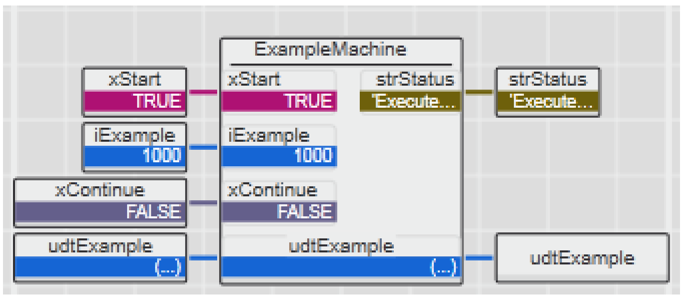
The following examples can be executed:

Function	iExample	Codesheet
This example uses the IL_CO_EMCY function block to receive emergency messages.	1000	E_1000
This example uses the IL_CO_NMT_Guard function block to set and read out the operation mode of the node 2.	2000	E_2000
This example uses the IL_CO_NMT function block to set the operation mode of the node 2.	3000	E_3000
This example uses the IL_CO_NodeGuard function block to read out the operation mode of the node 2.	4000	E_4000
This example uses the IL_CO_NodeInfo function block to read out information from the node 2.	5000	E_5000
This example uses the IL_CO_RD_WR function block to send a SDO and displays the answer.	6000	E_6000
This example uses the IL_CO_SDO_RD function block to Read out a SDO and displays the answer.	7000	E_7000
This example uses the IL_CO_SDO_WR function block to send a SDO and display the answer.	8000	E_8000
This example uses the IL_CO_PDO_RD function block to display the configurated PDO when it is received.	9000	E_9000
This example uses the IL_CO_PDO_WR function block to build a PDO and send it the the selected destination.	10000	E_10000
This example uses the IL_CO_Search function block to find all CANopen nodes in the CAN bus.	11000	E_11000
This example uses the IL_CO_SYNC function block to send a synchronization message to the CAN bus.	12000	E_12000

Note : The order for calling the function blocks is important. First the AXL_CAN_COMM must be called, then the CAN_TO_IL_STRUCT function block, followed by the desired IL_CO function block(s) and finally the CAN_TO_AXL_STRUCT function block.

14.3.2.1 Example machine

For executing the desired example start the function block by setting xStart to TRUE.



14.3.2.2 State machine: E_1000

```
(* IL_CO_EMCY *)
IF udtExample.iExample  = 1000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_EMCY *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_EMCY.xActivate  := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_EMCY ready *)
            IF udtExample.udtIL_CO_EMCY.xReady  = TRUE THEN
                udtExample.iState   := 30;
            END_IF;

        30: (* Now emergecy messages can be received and are displayed at the
            outputs *)
            IF udtExample.udtIL_CO_EMCY.xNDR    = TRUE THEN
                udtExample.iState   := 40;
            END_IF;

        40: (* Deactivate and continue with the example *)
            strStatus   := 'Rising edge on xContinue to deactivate the IL_CO_EMCY.';
            IF R_TRIG_Continue.Q    = TRUE THEN
                strStatus                           := '';
                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
                udtExample.udtIL_CO_EMCY.xActivate      := FALSE;

                udtExample.iState   := 50;
            END_IF;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState    := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;
    END_CASE;

END_IF;
```

14.3.2.3 State machine: E_2000

```
(* IL_CO_NMT_Guard *)
IF udtExample.iExample  = 2000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_NMT_Guard *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_NMT_Guard.xActivate := TRUE;
                udtExample.udtIL_CO_NMT_Guard.diNodeID  := DINT#2;
                udtExample.udtIL_CO_NMT_Guard.bOP_Mode  := BYTE#16#01;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_NMT_Guard ready and execute. Change operation mode of
            node 2 to operational mode *)
            IF udtExample.udtIL_CO_NMT_Guard.xReady = TRUE THEN
                udtExample.udtIL_CO_NMT_Guard.xExec := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xDone is TRUE, check if the results are correct and everthing
            can be deactivated *)
            IF udtExample.udtIL_CO_NMT_Guard.xDone   = TRUE THEN
                IF udtExample.udtIL_CO_NMT_Guard.diNodeID_RD    = DINT#2 AND
                    udtExample.udtIL_CO_NMT_Guard.xModeBoot     = FALSE AND
                    udtExample.udtIL_CO_NMT_Guard.xModeStop     = FALSE AND
                    udtExample.udtIL_CO_NMT_Guard.xModePreOP    = FALSE AND
                    udtExample.udtIL_CO_NMT_Guard.xModeOP       = TRUE
                THEN
                    udtExample.udtIL_CO_NMT_Guard.xActivate := FALSE;
                    udtExample.udtIL_CO_NMT_Guard.diNodeID  := DINT#0;
                    udtExample.udtIL_CO_NMT_Guard.bOP_Mode  := BYTE#16#00;
                    udtExample.udtIL_CO_NMT_Guard.xExec     := FALSE;

                    udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

                    udtExample.iState   := 40;
                END_IF;
            END_IF;

        40: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.4 State machine: E_3000

```
(* IL_CO_NMT *)
IF udtExample.iExample  = 3000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_NMT *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_NMT.xActivate   := TRUE;
                udtExample.udtIL_CO_NMT.diNodeID    := DINT#2;
                udtExample.udtIL_CO_NMT.bOP_Mode    := BYTE#16#01;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_NMT ready and execute. Change operation mode of
            node 2 to operational mode *)
            IF udtExample.udtIL_CO_NMT.xReady   = TRUE THEN
                udtExample.udtIL_CO_NMT.xExec   := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xDone is TRUE, the function block is finished and everthing
            can be deactivated *)
            IF udtExample.udtIL_CO_NMT.xDone    = TRUE THEN
                udtExample.udtIL_CO_NMT.xActivate   := FALSE;
                udtExample.udtIL_CO_NMT.diNodeID    := DINT#0;
                udtExample.udtIL_CO_NMT.bOP_Mode    := BYTE#16#00;
                udtExample.udtIL_CO_NMT.xExec       := FALSE;

                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

                udtExample.iState   := 40;
            END_IF;

        40: (* Wait for AXL_CAN_COMM to be deactivated *)

            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
```

```
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.5 State machine: E_4000

```
(* IL_CO_NodeGuard *)
IF udtExample.iExample  = 4000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_NodeGuard *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_NodeGuard.xActivate := TRUE;
                udtExample.udtIL_CO_NodeGuard.diNodeID  := DINT#2;
                udtExample.udtIL_CO_NodeGuard.tTimeOut  := t#5s;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_NodeGuard ready and execute *)
            IF udtExample.udtIL_CO_NodeGuard.xReady = TRUE THEN
                udtExample.udtIL_CO_NodeGuard.xExec := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xNDR is TRUE, the function block shows the current operation
            mode of node 2 *)
            IF udtExample.udtIL_CO_NodeGuard.xNDR   = TRUE THEN
                strStatus           := 'Rising edge on xContinue deactivates the IL_CO_NodeGuarc
                udtExample.iState   := 40;
            END_IF;

        40: (* Deactivate and continue with the example *)
            IF R_TRIG_Continue.Q    = TRUE THEN
                strStatus                               := '';
                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
                udtExample.udtIL_CO_NodeGuard.xActivate := FALSE;
                udtExample.udtIL_CO_NodeGuard.diNodeID  := DINT#0;
                udtExample.udtIL_CO_NodeGuard.tTimeOut  := t#0s;

                udtExample.iState   := 50;
            END_IF;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.6 State machine: E_5000

```
(* IL_CO_NodeInfo *)
IF udtExample.iExample  = 5000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_NodeInfo *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_NodeInfo.xActivate  := TRUE;
                udtExample.udtIL_CO_NodeInfo.diNodeID   := DINT#2;
                udtExample.udtIL_CO_NodeInfo.tTimeOut   := t#5s;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_NodeInfo ready and execute *)
            IF udtExample.udtIL_CO_NodeInfo.xReady  = TRUE THEN
                udtExample.udtIL_CO_NodeInfo.xExec  := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xDone is TRUE, the function block shows the info for node 2 *)
            IF udtExample.udtIL_CO_NodeInfo.xDone   = TRUE THEN
                strStatus           := 'Rising edge on xContinue to deactivate the IL_CO_NodeInf
                udtExample.iState   := 40;
            END_IF;

