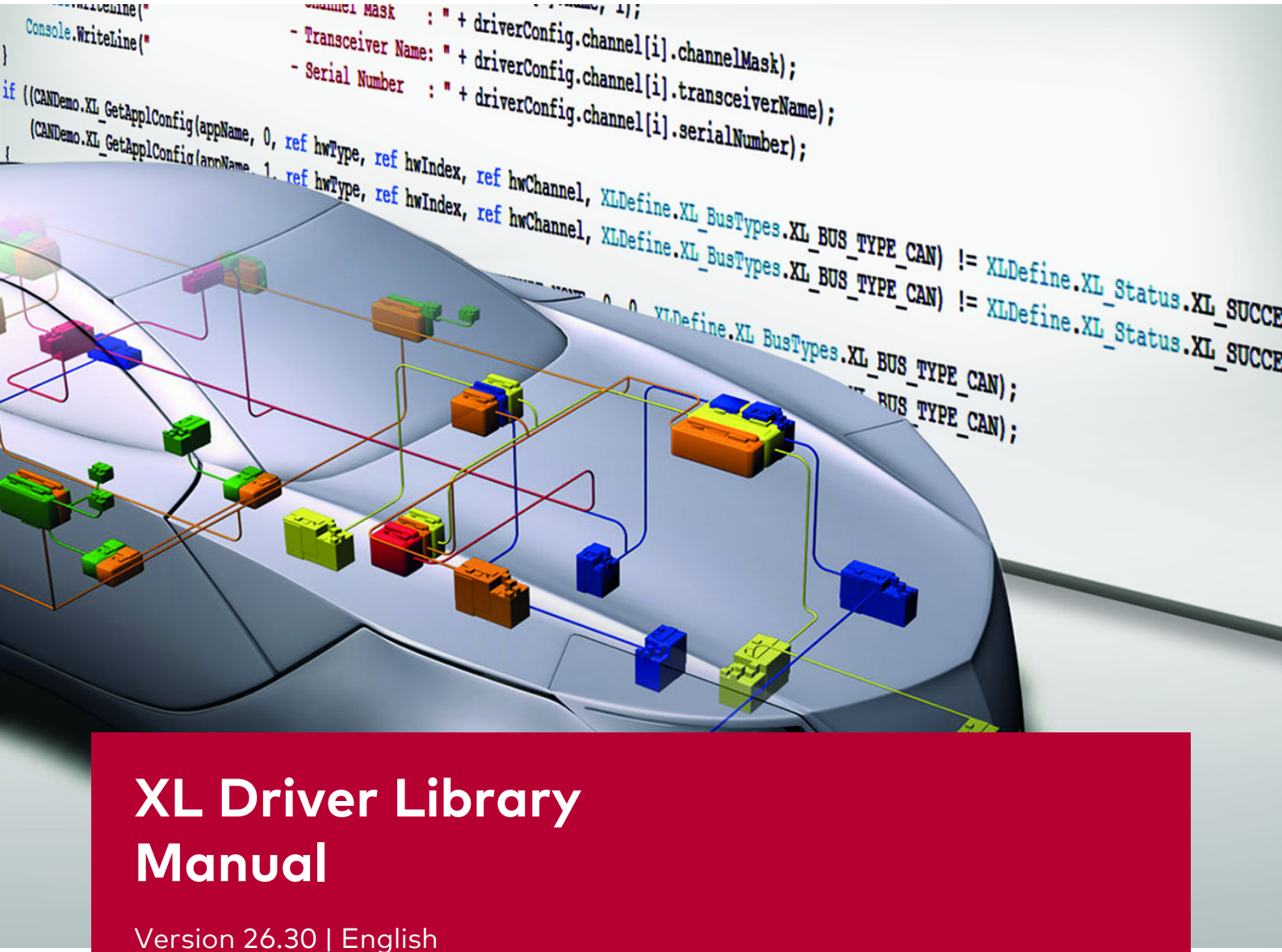
VECTOR



XL Driver Library Manual

Version 26.30 | English

Imprint

Vector Informatik GmbH
Ingersheimer Straße 24
D-70499 Stuttgart

The information and data given in this user manual can be changed without prior notice. No part of this manual may be reproduced in any form or by any means without the written permission of the publisher, regardless of which method or which instruments, electronic or mechanical, are used. All technical information, drafts, etc. are liable to law of copyright protection.