        40: (* Deactivate and continue with the example *)
            IF R_TRIG_Continue.Q    = TRUE THEN
                strStatus                               := '';
                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
                udtExample.udtIL_CO_NodeInfo.xActivate  := FALSE;
                udtExample.udtIL_CO_NodeInfo.diNodeID   := DINT#0;
                udtExample.udtIL_CO_NodeInfo.tTimeOut   := t#0s;

                udtExample.iState   := 50;
            END_IF;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;
        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.7 State machine: E_6000

```
(* IL_CO_RD_WR *)
IF udtExample.iExample  = 6000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_RD_WR *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_RD_WR.xActivate := TRUE;
                udtExample.udtIL_CO_RD_WR.diNodeID  := DINT#2;
                udtExample.udtIL_CO_RD_WR.bByte1    := BYTE#16#2F;
                udtExample.udtIL_CO_RD_WR.bByte2    := BYTE#16#00;
                udtExample.udtIL_CO_RD_WR.bByte3    := BYTE#16#18;
                udtExample.udtIL_CO_RD_WR.bByte4    := BYTE#16#02;
                udtExample.udtIL_CO_RD_WR.bByte5    := BYTE#16#00;
                udtExample.udtIL_CO_RD_WR.bByte6    := BYTE#16#00;
                udtExample.udtIL_CO_RD_WR.bByte7    := BYTE#16#00;
                udtExample.udtIL_CO_RD_WR.bByte8    := BYTE#16#00;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_RD_WR ready and execute *)
            IF udtExample.udtIL_CO_RD_WR.xReady = TRUE THEN
                udtExample.udtIL_CO_RD_WR.xSend := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xDone is TRUE, the function block shows the answer and we
            can check it *)
            IF udtExample.udtIL_CO_RD_WR.xDone  = TRUE THEN
                IF udtExample.udtIL_CO_RD_WR.bByte1_RD  = BYTE#16#60 AND
                    udtExample.udtIL_CO_RD_WR.bByte2_RD = BYTE#16#00 AND
                    udtExample.udtIL_CO_RD_WR.bByte3_RD = BYTE#16#18 AND
                    udtExample.udtIL_CO_RD_WR.bByte4_RD = BYTE#16#02 AND
                    udtExample.udtIL_CO_RD_WR.bByte5_RD = BYTE#16#00 AND
                    udtExample.udtIL_CO_RD_WR.bByte6_RD = BYTE#16#00 AND
                    udtExample.udtIL_CO_RD_WR.bByte7_RD = BYTE#16#00 AND
                    udtExample.udtIL_CO_RD_WR.bByte8_RD = BYTE#16#00
                THEN
                    udtExample.udtIL_CO_RD_WR.xActivate := FALSE;
                    udtExample.udtIL_CO_RD_WR.diNodeID  := DINT#0;
                    udtExample.udtIL_CO_RD_WR.bByte1    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.bByte2    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.bByte3    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.bByte4    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.bByte5    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.bByte6    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.bByte7    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.bByte8    := BYTE#16#00;
                    udtExample.udtIL_CO_RD_WR.xSend     := FALSE;

                    udtExample.iState   := 40;
                END_IF;
            END_IF;

        40: (* Deactivate and continue with the example *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

            udtExample.iState   := 50;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.8 State machine: E_7000

```
(* IL_CO_SDO_RD *)
IF udtExample.iExample  = 7000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_SDO_RD *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_SDO_RD.xActivate    := TRUE;
                udtExample.udtIL_CO_SDO_RD.diNodeID     := DINT#2;
                udtExample.udtIL_CO_SDO_RD.wIndex       := WORD#16#1800;
                udtExample.udtIL_CO_SDO_RD.bSubIndex    := BYTE#16#02;
                udtExample.udtIL_CO_SDO_RD.tTimeOut     := t#5s;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_SDO_RD ready and execute *)
            IF udtExample.udtIL_CO_SDO_RD.xReady    = TRUE THEN
                udtExample.udtIL_CO_SDO_RD.xExec    := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xNDR is TRUE, the function block shows the answer and we can
            deactivate it *)
            IF udtExample.udtIL_CO_SDO_RD.xNDR  = TRUE THEN
                udtExample.udtIL_CO_SDO_RD.xActivate    := FALSE;
                udtExample.udtIL_CO_SDO_RD.diNodeID     := DINT#0;
                udtExample.udtIL_CO_SDO_RD.wIndex       := WORD#0000;
                udtExample.udtIL_CO_SDO_RD.bSubIndex    := BYTE#16#00;
                udtExample.udtIL_CO_SDO_RD.tTimeOut     := t#0s;
                udtExample.udtIL_CO_SDO_RD.xExec        := FALSE;

                udtExample.iState   := 40;
            END_IF;

        40: (* Deactivate and continue with the example *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

            udtExample.iState   := 50;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;
                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;
END_IF;
```

14.3.2.9 State machine: E_8000

```
(* IL_CO_SDO_WR *)
IF udtExample.iExample  = 8000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_SDO_WR *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_SDO_WR.diNodeID     := DINT#2;
                udtExample.udtIL_CO_SDO_WR.tTimeOut     := t#5s;
                udtExample.udtIL_CO_SDO_WR.wIndex       := WORD#16#1800;
                udtExample.udtIL_CO_SDO_WR.bSubIndex    := BYTE#16#02;
                udtExample.udtIL_CO_SDO_WR.bData1       := BYTE#16#FE;
                udtExample.udtIL_CO_SDO_WR.bData2       := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.bData3       := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.bData4       := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.xActivate    := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_SDO_WR ready and execute *)
            IF udtExample.udtIL_CO_SDO_WR.xReady    = TRUE THEN
                udtExample.udtIL_CO_SDO_WR.xExec    := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xDone is TRUE, the function block is finished and we can
            deactivate it *)
            IF udtExample.udtIL_CO_SDO_WR.xDone = TRUE THEN
                udtExample.udtIL_CO_SDO_WR.xExec        := FALSE;
                udtExample.udtIL_CO_SDO_WR.diNodeID     := DINT#0;
                udtExample.udtIL_CO_SDO_WR.tTimeOut     := t#0s;
                udtExample.udtIL_CO_SDO_WR.wIndex       := WORD#16#0000;
                udtExample.udtIL_CO_SDO_WR.bSubIndex    := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.bData1       := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.bData2       := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.bData3       := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.bData4       := BYTE#16#00;
                udtExample.udtIL_CO_SDO_WR.xActivate    := FALSE;

                udtExample.iState   := 40;
            END_IF;

        40: (* Deactivate and continue with the example *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
            udtExample.iState   := 50;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.10 State machine: E_9000

```
(* IL_CO_PDO_RD *)
IF udtExample.iExample  = 9000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_PDO_RD *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_PDO_RD.xActivate    := TRUE;
                udtExample.udtIL_CO_PDO_RD.diNodeID     := DINT#2;
                udtExample.udtIL_CO_PDO_RD.wCOB         := WORD#16#0180;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_PDO_RD ready and execute *)
            IF udtExample.udtIL_CO_PDO_RD.xReady    = TRUE THEN

                udtExample.iState   := 30;
            END_IF;

        30: (* Now PDO messages can be received and are displayed at the outputs *)
            IF udtExample.udtIL_CO_PDO_RD.xNDR  = TRUE THEN

                udtExample.iState   := 40;
            END_IF;

        40: (* Deactivate and continue with the example *)
            strStatus   := 'Rising edge on xContinue to deactivate the IL_CO_PDO_RD.';
            IF R_TRIG_Continue.Q    = TRUE THEN
                strStatus                               := '';
                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
                udtExample.udtIL_CO_PDO_RD.xActivate    := FALSE;
                udtExample.udtIL_CO_PDO_RD.diNodeID     := DINT#0;
                udtExample.udtIL_CO_PDO_RD.wCOB         := WORD#16#0000;

                udtExample.iState   := 50;
            END_IF;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;
        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.11 State machine: E_10000