© Copyright 2026, Vector Informatik GmbH. All rights reserved.

Contents

1 Introduction **14**
1.1 About this User Manual 15
1.1.1 Warranty 16
1.1.2 Registered Trademarks 16
1.2 Important Notes 17

2 Overview **18**
2.1 General Information 19
2.2 Principles of the XL Driver Library 20
2.2.1 General Information 20
2.2.2 Step 1: Driver Initialization 21
2.2.3 Step 2: Channel Setup 23
2.2.4 Step 3: On Bus/Measurement Tasks 23
2.3 Driver Files and Examples 24
2.4 System Requirements 25
2.5 Additional Information 26
2.6 License Management 27
2.6.1 General Information 27

3 Common Commands **28**
3.1 Introduction 29
3.2 Support for More Than 64 Channels 30
3.2.1 Flowchart for up to 64 Channels 30
3.2.2 Flowchart for More Than 64 Channels 30
3.3 Functions 31
3.3.1 xlOpenDriver 31
3.3.2 xlCloseDriver 31
3.3.3 xlGetApplConfig 31
3.3.4 xlSetApplConfig 32
3.3.5 xlGetDriverConfig 33
3.3.6 xlGetRemoteDriverConfig 33
3.3.7 xlGetChannelIndex 34
3.3.8 xlGetChannelMask 35
3.3.9 xlCreatePort 36
3.3.10 xlAddChannelToPort 37
3.3.11 xlFinalizePort 38
3.3.12 xlOpenPort 38
3.3.13 xlClosePort 40

3.3.14 xlSetTimerRate 40
3.3.15 xlSetTimerRateAndChannel 41
3.3.16 xlResetClock 41
3.3.17 xlSetNotification 42
3.3.18 xlFlushReceiveQueue 43
3.3.19 xlGetReceiveQueueLevel 43
3.3.20 xlActivateChannel 44
3.3.21 xlReceive 45
3.3.22 xlGetEventString 46
3.3.23 xlGetErrorString 47
3.3.24 xlGetSyncTime 47
3.3.25 xlGetChannelTime 48
3.3.26 xlGenerateSyncPulse 49
3.3.27 xlPopupHwConfig 49
3.3.28 xlDeactivateChannel 49
3.3.29 xlGetLicenseInfo 50
3.3.30 xlSetGlobalTimeSync 50
3.3.31 xlGetKeymanBoxes 51
3.3.32 xlGetKeymanInfo 51
3.3.33 xlCreateDriverConfig 52
3.3.34 xlDestroyDriverConfig 54
3.4 Structs 55
3.4.1 XLdriverConfig 55
3.4.2 XLchannelConfig 55
3.4.3 XLbusParams 58
3.4.4 XLlicenseInfo 62
3.4.5 XLapiIDriverConfigV1 63
3.4.6 XLchannelDrvConfigV1 63
3.4.7 XLdeviceDrvConfigV1 65
3.4.8 XLnetworkDrvConfigV1 67
3.4.9 XLswitchDrvConfigV1 67
3.4.10 XLvirtualportDrvConfigV1 68
3.4.11 XLmeasurementpointDrvConfigV1 69
3.4.12 XLdllDrvConfigV1 69
3.5 Events 70
3.5.1 XLevent 70
3.5.2 XL Tag Data 71
3.5.3 XL Sync Pulse 71
3.5.4 XL Transceiver 72
3.5.5 XL Timer 72

4 TimeSync Commands 73
4.1 Introduction 74
4.1.1 General Information 74
4.1.2 Time Sync Basics 74
4.1.3 Time Sync Example 75
4.2 Flowchart Time Sync Status (Channel Based) 77
4.3 Flowchart Time Sync Status (Network Based) 78
4.4 Flowchart Application Clock 79
4.5 Application Examples 80
4.5.1 xlVTssDemo 80
4.6 Functions 83
4.6.1 xlTsCreateClock 83
4.6.2 xlTsDestroyClock 83
4.6.3 xlTsGetDomainTime 83
4.6.4 xlTsResetClocks 84
4.6.5 xlNetTsResetClocks 85
4.6.6 xlTsSetNotification 85
4.6.7 xlNetTsSetNotification 85
4.6.8 xlTsGetStatus 86
4.6.9 xlNetTsGetStatus 87
4.7 Structs 88
4.7.1 XLtsClkUuid 88
4.7.2 XltsTimeScale 88
4.7.3 XLtsLeapSeconds 89