```
(* IL_CO_PDO_WR *)
IF udtExample.iExample  = 10000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_PDO_WR *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_PDO_WR.diNodeID := DINT#2;
                udtExample.udtIL_CO_PDO_WR.iDLC     := 8;
                udtExample.udtIL_CO_PDO_WR.wCOB     := WORD#16#0750;
                udtExample.udtIL_CO_PDO_WR.bData0   := BYTE#16#81;
                udtExample.udtIL_CO_PDO_WR.bData1   := BYTE#16#30;
                udtExample.udtIL_CO_PDO_WR.bData2   := BYTE#16#04;
                udtExample.udtIL_CO_PDO_WR.bData3   := BYTE#16#C1;
                udtExample.udtIL_CO_PDO_WR.bData4   := BYTE#16#C2;
                udtExample.udtIL_CO_PDO_WR.bData5   := BYTE#16#C3;
                udtExample.udtIL_CO_PDO_WR.bData6   := BYTE#16#C4;
                udtExample.udtIL_CO_PDO_WR.bData7   := BYTE#16#C5;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_PDO_WR ready and execute *)
            IF udtExample.udtIL_CO_PDO_WR.xReady = TRUE THEN
                udtExample.udtIL_CO_PDO_WR.xExecute := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30: (* After xDone is TRUE, the function block is finished and we can
            deactivate it *)
            IF udtExample.udtIL_CO_PDO_WR.xDone = TRUE THEN
                udtExample.udtIL_CO_PDO_WR.diNodeID     := DINT#0;
                udtExample.udtIL_CO_PDO_WR.iDLC         := 0;
                udtExample.udtIL_CO_PDO_WR.wCOB         := WORD#16#0000;
                udtExample.udtIL_CO_PDO_WR.bData0       := BYTE#16#00;
                udtExample.udtIL_CO_PDO_WR.bData1       := BYTE#16#00;
                udtExample.udtIL_CO_PDO_WR.bData2       := BYTE#16#00;
                udtExample.udtIL_CO_PDO_WR.bData3       := BYTE#16#00;
                udtExample.udtIL_CO_PDO_WR.bData4       := BYTE#16#00;
                udtExample.udtIL_CO_PDO_WR.bData5       := BYTE#16#00;
                udtExample.udtIL_CO_PDO_WR.bData6       := BYTE#16#00;
                udtExample.udtIL_CO_PDO_WR.bData7       := BYTE#16#00;
                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

                udtExample.iState   := 40;
            END_IF;
        40: (* Deactivate and continue with the example *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

            udtExample.iState   := 50;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.12 State machine: E_11000

```
(* IL_CO_Search *)
IF udtExample.iExample  = 11000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_Search *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_SEARCH.xActivate    := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* After xDone is TRUE, the function block is finished and all found
            nodes can be found in arrNames Then we can deactivate everything *)
            IF udtExample.udtIL_CO_SEARCH.xDone = TRUE THEN
                udtExample.udtIL_CO_SEARCH.xActivate    := FALSE;

                udtExample.iState   := 30;
            END_IF;

        30: (* Deactivate and continue with the example *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

            udtExample.iState   := 40;

        40: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;

    END_CASE;

END_IF;
```

14.3.2.13 State machine: E_12000

```
(* IL_CO_SYNC *)
IF udtExample.iExample  = 12000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;

            udtExample.iState   := 10;

        10: (* Activate IL_CO_SYNC *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = TRUE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#5s;
                udtExample.udtAXL_CAN_COMM.xSend        := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;

                udtExample.udtIL_CO_SYNC.xActivate  := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for IL_CO_SYNC ready and execute *)
            IF udtExample.udtIL_CO_SYNC.xReady  = TRUE THEN
                udtExample.udtIL_CO_SYNC.xExec  := TRUE;

                udtExample.iState   := 30;
            END_IF;

        30 : (* After xDone is TRUE, the function block is finished and we
            can deactivate it *)
            IF udtExample.udtIL_CO_SYNC.xDone   = TRUE THEN
                udtExample.udtIL_CO_SYNC.xExec      := FALSE;
                udtExample.udtIL_CO_SYNC.xActivate  := FALSE;

                udtExample.iState   := 40;
            END_IF;

        40: (* Deactivate and continue with the example *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;

            udtExample.iState   := 50;

        50: (* Wait for AXL_CAN_COMM to be deactivated *)
            IF udtExample.udtAXL_CAN_COMM.xActive   = FALSE THEN
                udtExample.udtAXL_CAN_COMM.tBusTimeout  := T#0s;
                udtExample.udtAXL_CAN_COMM.xSend        := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;

                udtExample.iState   := 999;
            END_IF;

        999 : (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 15000;
    END_CASE;

END_IF;
```

14.4 Example 4: CAN_*_EXA_AXL_J1939

This example describes the use of the IL_J1939_RD, IL_J1939_RD_Multi and IL_J1939_WR with the AXL_CAN_COMM function block.

14.4.1 Plant

For this example, the following hardware is used:

- AXC F 2152 (2404267)
- AXL F IF CAN 1H (2702668)

14.4.2 Example description

The project contains one startup example for each function block. They can be found inside the ExampleMachine function block.

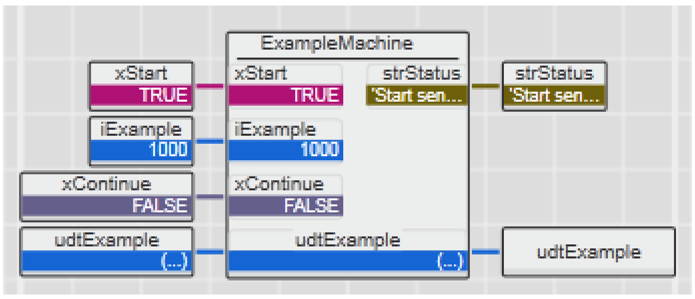
There are state machines for every step we have to take care of when using one block.

The following examples can be executed :

iExample	Codesheet	Description
1000	E_1000	Receive messages with the IL_J1939_RD.
2000	E_2000	Send messages with the IL_J1939_WR.
3000	E_3000	Receive multi-messages with the IL_J1939_RD_Multi.

14.4.2.1 Example machine

We can execute the desired example by selecting iExample and setting xStart to TRUE.



14.4.2.2 State machine: E_1000

```
IF udtExample.iExample  = 1000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example ';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate the AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;

            udtExample.iState   := 10;

        10: (* Wait until AXL_CAN_COMM active *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = TRUE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0002
            THEN
                (* Now we want our J1939 device to send messages with a cycle of 1000 ms.
                In our example, the message has the ID 16#18FC31F8 and data bytes of 16#01
                to 16#08 *)
                strStatus   := 'Start sending the message with ID 16#18FC31F8.';

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for manual trigger xContinue *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                strStatus                           := 'Execute E_1000';
                (* Then we activate and configurate the IL_J1939 function block *)
                udtExample.udtIL_J1939_RD.xActivate := TRUE;
                udtExample.udtIL_J1939_RD.dwCAN_ID  := DWORD#16#18FC31F8;
                udtExample.udtIL_J1939_RD.tControl  := t#1100ms;

                udtExample.iState   := 30;
            END_IF;

        30: (* Wait until IL_J1939_RD is ready *)
            IF
                udtExample.udtIL_J1939_RD.xReady    = TRUE AND
                udtExample.udtIL_J1939_RD.xError    = FALSE AND
                udtExample.udtIL_J1939_RD.wDiagCode = WORD#16#8000
            THEN
                udtExample.iState   := 40;
            END_IF;

        40: (* Check if the message was received *)
            IF
                udtExample.udtIL_J1939_RD.xReady    = TRUE AND
                udtExample.udtIL_J1939_RD.xError    = FALSE AND
                udtExample.udtIL_J1939_RD.wDiagCode = WORD#16#8000 AND
                udtExample.udtIL_J1939_RD.bByte1    = BYTE#16#01 AND
                udtExample.udtIL_J1939_RD.bByte2    = BYTE#16#02 AND
                udtExample.udtIL_J1939_RD.bByte3    = BYTE#16#03 AND
                udtExample.udtIL_J1939_RD.bByte4    = BYTE#16#04 AND
                udtExample.udtIL_J1939_RD.bByte5    = BYTE#16#05 AND
                udtExample.udtIL_J1939_RD.bByte6    = BYTE#16#06 AND
                udtExample.udtIL_J1939_RD.bByte7     = BYTE#16#07 AND
                udtExample.udtIL_J1939_RD.bByte8     = BYTE#16#08
            THEN
                (* The xNDR is conncted to a CTU that counts up to 100. This way, we can
                be sure our message is received constantly *)
                IF CTU_NDR.CV    = 100 THEN
                    xResetCTU     := TRUE;

                    udtExample.iState   := 50;
                END_IF;
            END_IF;

        50: (* Reset everything *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
            udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;
            udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;

            udtExample.udtIL_J1939_RD.xActivate := FALSE;
            udtExample.udtIL_J1939_RD.dwCAN_ID  := DWORD#16#00000000;
            udtExample.udtIL_J1939_RD.tControl  := t#0s;

            udtExample.udtCanData.arrMessagesReceive[0].xUsed           := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].diID            := DINT#0;
            udtExample.udtCanData.arrMessagesReceive[0].iDLC            := 0;
            udtExample.udtCanData.arrMessagesReceive[0].udiSequence     := UDINT#0;
            udtExample.udtCanData.arrMessagesReceive[0].xFrameFormat    := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].xFrameType      := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[1]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[2]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[3]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[4]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[5]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[6]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[7]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[8]      := BYTE#16#00;

            xResetCTU   := FALSE;
            strStatus   := 'Stop sending the message with ID 16#18FC31F8.';

            udtExample.iState   := 60;

        60: (* Wait for manual trigger *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                udtExample.iState   := 70;
            END_IF;


        70: (* Check if the function blocks are deactivated *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = FALSE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#0000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0000 AND
                udtExample.udtIL_J1939_RD.xReady        = FALSE AND
                udtExample.udtIL_J1939_RD.xError        = FALSE AND
                udtExample.udtIL_J1939_RD.wDiagCode     = WORD#16#0000
            THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 32000;
        END_CASE;

END_IF;
```