5 CAN Commands 90
5.1 Introduction 91
5.2 Flowchart 92
5.3 Functions 93
5.3.1 xlCanSetChannelMode 93
5.3.2 xlCanSetChannelOutput 93
5.3.3 xlCanSetReceiveMode 94
5.3.4 xlCanSetChannelTransceiver 94
5.3.5 xlCanSetChannelParams 96
5.3.6 xlCanSetChannelParamsC200 97
5.3.7 xlCanSetChannelBitrate 97
5.3.8 xlCanSetChannelAcceptance 98
5.3.9 xlCanAddAcceptanceRange 99
5.3.10 xlCanRemoveAcceptanceRange 100

5.3.11 xlCanResetAcceptance 101
5.3.12 xlCanRequestChipState 101
5.3.13 xlCanTransmit 102
5.3.14 xlCanFlushTransmitQueue 103
5.4 Structs 104
5.4.1 XLchipParams 104
5.5 Events 105
5.5.1 XL CAN Message 105
5.5.2 XL Chip State 106
5.6 Application Examples 108
5.6.1 xlCANdemo 108
5.6.2 xlCANcontrol 110

6 CAN FD Commands 113

6.1 Introduction 114
6.2 Flowchart 115
6.3 Functions 116
6.3.1 xlCanFdSetConfiguration 116
6.3.2 xlCanTransmitEx 116
6.3.3 xlCanReceive 117
6.3.4 xlCanGetEventString 117
6.4 Structs 118
6.4.1 XLcanFdConf 118
6.5 Events 121
6.5.1 XLcanTxEvent 121
6.5.2 XL_CAN_TX_MSG 121
6.5.3 XLcanRxEvent 122
6.5.4 XL_CAN_EV_RX_MSG 123
6.5.5 XL_CAN_EV_ERROR 124
6.5.6 XL_CAN_EV_CHIP_STATE 125
6.5.7 XL_CAN_EV_TX_REQUEST 126
6.5.8 XL_SYNC_PULSE_EV 126

7 Ethernet Commands 128

7.1 Introduction 129
7.1.1 General Information 129
7.1.2 Network-Based API vs. Channel-Based API 130
7.1.3 Device Support 133
7.1.4 Switching Access Mode 134

7.2 Network Based Access Mode 136
7.2.1 Basic Concept 136
7.2.2 Definitions 137
7.2.3 General Information 140
7.2.4 Flowchart 144
7.2.5 Functions 145
7.2.6 Structs 152
7.2.7 Events 153
7.2.8 Application Examples 159
7.3 Channel Based Access Mode 163
7.3.1 Flowchart 164
7.3.2 Functions 165
7.3.3 Structs 170
7.3.4 Events 175
7.3.5 Application Examples 185

8 LIN Commands 188

8.1 Introduction 189
8.2 Flowchart 190
8.3 LIN Basics 191
8.4 Functions 192
8.4.1 xlLinSetChannelParams 192
8.4.2 xlLinSetDLC 192
8.4.3 xlLinSetChecksum 193
8.4.4 xlLinSetSlave 194
8.4.5 xlLinSwitchSlave 195
8.4.6 xlLinSendRequest 195
8.4.7 xlLinWakeUp 196
8.4.8 xlLinSetSleepMode 196
8.5 Structs 198
8.5.1 XLlinStatPar 198
8.6 Events 199
8.6.1 XL LIN Message API 199
8.6.2 XL LIN Message 199
8.6.3 XL LIN Error Message 200
8.6.4 XL LIN Sync Error 200
8.6.5 LIN No Answer 200
8.6.6 LIN Wake Up 200
8.6.7 LIN Sleep 201
8.6.8 LIN CRC Info 201