14.4.2.3 State machine: E_2000

```
IF udtExample.iExample  = 2000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example ';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate the AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;
            udtExample.udtAXL_CAN_COMM.xSend        := TRUE;

            udtExample.iState   := 10;

        10: (* Wait until AXL_CAN_COMM is active *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = TRUE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0001
            THEN
                (* Now we prepare an example J1939 message that we want to send *)
                udtExample.udtIL_J1939_WR.dwCAN_ID      := DWORD#16#18FC13F9;
                udtExample.udtIL_J1939_WR.iDataLength   := 8;
                (* For this example, the data are just random *)
                udtExample.udtIL_J1939_WR.bByte1        := BYTE#16#10;
                udtExample.udtIL_J1939_WR.bByte2        := BYTE#16#20;
                udtExample.udtIL_J1939_WR.bByte3        := BYTE#16#30;
                udtExample.udtIL_J1939_WR.bByte4        := BYTE#16#40;
                udtExample.udtIL_J1939_WR.bByte5        := BYTE#16#50;
                udtExample.udtIL_J1939_WR.bByte6        := BYTE#16#60;
                udtExample.udtIL_J1939_WR.bByte7        := BYTE#16#70;
                udtExample.udtIL_J1939_WR.bByte8        := BYTE#16#80;
                (* Here we set the cycle time of the message *)
                udtExample.udtIL_J1939_WR.tInterval     := t#1s;
                udtExample.udtIL_J1939_WR.xActivate     := TRUE;

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait until IL_J1939_WR is active *)
            IF
                udtExample.udtIL_J1939_WR.xReady    = TRUE AND
                udtExample.udtIL_J1939_WR.xError    = FALSE AND
                udtExample.udtIL_J1939_WR.wDiagCode = WORD#16#8000
            THEN
                (* The xDone output, that shows that the message was send, is connected
                to a CTU. That way, we wait until we send the message 100 times. *)
                IF CTU_DONE.CV  = 100 THEN
                    udtExample.udtIL_J1939_WR.xActivate := FALSE;
                    xResetCTU                           := TRUE;

                    udtExample.iState   := 30;
                END_IF;
            END_IF;

        30: (* Reset everything *)
            xResetCTU   := FALSE;

            udtExample.udtIL_J1939_WR.dwCAN_ID      := DWORD#16#00000000;
            udtExample.udtIL_J1939_WR.iDataLength   := 0;
            udtExample.udtIL_J1939_WR.bByte1         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.bByte2         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.bByte3         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.bByte4         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.bByte5         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.bByte6         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.bByte7         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.bByte8         := BYTE#16#00;
            udtExample.udtIL_J1939_WR.tInterval      := t#0s;

            udtExample.udtAXL_CAN_COMM.xActivate     := FALSE;
            udtExample.udtAXL_CAN_COMM.xSend         := FALSE;

            udtExample.udtCanData.arrMessagesSend[0].xUsed          := FALSE;
            udtExample.udtCanData.arrMessagesSend[0].diID           := DINT#0;
            udtExample.udtCanData.arrMessagesSend[0].iDLC           := 0;
            udtExample.udtCanData.arrMessagesSend[0].udiSequence    := UDINT#0;
            udtExample.udtCanData.arrMessagesSend[0].xFrameFormat   := FALSE;
            udtExample.udtCanData.arrMessagesSend[0].xFrameType     := FALSE;
            udtExample.udtCanData.arrMessagesSend[0].arrData[1]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[2]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[3]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[4]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[5]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[6]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[7]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[8]     := BYTE#16#00;

            udtExample.iState   := 40;

        40: (* Check if the function blocks are deactivated *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = FALSE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#0000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0000 AND
                udtExample.udtIL_J1939_WR.xReady        = FALSE AND
                udtExample.udtIL_J1939_WR.xError        = FALSE AND
                udtExample.udtIL_J1939_WR.wDiagCode     = WORD#16#0000
            THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 32000;

    END_CASE;

END_IF;
```