8.7 Application Examples ... 202
8.7.1 xlLINExample ... 202

9 K-Line Commands ... 204

9.1 Introduction ... 205
9.2 Flowchart ... 206
9.3 Functions ... 207
9.3.1 xlKlineFastInitTester ... 207
9.3.2 xlKlineInit5BdEcu ... 207
9.3.3 xlKlineInit5BdTester ... 208
9.3.4 xlKlineSetBaudrate ... 208
9.3.5 xlKlineSetCommunicationTimingEcu ... 208
9.3.6 xlKlineSetCommunicationTimingTester ... 209
9.3.7 xlKlineSetUartParams ... 209
9.3.8 xlKlineSwitchHighspeedMode ... 210
9.3.9 xlKlineSwitchTesterResistor ... 210
9.3.10 xlKlineTransmit ... 211
9.4 Structs ... 212
9.4.1 XLkline5BdEcu ... 212
9.4.2 XLkline5BdTester ... 213
9.4.3 XLklineInitTester ... 213
9.4.4 XLklineSetComEcu ... 214
9.4.5 XLklineSetComTester ... 214
9.4.6 XLklineUartParameter ... 215
9.5 Events ... 216
9.5.1 K-Line Data ... 216
9.5.2 K-Line Confirmation Event ... 216
9.5.3 K-Line Error Event ... 217
9.5.4 K-Line ECU 5Bd Error ... 217
9.5.5 K-Line Tester 5Bd Error ... 218
9.5.6 K-Line ibsErr Error ... 218
9.5.7 K-Line RXTX Error ... 219
9.5.8 K-Line RX Data ... 219
9.5.9 K-Line Tester 5Bd ... 220
9.5.10 K-Line ECU Fastinit WU Pattern ... 220
9.5.11 K-Line Tester Fastinit WU Pattern ... 220
9.5.12 K-Line TX Data ... 221

10 D/A IO Commands (IOcab) ... 222

10.1 Introduction ... 223

10.2 Flowchart ... 224
10.3 Functions ... 225
10.3.1 xlDAIOSetAnalogParameters ... 225
10.3.2 xlDAIOSetAnalogOutput ... 226
10.3.3 xlDAIOSetAnalogTrigger ... 227
10.3.4 xlDAIOSetDigitalParameters ... 227
10.3.5 xlDAIOSetDigitalOutput ... 228
10.3.6 xlDAIOSetPWMOutput ... 229
10.3.7 xlDAIOSetMeasurementFrequency ... 230
10.3.8 xlDAIORequestMeasurement ... 230
10.4 Events ... 231
10.4.1 XL DAIO Data ... 231
10.5 Application Examples ... 233
10.5.1 xlDAIOexample ... 233
10.5.2 xlDAIOdemo ... 235

11 D/A IO Commands (Piggybacks) ... 237
11.1 Introduction ... 238
11.2 Flowchart ... 239
11.3 Functions ... 240
11.3.1 xlIoSetTriggerMode (IOpiggy) ... 240
11.3.2 xlIoConfigurePorts ... 240
11.3.3 xlIoSetDigInThreshold ... 241
11.3.4 xlIoSetDigOutLevel ... 241
11.3.5 xlIoSetDigitalOutput ... 241
11.3.6 xlIoSetAnalogOutput ... 242
11.3.7 xlIoStartSampling ... 242
11.4 Structs ... 244
11.4.1 XLdaioTriggerMode ... 244
11.4.2 XLdaioSetPort ... 245
11.4.3 XLdaioDigitalParams (IOpiggy) ... 247
11.4.4 XLdaioAnalogParams ... 247
11.5 Events ... 249
11.5.1 XL DAIO Piggy Data ... 249
11.5.2 XL IO Analog Data ... 249
11.5.3 XL IO Digital Data ... 250

12 D/A IO Commands (Other) ... 251
12.1 Introduction ... 252

12.2 Flowchart ... 253
12.3 Functions ... 254
12.3.1 xlIoSetTriggerMode (VN1600) ... 254
12.3.2 xlIoSetDigitalOutput ... 254
12.4 Structs ... 255
12.4.1 XLdaioDigitalParams ... 255
12.5 Events ... 256
12.5.1 XL DAIO Piggy Data ... 256
12.5.2 XL IO Analog Data ... 256
12.5.3 XL IO Digital Data ... 257

13 FlexRay Commands ... 258
13.1 Introduction ... 259
13.2 Flowchart ... 260
13.3 Free Library and Advanced Library ... 261
13.4 FlexRay Basics ... 262
13.4.1 Introduction ... 262
13.4.2 Data Transmission Requirements ... 262
13.4.3 FlexRay Communication Architecture ... 263
13.4.4 Deterministic and Dynamic ... 265
13.4.5 CRC-Protected Data Transmission ... 267
13.5 Functions ... 268
13.5.1 xlFrSetConfiguration ... 268
13.5.2 xlFrGetChannelConfiguration ... 268
13.5.3 xlFrSetMode ... 268
13.5.4 xlFrInitStartupAndSync ... 269
13.5.5 xlFrSetupSymbolWindow ... 270
13.5.6 xlFrActivateSpy ... 270
13.5.7 xlSetTimerBasedNotify ... 271
13.5.8 xlFrReceive ... 271
13.5.9 xlFrTransmit ... 272
13.5.10 xlFrSetTransceiverMode ... 272
13.5.11 xlFrSendSymbolWindow ... 273
13.5.12 xlFrSetAcceptanceFilter ... 273
13.6 Structs ... 275
13.6.1 XLfrClusterConfig ... 275
13.6.2 XLfrChannelConfig ... 280
13.6.3 XLfrMode ... 280
13.6.4 XLfrAcceptanceFilter ... 282
13.7 Events ... 283

13.7.1 XLfrEvent ... 283
13.7.2 XL_FR_START_CYCLE_EV ... 284
13.7.3 XL_FR_RX_FRAME_EV ... 285
13.7.4 XL_FR_TX_FRAME_EV ... 287
13.7.5 XL_FR_TXACK_FRAME ... 288
13.7.6 XL_FR_INVALID_FRAME ... 288
13.7.7 XL_FR_WAKEUP_EV ... 288
13.7.8 XL_FR_SYMBOL_WINDOW_EV ... 289
13.7.9 XL_FR_ERROR_EV ... 290
13.7.10 XL_FR_ERROR_POC_MODE_EV ... 290
13.7.11 XL_FR_ERROR_SYNC_FRAMES_BELOWMIN ... 291
13.7.12 XL_FR_ERROR_SYNC_FRAMES_EV ... 291
13.7.13 XL_FR_ERROR_CLOCK_CORR_FAILURE_EV ... 291
13.7.14 XL_FR_ERROR_NIT_FAILURE_EV ... 292
13.7.15 XL_FR_ERROR_CC_ERROR_EV ... 292
13.7.16 XL_FR_STATUS_EV ... 293
13.7.17 XL_FR_NM_VECTOR_EV ... 294
13.7.18 XL_FR_SPY_FRAME_EV ... 295
13.7.19 XL_FR_SPY_SYMBOL_EV ... 295
13.7.20 XL_APPLICATION_NOTIFICATION_EV ... 296
13.8 Application Examples ... 297
13.8.1 xlFlexDemo ... 297
13.8.2 xlFlexDemoCmdLine ... 299
13.8.3 Fibex2CSharpReaderDemo ... 300

14 ARINC 429 Commands ... 301
14.1 Introduction ... 302
14.2 Flowchart ... 303
14.3 Functions ... 304
14.3.1 xlA429SetChannelParams ... 304
14.3.2 xlA429Transmit ... 305
14.3.3 xlA429Receive ... 307
14.4 Structs ... 309
14.4.1 XL_A429_PARAMS ... 309
14.4.2 XL_A429_MSG_TX ... 314
14.5 Events ... 317
14.5.1 XLa429Event ... 317
14.5.2 XL_A429_EV_TX_OK ... 318
14.5.3 XL_A429_EV_TAG_TX_ERR ... 319
14.5.4 XL_A429_EV_TAG_RX_OK ... 320

14.5.5 XL_A429_EV_TAG_RX_ERR 321
14.5.6 XL_A429_EV_BUS_STATISTIC 323
14.6 Application Examples 325
14.6.1 xlA429Control 325

15 .NET Wrapper 326
15.1 Introduction 327
15.2 Vector.XlApi 328
15.2.1 Overview 328
15.2.2 Design 329
15.2.3 Including the Wrapper in a Project 331
15.2.4 Relationship with vxlapi_NET 333
15.3 vxlapi_NET 334
15.3.1 Overview 334
15.3.2 XLDriver - Accessing Driver 336
15.3.3 XLClass - Storing Data/Parameters 337
15.3.4 XLDefine - Using Predefined Values 338
15.3.5 Including the Wrapper in a New .NET Project 339
15.4 Application Examples 341
15.4.1 xlCanDemo Vector.XlApi 341
15.4.2 xlCanFdDemo Vector.XlApi 342
15.4.3 xlLinDemo Vector.XlApi 343
15.4.4 xlLinDemo Single Vector.XlApi 344
15.4.5 xlIoPiggyDemo Vector.XlApi 345
15.4.6 xlNetEthDemo Vector.XlApi 347
15.4.7 xlTimeSyncMonitor .NET 348
15.4.8 xlCANdemo .NET 350
15.4.9 xlCANFDdemo .NET 351
15.4.10 xlLINDemo .NET 352
15.4.11 xlDAIOexample .NET 353
15.4.12 xlIOpiggyExample .NET 356
15.4.13 xlEthernetDemo .NET 358
15.4.14 xlFRdemo .NET 359