14.4.2.4 State machine: E_3000

```
IF udtExample.iExample  = 3000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example ';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate the AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;

            udtExample.iState   := 10;

        10: (* Wait ultil AXL_CAN_COMM is active *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = TRUE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0002
            THEN
                (* For this example, we prepared a J1939 TPCM BAM message that is send
                in a 500ms cycle. Also we prepared 3 TPDT messages. The TCPM message has the
                ID 16#04ECFF02 and the TPDT message 16#04EBFF02. *)
                strStatus   := 'Start sending the TPCM.BAM message.';

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for manual trigger *)
            IF R_TRIG_CONTINUE.Q THEN
                (* After the TCPM message is sending, we can activate the function block *)
                strStatus                                 := 'Execute E_3000';
                udtExample.udtIL_J1939_RD_MULTI.xActivate   := TRUE;
                udtExample.udtIL_J1939_RD_MULTI.dwCAN_ID    := DWORD#16#04ECFF02;
                (* Here we select which of the packeges we want to show at the outputs *)
                udtExample.udtIL_J1939_RD_MULTI.bPackage    := BYTE#16#01;
                udtExample.udtIL_J1939_RD_MULTI.tControl    := t#600ms;

                udtExample.iState   := 30;
            END_IF;

        30: (* Wait ultil IL_J1939_RD_MULTI is ready *)
            IF
                udtExample.udtIL_J1939_RD_MULTI.xReady      = TRUE AND
                udtExample.udtIL_J1939_RD_MULTI.xError      = FALSE AND
                udtExample.udtIL_J1939_RD_MULTI.wDiagCode   = WORD#16#8000
            THEN
                (* Now we can send the first TPDT message. It has the ID 16#04EBFF02 and
                8 data bytes of 16#01, 16#F1, 16#F1, 16#F1, 16#F1, 16#F1, 16#F1 and 16#F1. *)
                strStatus   := 'Send the first TPDT message once.';

                udtExample.iState   := 40;
            END_IF;

        40: (* Wait for manual trigger and check if the message was received *)
            IF R_TRIG_CONTINUE.Q THEN
                strStatus   := 'Execute E_3000.';
                IF
                    udtExample.udtIL_J1939_RD_MULTI.xReady      = TRUE AND
                    udtExample.udtIL_J1939_RD_MULTI.xError      = FALSE AND
                    udtExample.udtIL_J1939_RD_MULTI.wDiagCode   = WORD#16#8000 AND
                    udtExample.udtIL_J1939_RD_MULTI.bPackageMAX = BYTE#16#03 AND
                    (* Package number *)
                    udtExample.udtIL_J1939_RD_MULTI.bByte1      = BYTE#16#01 AND
                    (* Data bytes 1 - 7 *)
                    udtExample.udtIL_J1939_RD_MULTI.bByte2      = BYTE#16#F1 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte3      = BYTE#16#F1 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte4      = BYTE#16#F1 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte5      = BYTE#16#F1 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte6      = BYTE#16#F1 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte7      = BYTE#16#F1 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte8      = BYTE#16#F1
                THEN
                    udtExample.iState   := 50;
                END_IF;
            END_IF;

        50: (* Send second TPDT message *)
            (* Now we can select that we want to see the second package and send
            the second TPDT message.
            It has the ID 16#04EBFF02 and 8 data bytes of 16#02, 16#F2, 16#F2,
            16#F2, 16#F2, 16#F2, 16#F2 and 16#F2. *)
            udtExample.udtIL_J1939_RD_MULTI.bPackage    := BYTE#16#02;

            strStatus   := 'Send the second TPDT message once.';

            udtExample.iState   := 60;

        60: (* Wait for manual trigger and check if the message was received *)
            IF R_TRIG_CONTINUE.Q THEN
                strStatus   := 'Execute E_3000.';
                IF
                    udtExample.udtIL_J1939_RD_MULTI.xReady      = TRUE AND
                    udtExample.udtIL_J1939_RD_MULTI.xError      = FALSE AND
                    udtExample.udtIL_J1939_RD_MULTI.wDiagCode   = WORD#16#8000 AND
                    udtExample.udtIL_J1939_RD_MULTI.bPackageMAX = BYTE#16#03 AND
                    (* Package number *)
                    udtExample.udtIL_J1939_RD_MULTI.bByte1  = BYTE#16#02 AND
                    (* Data bytes 1 - 7 *)
                    udtExample.udtIL_J1939_RD_MULTI.bByte2  = BYTE#16#F2 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte3  = BYTE#16#F2 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte4  = BYTE#16#F2 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte5  = BYTE#16#F2 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte6  = BYTE#16#F2 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte7  = BYTE#16#F2 AND
                    udtExample.udtIL_J1939_RD_MULTI.bByte8  = BYTE#16#F2
                THEN
                    (* Example finished, reset everything *)
                    udtExample.udtIL_J1939_RD_MULTI.xActivate   := FALSE;

                    strStatus   := 'Stop sending the TPCM.BAM message.';

                    udtExample.iState   := 70;
                END_IF;
            END_IF;

        70: (* Wait for manual trigger *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                strStatus                               := 'Execute E_3000.';
                (* Reset everything *)
                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
```

```
            udtExample.udtIL_J1939_RD_MULTI.dwCAN_ID    := DWORD#16#00000000;
            udtExample.udtIL_J1939_RD_MULTI.bPackage    := BYTE#16#00;
            udtExample.udtIL_J1939_RD_MULTI.tControl    := t#0s;

            udtExample.udtCanData.arrMessagesReceive[0].xUsed           := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].diID            := DINT#0;
            udtExample.udtCanData.arrMessagesReceive[0].iDLC            := 0;
            udtExample.udtCanData.arrMessagesReceive[0].udiSequence     := UDINT#0;
            udtExample.udtCanData.arrMessagesReceive[0].xFrameFormat    := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].xFrameType      := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[1]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[2]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[3]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[4]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[5]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[6]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[7]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[8]      := BYTE#16#00;

            udtExample.udtCanData.arrMessagesReceive[1].xUsed           := FALSE;
            udtExample.udtCanData.arrMessagesReceive[1].diID            := DINT#0;
            udtExample.udtCanData.arrMessagesReceive[1].iDLC            := 0;
            udtExample.udtCanData.arrMessagesReceive[1].udiSequence     := UDINT#0;
            udtExample.udtCanData.arrMessagesReceive[1].xFrameFormat    := FALSE;
            udtExample.udtCanData.arrMessagesReceive[1].xFrameType      := FALSE;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[1]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[2]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[3]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[4]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[5]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[6]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[7]      := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[1].arrData[8]      := BYTE#16#00;

            udtExample.iState   := 80;
        END_IF;

    80: (* Check if the function blocks are deactivated *)
        IF
            udtExample.udtAXL_CAN_COMM.xActive          = FALSE AND
            udtExample.udtAXL_CAN_COMM.xError           = FALSE AND
            udtExample.udtAXL_CAN_COMM.wDiagCode        = WORD#16#0000 AND
            udtExample.udtAXL_CAN_COMM.wAddDiagCode     = WORD#16#0000 AND
            udtExample.udtIL_J1939_RD_MULTI.xReady      = FALSE AND
            udtExample.udtIL_J1939_RD_MULTI.xError      = FALSE AND
            udtExample.udtIL_J1939_RD_MULTI.wDiagCode   = WORD#16#0000
        THEN
            udtExample.iState   := 1000;
        END_IF;

    1000: (* Successful finished *)
        udtExample.iState   := 0;
        udtExample.iExample := 32000;

  END_CASE;

END_IF;
```

14.5 Example 5: CAN_*_EXA_AXL_NMEA

This example describes the use of the IL_NMEA_RD, IL_NMEA_RD_Multi and IL_NMEA_WR with the AXL_CAN_COMM function block.

14.5.1 Plant

For this example, the following hardware is used:

- AXC F 2152 (2404267)
- AXL F IF CAN 1H (2702668)

14.5.2 Example description

The project contains one startup example for each function block. They can be found inside the ExampleMachine function block.

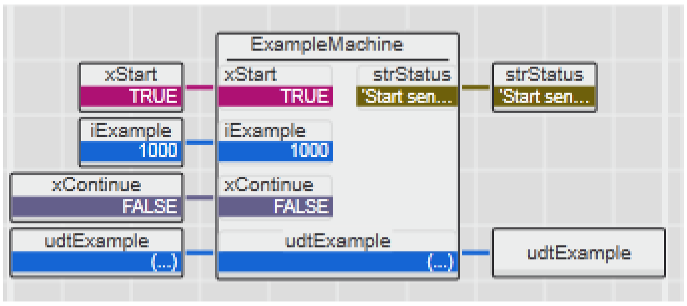
There are state machines for every step we have to take care of when using one block.

The following examples can be executed :

iExample	Codesheet	Description
1000	E_1000	Receive messages with the IL_NMEA_RD.
2000	E_2000	Send messages with the IL_NMEA_WR.
3000	E_3000	Receive multi-messages with the IL_NMEA_RD_Multi.

14.5.2.1 Example machine

We can execute the desired example by selecting iExample and setting xStart to TRUE.



14.5.2.2 State machine: E_1000

```
IF udtExample.iExample  = 1000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example ';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate the AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;

            udtExample.iState   := 10;

        10: (* Wait until AXL_CAN_COMM active *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = TRUE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0002
            THEN
                (* Now we want our NMEA device to send messages with a cycle of 1000 ms.
                In our example, the message has the ID 16#18FC31F8 and data bytes of 16#01
                to 16#08 *)
                strStatus   := 'Start sending the message with ID 16#18FC31F8.';

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for manual trigger xContinue *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                strStatus                           := 'Execute E_1000';
                (* Then we activate and configurate the IL_NMEA function block *)
                udtExample.udtIL_NMEA_RD.xActivate  := TRUE;
                udtExample.udtIL_NMEA_RD.dwCAN_ID   := DWORD#16#18FC31F8;
                udtExample.udtIL_NMEA_RD.tControl   := t#1100ms;

                udtExample.iState   := 30;
            END_IF;

        30: (* Wait until IL_NMEA_RD is ready *)
            IF
                udtExample.udtIL_NMEA_RD.xReady     = TRUE AND
                udtExample.udtIL_NMEA_RD.xError     = FALSE AND
                udtExample.udtIL_NMEA_RD.wDiagCode  = WORD#16#8000
            THEN
                udtExample.iState   := 40;
            END_IF;

        40: (* Check if the message was received *)
            IF
                udtExample.udtIL_NMEA_RD.xReady     = TRUE AND
                udtExample.udtIL_NMEA_RD.xError     = FALSE AND
                udtExample.udtIL_NMEA_RD.wDiagCode  = WORD#16#8000 AND
                udtExample.udtIL_NMEA_RD.bByte1     = BYTE#16#01 AND
                udtExample.udtIL_NMEA_RD.bByte2     = BYTE#16#02 AND
                udtExample.udtIL_NMEA_RD.bByte3     = BYTE#16#03 AND
                udtExample.udtIL_NMEA_RD.bByte4     = BYTE#16#04 AND
                udtExample.udtIL_NMEA_RD.bByte5     = BYTE#16#05 AND
                udtExample.udtIL_NMEA_RD.bByte6     = BYTE#16#06 AND
                udtExample.udtIL_NMEA_RD.bByte7      = BYTE#16#07 AND
                udtExample.udtIL_NMEA_RD.bByte8      = BYTE#16#08
            THEN
                (* The xNDR is conncted to a CTU that counts up to 100. This way, we can
                be sure our message is received constantly *)
                IF CTU_NDR.CV    = 100 THEN
                    xResetCTU           := TRUE;
                    udtExample.iState   := 50;
                END_IF;
            END_IF;

        50: (* Reset everything *)
            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
            udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;
            udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;

            udtExample.udtIL_NMEA_RD.xActivate  := FALSE;
            udtExample.udtIL_NMEA_RD.dwCAN_ID   := DWORD#16#00000000;
            udtExample.udtIL_NMEA_RD.tControl   := t#0s;

            udtExample.udtCanData.arrMessagesReceive[0].xUsed            := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].diID             := DINT#0;
            udtExample.udtCanData.arrMessagesReceive[0].iDLC             := 0;
            udtExample.udtCanData.arrMessagesReceive[0].udiSequence      := UDINT#0;
            udtExample.udtCanData.arrMessagesReceive[0].xFrameFormat     := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].xFrameType       := FALSE;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[1]       := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[2]       := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[3]       := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[4]       := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[5]       := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[6]       := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[7]       := BYTE#16#00;
            udtExample.udtCanData.arrMessagesReceive[0].arrData[8]       := BYTE#16#00;

            xResetCTU   := FALSE;

            strStatus   := 'Stop sending the message with ID 16#18FC31F8.';

            udtExample.iState   := 60;

        60: (* Wait for manual trigger *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                udtExample.iState   := 70;
            END_IF;

        70: (* Check if the function blocks are deactivated *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = FALSE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#0000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0000 AND
                udtExample.udtIL_NMEA_RD.xReady         = FALSE AND
                udtExample.udtIL_NMEA_RD.xError         = FALSE AND
                udtExample.udtIL_NMEA_RD.wDiagCode      = WORD#16#0000
            THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 32000;

    END_CASE;
```

```
END_IF;
```

14.5.2.3 State machine: E_2000

```
IF udtExample.iExample  = 2000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example ';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate the AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;
            udtExample.udtAXL_CAN_COMM.xSend        := TRUE;

            udtExample.iState   := 10;

        10: (* Wait until AXL_CAN_COMM is active *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = TRUE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0001
            THEN
                (* Now we prepare an example NMEA message that we want to send *)
                udtExample.udtIL_NMEA_WR.dwCAN_ID       := DWORD#16#18FC13F9;
                udtExample.udtIL_NMEA_WR.iDataLength    := 8;
                (* For this example, the data are just random *)
                udtExample.udtIL_NMEA_WR.bByte1         := BYTE#16#10;
                udtExample.udtIL_NMEA_WR.bByte2         := BYTE#16#20;
                udtExample.udtIL_NMEA_WR.bByte3         := BYTE#16#30;
                udtExample.udtIL_NMEA_WR.bByte4         := BYTE#16#40;
                udtExample.udtIL_NMEA_WR.bByte5         := BYTE#16#50;
                udtExample.udtIL_NMEA_WR.bByte6         := BYTE#16#60;
                udtExample.udtIL_NMEA_WR.bByte7         := BYTE#16#70;
                udtExample.udtIL_NMEA_WR.bByte8         := BYTE#16#80;
                (* Here we set the cycle time of the message *)
                udtExample.udtIL_NMEA_WR.tInterval      := t#1s;
                udtExample.udtIL_NMEA_WR.xActivate      := TRUE;
                udtExample.iState   := 20;
            END_IF;

        20: (* Wait until IL_NMEA_WR is active *)
            IF
                udtExample.udtIL_NMEA_WR.xReady     = TRUE AND
                udtExample.udtIL_NMEA_WR.xError     = FALSE AND
                udtExample.udtIL_NMEA_WR.wDiagCode  = WORD#16#8000
            THEN
                (* The xDone output, that shows that the message was send, is connected
                to a CTU. That way, we wait until we send the message 100 times. *)
                IF CTU_DONE.CV  = 100 THEN
                    udtExample.udtIL_NMEA_WR.xActivate  := FALSE;
                    xResetCTU                           := TRUE;

                    udtExample.iState   := 30;
                END_IF;
            END_IF;

        30: (* Reset everything *)
            xResetCTU   := FALSE;

            udtExample.udtIL_NMEA_WR.dwCAN_ID       := DWORD#16#00000000;
            udtExample.udtIL_NMEA_WR.iDataLength    := 0;
            udtExample.udtIL_NMEA_WR.bByte1         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.bByte2         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.bByte3         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.bByte4         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.bByte5         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.bByte6         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.bByte7         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.bByte8         := BYTE#16#00;
            udtExample.udtIL_NMEA_WR.tInterval      := t#0s;

            udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
            udtExample.udtAXL_CAN_COMM.xSend        := FALSE;

            udtExample.udtCanData.arrMessagesSend[0].xUsed          := FALSE;
            udtExample.udtCanData.arrMessagesSend[0].diID           := DINT#0;
            udtExample.udtCanData.arrMessagesSend[0].iDLC           := 0;
            udtExample.udtCanData.arrMessagesSend[0].udiSequence    := UDINT#0;
            udtExample.udtCanData.arrMessagesSend[0].xFrameFormat   := FALSE;
            udtExample.udtCanData.arrMessagesSend[0].xFrameType     := FALSE;
            udtExample.udtCanData.arrMessagesSend[0].arrData[1]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[2]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[3]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[4]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[5]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[6]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[7]     := BYTE#16#00;
            udtExample.udtCanData.arrMessagesSend[0].arrData[8]     := BYTE#16#00;

            udtExample.iState   := 40;

        40: (* Check if the function blocks are deactivated *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = FALSE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#0000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0000 AND
                udtExample.udtIL_NMEA_WR.xReady         = FALSE AND
                udtExample.udtIL_NMEA_WR.xError         = FALSE AND
                udtExample.udtIL_NMEA_WR.wDiagCode      = WORD#16#0000
            THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 32000;

    END_CASE;

END_IF;
```