16 Error Codes 360
16.1 Common Error Codes 361
16.2 Time Sync Error Codes 363
16.3 CAN FD Error Codes 364
16.4 FlexRay Error Codes 365

16.5 Ethernet Error Codes 366

17 Important Notes - Details 367

17.1 Safety Instructions and Hazard Warnings 368
17.1.1 Proper Use and Intended Purpose 368
17.1.2 Hazards 369
17.2 Disclaimer 370
17.3 Disposal of Vector Hardware 371

18 Wichtige Hinweise 372

18.1 Sicherheits- und Gefahrenhinweise 373
18.1.1 Sach- und bestimmungsgemäßer Gebrauch 373
18.1.2 Gefahren 374
18.2 Haftungsausschluss 375
18.3 Entsorgung von Vector Hardware 376

1 Introduction

In this chapter you find the following information:

1.1 About this User Manual **15**
1.1.1 Warranty 16
1.1.2 Registered Trademarks 16
1.2 Important Notes **17**

1.1 About this User Manual

Conventions

In the two following charts you will find the conventions used in the user manual regarding utilized spellings and symbols.

Style	Utilization
bold	Blocks, surface elements, window and dialog names of the software. Accentuation of warnings and advices. **[OK]** Push buttons in brackets **File|Save** Notation for menus and menu entries
`Source Code`	File name and source code.
Hyperlink	Hyperlinks and references.
<CTRL>+<S>	Notation for shortcuts.

Symbol	Utilization
	This symbol calls your attention to warnings.
	Here you can obtain supplemental information.
	Here you can find additional information.
	Here is an example that has been prepared for you.
	Step-by-step instructions provide assistance at these points.
	Instructions on editing files are found at these points.
	This symbol warns you not to edit the specified file.

1.1.1 Warranty

Restriction of warranty

We reserve the right to change the contents of the documentation and the software without notice. Vector Informatik GmbH assumes no liability for correct contents or damages which are resulted from the usage of the documentation. We are grateful for references to mistakes or for suggestions for improvement to be able to offer you even more efficient products in the future.

1.1.2 Registered Trademarks

Registered trademarks

All trademarks mentioned in this documentation and if necessary third party registered are absolutely subject to the conditions of each valid label right and the rights of particular registered proprietor. All trademarks, trade names or company names are or can be trademarks or registered trademarks of their particular proprietors. All rights which are not expressly allowed are reserved. If an explicit label of trademarks, which are used in this documentation, fails, should not mean that a name is free of third party rights.

- Windows, Windows 7, Windows 8.1, Windows 10, Windows 11 are trademarks of the Microsoft Corporation.

1.2 Important Notes



WARNING!
We provide our important notes and safety instructions in several languages, including Englisch (EN) and German (DE). For more details, see the relevant section:

- EN: Important Notes - Details
- DE: Wichtige Hinweise

2 Overview

In this chapter you find the following information:

2.1 General Information **19**
2.2 Principles of the XL Driver Library **20**
2.3 Driver Files and Examples **24**
2.4 System Requirements **25**
2.5 Additional Information **26**
2.6 License Management **27**

2.1 General Information

The XL Driver Library

This document describes the **XL Driver Library** (XL API) which enables the development of own applications for CAN, CAN FD, LIN, Ethernet, FlexRay, digital/analog input/output (DAIO) or ARINC on supported Vector devices.

The XL API abstracts the underlying Vector devices so applications are independent of hardware and operating systems.