14.5.2.4 State machine: E_3000

```
IF udtExample.iExample  = 3000 THEN

    CASE udtExample.iState OF

        0: (* Init *)
            strStatus   := 'Execute Example ';
            strStatus   := CONCAT(strStatus, TO_STRING(udtExample.iExample,'{0:d}'));

            (* Activate the AXL_CAN_COMM *)
            udtExample.udtAXL_CAN_COMM.xActivate    := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceiveMode := TRUE;
            udtExample.udtAXL_CAN_COMM.xReceive     := TRUE;

            udtExample.iState   := 10;

        10: (* Wait ultil AXL_CAN_COMM is active *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive      = TRUE AND
                udtExample.udtAXL_CAN_COMM.xError       = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode    = WORD#16#8000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode = WORD#16#0002
            THEN
                (* For this example, we prepared a NMEA TPCM BAM message that is send
                in a 500ms cycle. Also we prepared 3 TPDT messages. The TCPM message has the
                ID 16#04ECFF02 and the TPDT message 16#04EBFF02. *)
                strStatus   := 'Start sending the TPCM.BAM message.';

                udtExample.iState   := 20;
            END_IF;

        20: (* Wait for manual trigger *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                (* After the TCPM message is sending, we can activate the function block *)
                strStatus                               := 'Execute E_3000';
                udtExample.udtIL_NMEA_RD_MULTI.xActivate    := TRUE;
                udtExample.udtIL_NMEA_RD_MULTI.dwCAN_ID     := DWORD#16#04ECFF02;
                (* Here we select which of the packeges we want to show at the outputs *)
                udtExample.udtIL_NMEA_RD_MULTI.bPackage     := BYTE#16#01;
                udtExample.udtIL_NMEA_RD_MULTI.tControl     := t#600ms;

                udtExample.iState   := 30;
            END_IF;

        30: (* Wait ultil IL_NMEA_RD_MULTI is ready *)
            IF
                udtExample.udtIL_NMEA_RD_MULTI.xReady       = TRUE AND
                udtExample.udtIL_NMEA_RD_MULTI.xError       = FALSE AND
                udtExample.udtIL_NMEA_RD_MULTI.wDiagCode    = WORD#16#8000
            THEN
                (* Now we can send the first TPDT message. It has the ID 16#04EBFF02
                and 8 data bytes of 16#01, 16#F1, 16#F1, 16#F1, 16#F1, 16#F1, 16#F1
                and 16#F1. *)
                strStatus           := 'Send the first TPDT message once.';
                udtExample.iState   := 40;
            END_IF;

        40: (* Wait for manual trigger and check if the message was received *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                strStatus   := 'Execute E_3000.';
                IF
                    udtExample.udtIL_NMEA_RD_MULTI.xReady       = TRUE AND
                    udtExample.udtIL_NMEA_RD_MULTI.xError       = FALSE AND
                    udtExample.udtIL_NMEA_RD_MULTI.wDiagCode    = WORD#16#8000 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bPackageMAX  = BYTE#16#03 AND
                    (* Package number *)
                    udtExample.udtIL_NMEA_RD_MULTI.bByte1       = BYTE#16#01 AND
                    (* Data bytes 1 - 7 *)
                    udtExample.udtIL_NMEA_RD_MULTI.bByte2       = BYTE#16#F1 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte3       = BYTE#16#F1 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte4       = BYTE#16#F1 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte5       = BYTE#16#F1 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte6       = BYTE#16#F1 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte7       = BYTE#16#F1 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte8       = BYTE#16#F1
                THEN
                    udtExample.iState   := 50;
                END_IF;
            END_IF;

        50: (* Send second TPDT message *)
            (* Now we can select that we want to see the second package and send
            the second TPDT message.
            It has the ID 16#04EBFF02 and 8 data bytes of 16#02, 16#F2, 16#F2,
            16#F2, 16#F2, 16#F2, 16#F2 and 16#F2. *)
            udtExample.udtIL_NMEA_RD_MULTI.bPackage := BYTE#16#02;

            strStatus   := 'Send the second TPDT message once.';

            udtExample.iState   := 60;

        60: (* Wait for manual trigger and check if the message was received *)
            IF R_TRIG_CONTINUE.Q    = TRUE THEN
                strStatus   := 'Execute E_3000.';
                IF
                    udtExample.udtIL_NMEA_RD_MULTI.xReady       = TRUE AND
                    udtExample.udtIL_NMEA_RD_MULTI.xError       = FALSE AND
                    udtExample.udtIL_NMEA_RD_MULTI.wDiagCode    = WORD#16#8000 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bPackageMAX  = BYTE#16#03 AND
                    (* Package number *)
                    udtExample.udtIL_NMEA_RD_MULTI.bByte1       = BYTE#16#02 AND
                    (* Data bytes 1 - 7 *)
                    udtExample.udtIL_NMEA_RD_MULTI.bByte2       = BYTE#16#F2 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte3       = BYTE#16#F2 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte4       = BYTE#16#F2 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte5       = BYTE#16#F2 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte6       = BYTE#16#F2 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte7       = BYTE#16#F2 AND
                    udtExample.udtIL_NMEA_RD_MULTI.bByte8       = BYTE#16#F2
                THEN
                    (* Example finished, reset everything *)
                    udtExample.udtIL_NMEA_RD_MULTI.xActivate    := FALSE;

                    strStatus   := 'Stop sending the TPCM.BAM message.';

                    udtExample.iState   := 70;
                END_IF;
            END_IF;

        70: (* Wait for manual trigger *)
            IF R_TRIG_CONTINUE.Q = TRUE THEN
                strStatus                               := 'Execute E_3000.';
                (* Reset everything *)
                udtExample.udtAXL_CAN_COMM.xActivate    := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceiveMode := FALSE;
                udtExample.udtAXL_CAN_COMM.xReceive     := FALSE;
```

```
                udtExample.udtIL_NMEA_RD_MULTI.dwCAN_ID := DWORD#16#00000000;
                udtExample.udtIL_NMEA_RD_MULTI.bPackage := BYTE#16#00;
                udtExample.udtIL_NMEA_RD_MULTI.tControl := t#0s;

                udtExample.udtCanData.arrMessagesReceive[0].xUsed           := FALSE;
                udtExample.udtCanData.arrMessagesReceive[0].diID            := DINT#0;
                udtExample.udtCanData.arrMessagesReceive[0].iDLC            := 0;
                udtExample.udtCanData.arrMessagesReceive[0].udiSequence     := UDINT#0;
                udtExample.udtCanData.arrMessagesReceive[0].xFrameFormat    := FALSE;
                udtExample.udtCanData.arrMessagesReceive[0].xFrameType      := FALSE;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[1]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[2]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[3]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[4]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[5]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[6]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[7]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[0].arrData[8]      := BYTE#16#00;

                udtExample.udtCanData.arrMessagesReceive[1].xUsed           := FALSE;
                udtExample.udtCanData.arrMessagesReceive[1].diID            := DINT#0;
                udtExample.udtCanData.arrMessagesReceive[1].iDLC            := 0;
                udtExample.udtCanData.arrMessagesReceive[1].udiSequence     := UDINT#0;
                udtExample.udtCanData.arrMessagesReceive[1].xFrameFormat    := FALSE;
                udtExample.udtCanData.arrMessagesReceive[1].xFrameType      := FALSE;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[1]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[2]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[3]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[4]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[5]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[6]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[7]      := BYTE#16#00;
                udtExample.udtCanData.arrMessagesReceive[1].arrData[8]      := BYTE#16#00;

                udtExample.iState   := 80;
            END_IF;

        80: (* Check if the function blocks are deactivated *)
            IF
                udtExample.udtAXL_CAN_COMM.xActive          = FALSE AND
                udtExample.udtAXL_CAN_COMM.xError           = FALSE AND
                udtExample.udtAXL_CAN_COMM.wDiagCode        = WORD#16#0000 AND
                udtExample.udtAXL_CAN_COMM.wAddDiagCode     = WORD#16#0000 AND
                udtExample.udtIL_NMEA_RD_MULTI.xReady       = FALSE AND
                udtExample.udtIL_NMEA_RD_MULTI.xError       = FALSE AND
                udtExample.udtIL_NMEA_RD_MULTI.wDiagCode    = WORD#16#0000
            THEN
                udtExample.iState   := 1000;
            END_IF;

        1000: (* Successful finished *)
            udtExample.iState   := 0;
            udtExample.iExample := 32000;

    END_CASE;

END_IF;
```

15 Appendix

15.1 Diag codes of used firmware function blocks

15.1.1 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.

15.1.2 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.

15.1.3 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).

15.1.4 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).

15.2 Data types

```
TYPE

    AXL_CAN_UDT_DIAGSTATE : STRUCT
        uiErrorNumber   : UINT; (* Error number *)
        usiPrio         : USINT; (* Priority:
                                     0x0 = No error;
                                     0x1 = Error;
                                     0x2 = Warning;
                                     0x81 = Quitted error;
                                     0x82 = Quitted warning; *)
        usiChannel      : USINT; (* Channel/Group/Module
                                     0x0 = No error;
                                     0xFF = Entire module *)
        uiErrorCode     : UINT; (* Error code *)
        usiAddInfo      : USINT; (* Additional information *)
        strErrorMessage : STRING; (* Error message *)
    END_STRUCT;

    CAN_UDT_11BitFilter : STRUCT
        uiFrom   : UINT; (*Start ID in range*)
        uiTo     : UINT; (*End ID in range*)
    END_STRUCT;

    CAN_ARR_11BitFilterRange : ARRAY [0..59] OF CAN_UDT_11BitFilter;

    CAN_ARR_B_0_63  : ARRAY [0..63] OF BYTE;
    CAN_ARR_B_1_8   : ARRAY [1..8] OF BYTE;

    CAN_UDT_MESSAGE : STRUCT
        xUsed           : BOOL; (* Indicates if a message has already
                                     been processed*)
        diID            : DINT; (* ID of the CAN-Message *)
        iDLC            : INT; (* Data length *)
        udiSequence     : UDINT; (* How often a message with the same ID
                                     was received (Receive-Mode 1) *)
        xFrameFormat    : BOOL; (* True = 29 Bit ID, False = 11 Bit ID *)
        xFrameType      : BOOL; (* True = RTR *)
        arrData         : CAN_ARR_B_1_8; (* Data bytes *)
    END_STRUCT;

    CAN_ARR_MESSAGES_0_199      : ARRAY [0..199] OF CAN_UDT_MESSAGE;
    CAN_ARR_MESSAGES_0_3        : ARRAY [0..3] OF CAN_UDT_MESSAGE;
    CAN_ARR_BOX_MESSAGES_0_19   : ARRAY [0..19] OF CAN_UDT_MESSAGE;

    CAN_UDT_LAST_MESSAGES : STRUCT
        udtLastSendMessage      : CAN_UDT_MESSAGE;
        udtLastSendHighMessage  : CAN_UDT_MESSAGE;
        udtLastReceiveMessage   : CAN_UDT_MESSAGE;
    END_STRUCT;

    CAN_UDT_DATA : STRUCT
        xTx                 : BOOL; (* Toggles receive and send-mode *)
        xActivate           : BOOL; (* Indicates if AXL_CAN_COMM is activated *)
        xActive             : BOOL; (* Indicates if AXL_CAN_COMM is active *)
        xSend               : BOOL; (* Indicates if xSend for AXL_CAN_COMM is set *)
        (* Array of messages to send *)
        arrMessagesSend     : CAN_ARR_MESSAGES_0_199;
        (* Array of received messages *)
        arrMessagesReceive  : CAN_ARR_MESSAGES_0_199;
        (* Array of messages to send with high prio *)
        arrMessagesSendHigh : CAN_ARR_MESSAGES_0_3;
        (* Struct with the last send /received messages *)
        udtLastMessages     : CAN_UDT_LAST_MESSAGES;
    END_STRUCT;

    CAN_UDT_29BitFilter : STRUCT
        udiFrom : UDINT;
        udiTo : UDINT;
    END_STRUCT;

    CAN_ARR_29BitFilterRange : ARRAY [0..29] OF CAN_UDT_29BitFilter;

        CN_ARR_B_1_128      : ARRAY [1..128] OF BYTE; (* Array for Supi-Data *)
    CN_ARR_B_1_8        : ARRAY [1..8] OF BYTE; (* Array for CAN Message *)
    CN_ARR_SUPI_1_64    : ARRAY [1..64] OF BYTE; (* Array for single Supi *)

    (* Structure for process data *)
    CN_udt_SUPI_1_2 : STRUCT
        Supi1   : CN_ARR_SUPI_1_64;
        Supi2   : CN_ARR_SUPI_1_64;
    END_STRUCT;

    CN_udt_String : STRUCT
        xSendMesseage       : BOOL;
        xActivate           : BOOL;
        xReady              : BOOL;
        xUsed               : BOOL;
        xActive             : BOOL;
        iLenOfMsgArray      : INT;
        iLenOfMsgHighArray  : INT;
        iCC_OVR             : INT; (* Number of reported Overruns *)
    END_STRUCT;

    (* Struct for FIFO of TX messages (PLC to CAN) *)
    CN_udt_RxCanMessage : STRUCT
        xUsed           : BOOL; (* Receive(Tx) -> new message |
                                    Transmit(Rx) -> new message to send *)
        diID            : DINT; (* CAN-ID *)
        iDLC            : INT; (* Message length ( Tx incl. RTR and
                                    ID |  Rx only data length) *)
        udiSequence     : UDINT; (* Counter of received message with this ID *)
        usiFrameFormat  : USINT; (* 0 = default    1 = xtended *)
        usiFrameType    : USINT; (* 0 = D (Data) 1 = R (RTR) *)
        arrData         : CN_ARR_B_1_8;
    END_STRUCT;

    (* Additionally the last send / received message is playced in the
    element 0 of each array *)
    CN_ARR_RxCanMessage     : ARRAY [0..200] OF CN_udt_RxCanMessage;
    CN_ARR_RxHighCanMessage : ARRAY [0..4] OF CN_udt_RxCanMessage;

    (* Struct for FIFO of TX messages (CAN to PLC) *)
    CN_udt_TxCanMessage : STRUCT
        xUsed           : BOOL; (* Receive(Tx) -> new message |
                                    Transmit(Rx) -> new message to send *)
        diID            : DINT; (* CAN-ID *)
        iDLC            : INT; (* Message length ( Tx incl. RTR and
                                    ID |  Rx only data length) *)
        udiSequence     : UDINT; (* Counter of received message with this ID *)
        usiFrameFormat  : USINT; (* 0 = default    1 = xtended *)
        usiFrameType    : USINT; (* 0 = D (Data) 1 = R (RTR) *)
        arrData         : CN_ARR_B_1_8;
    END_STRUCT;
```

```
    (* Additionally the last send / received message is playced in the
    element 0 of each array *)
    CN_ARR_TxCanMessage : ARRAY [0..200] OF CN_udt_TxCanMessage;

    (* FB interface struct *)
    CN_udt_RxTx : STRUCT
        Can          : CN_udt_String;
        CanRx        : CN_ARR_RxCanMessage;
        CanTx        : CN_ARR_TxCanMessage;
        CanRxHigh    : CN_ARR_RxHighCanMessage;
    END_STRUCT;

    (* Diag data *)
    CN_udt_Diagnostic_CanNet : STRUCT
        iUsedRxBuf          : INT; (* %-value *)
        iMaxUsedRxBuf       : INT; (* %-value *)

        iTxCycleTime        : INT; (* Roundtrip for module, only for
                                    confirmed messages *)
        iUsedTxBuf          : INT; (* %-value *)
        iMaxUsedTxBuf       : INT; (* %-value *)
        iRxCycleTimeCur     : INT; (* current roundtrip for module, only for
                                    confirmed messages *)
        iRxCycleTimeMax     : INT; (* max. roundtrip for module, only for
                                    confirmed messages *)
    END_STRUCT;

    CN_arrByte_1_8   : ARRAY [1..8] OF BYTE;
    CN_arrWord_1_24  : ARRAY [1..24] OF WORD;
    CN_arrReal_1_24  : ARRAY [1..24] OF REAL;
    CN_arrReal_1_2   : ARRAY [1..2] OF REAL;

    CN_String20      : String (20);

    udtName_1_16 : ARRAY[1..16] OF CN_String20;
    (* Struct for REFUgak *)
    CN_udtREFUgak : STRUCT

        xInit           : BOOL; (* FALSE = read the device information *)
        xNDR            : BOOL; (* New data receive *)
        udiSNR          : UDINT; (* Serialnumber Index 0x1018,4 *)
        sHW             : CN_String20; (* Hardwareversion Index 0x1009,0 *)
        sFW             : CN_String20; (* Firmwareversion Index 0x100A,0 *)
        sName           : CN_String20; (* Device name Index 0x1008,0x00 *)
        bError          : BYTE; (* Error register 0x1001 *)
        wStatus         : WORD; (* Index 0x1002 *)
        iNbrDevice      : INT; (* Number of virtual devices *)
        dwString        : DWORD; (* Strings are configured/avaible *)
        rTemperature    : CN_arrReal_1_2; (* Index 0x6190,1 & Index 0x6190,2 *)
        arrCurrent      : CN_arrReal_1_24; (* Current SubIndex 0x6175,1 to 0x6184,
                                            16 and 0x6175,2 to 0x617C,2 *)
        arrVoltage      : CN_arrReal_1_24; (* Voltage SubIndex 0x6165,1 to 0x6174,
                                            16 and 0x6165,2 to 0x616C,2 *)
        arrDiagnostic   : CN_arrWord_1_24; (* Diagnostic SubIndex 0x6145,1 to 0x6154,
                                            16 and 0x6145,2 to 0x614C,2 *)
    END_STRUCT;

    PSDI             : ARRAY [1..8] OF BYTE;
    LPSDO            : ARRAY [1..32] OF BYTE;
    CN_ARR_B_1_5     : ARRAY [1..5] OF BYTE;
END_TYPE
```

16 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