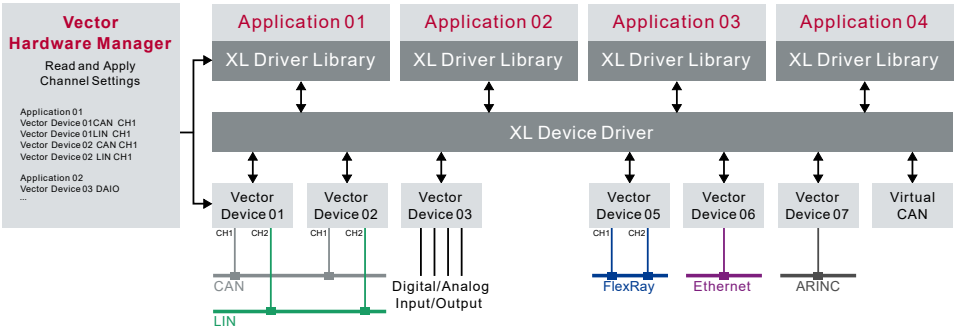


Figure 1: Example of applications using the XL Driver Library to access Vector devices

Vector Hardware Manager

The Vector Hardware Manager is required to configure the hardware and set up the physical channel assignment and is part of the Vector Driver Package, which can be downloaded from:
https://www.vector.com/latest-driver-setup-windows.

During the driver installation, the Vector Hardware Manager is automatically installed in the background. The management of the application settings can be either done in the tool or via get/set functions of the **XL Driver Library**. The applications can read the parameters at run time via a user defined application name. The provided XL API examples (e. g. `xlCANcontrol.exe`) create a new application name (if not already present) for channel assignments.

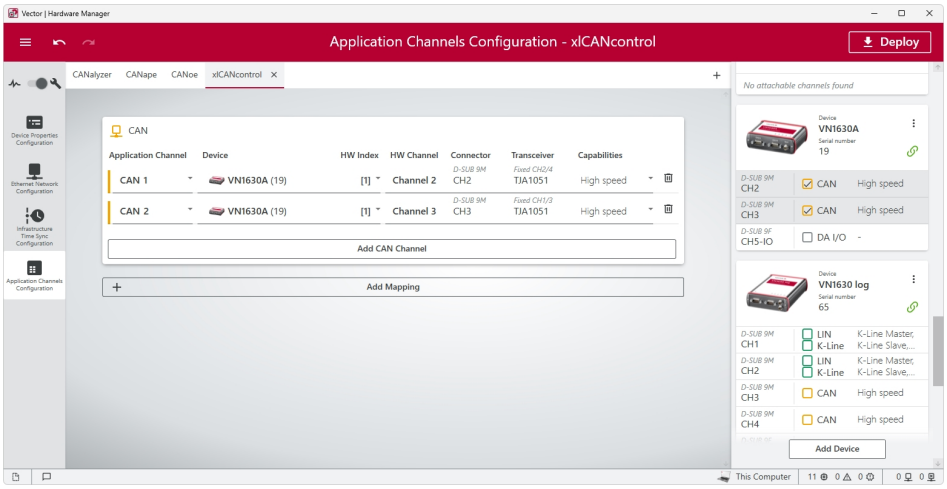


Figure 2: Example of hardware settings

Reference
Please refer to the user manual of your Vector device for detailed information on the hardware installation and to the Vector Hardware Manager help.

2.2 Principles of the XL Driver Library

2.2.1 General Information

Accessing Vector devices

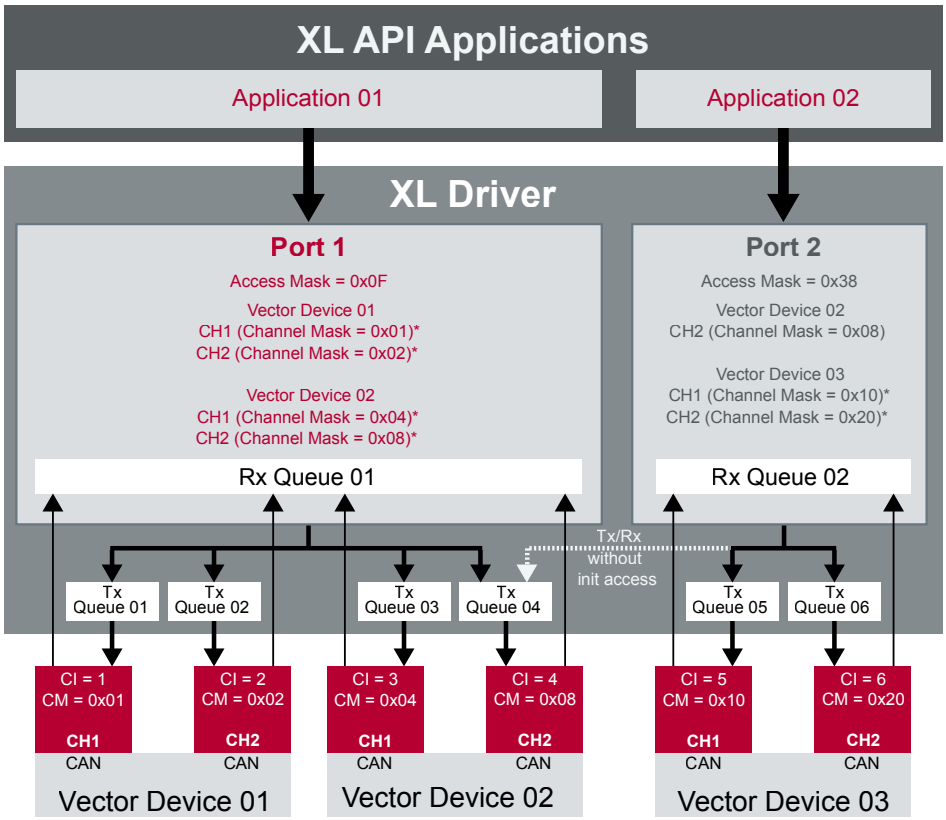
The usage of the **XL Driver Library** can be split into three major steps:

- **Step 1: Driver initialization**
  Initialization of a driver port with the selected channels of a certain bus type.
- **Step 2: Channel setup**
  Configuration of the opened port and its channels.
- **Step 3: On bus/measurement tasks**
  Definition of main tasks for Tx and Rx messages.

2.2.2 Step 1: Driver Initialization

Selecting device and channels

Before a message can be transmitted or received, you have to specify the required channels of one or more supported Vector devices. Though this is typically done via the Vector Hardware Manager, the following sections provide background information on indexing of hardware channels which is required in almost each function call.



*application has **init access** on that channel
CI: Channel Index
CM: Channel Mask

Figure 3: Principle structure of CAN applications

Channel index

The Vector device channels are identified by their **channel index** which is a global application specific value provided by the driver. The order of the channel indexes always depends on the installed and connected Vector devices.

Channel mask

To address one or more available channels, a so-called **channel mask** is required which is a **channel index** based bit mask. The rule is as follows:

```
channel mask = 1 << channel index
```

Note
The way how to determine a channel mask of a specific device channel will be explained later (see section xlOpenPort on page 38).

Port handle

Once the channel mask is passed over to the open port function, the **XL Driver Library** returns a specific **port handle** that is used for all subsequent function calls on those channels.

Access mask

To access individual channels of the opened port, a so-called **access mask** has to be passed to almost each XL API function call. The access mask is a bit mask derived from the channel mask. To refer to multiple channels, individual access masks can be combined, e. g.:

Device No.	Channel Index	Access Mask (bin)	Access Mask (hex)
01	1	0b0000000**1**	0x001
	2	0b0000001**0**	0x002
02	3	0b**01**000000	0x040
	4	0b**1**0000000	0x080
	1 + 4	**0b10000001**	**0x81**

Note
The selected channels have to be of the **same** bus type. Otherwise no valid port handle will be returned by the **XL Driver Library**.

Init access

The very first application port that accesses a certain channel gets the property **init access** for that channel. This property is assigned for each individual channel and enables the application to change its settings. **Init access** is granted to only one application port.

Multiple applications

In general, if a different application demands access on device channels, the **XL Driver Library** returns another port handle. Depending on the bus type, applications can access a specific channel at the same time without **init access** (e. g. CAN), but there are also bus types which have no or only a limited multi application support (e. g. LIN).

Reference
For further details on the multi application support please refer to the introductions in each bus section.

Note
An application can also open multiple ports (e. g. when using multiple bus types at the same time, e g. CAN and FlexRay).

2.2.3 Step 2: Channel Setup

Hardware initialization

The channels can be activated and are ready for operation.



Reference
For further information on the channel setup please refer to the flowchart at the beginning of the according bus section.

2.2.4 Step 3: On Bus/Measurement Tasks

Transmitting messages

After the driver has been initialized and the channels set up, the actual functionality is performed in the main task. Each physical channel is equipped with its own transmit queue. The transmit messages are added to the matching queue as selected by the access mask.

Receiving messages

The received messages are copied to the common **receive queue** of the according port. Messages stored in this queue can be read either by polling or via event driven notifications (`WaitForSingleObject`).

2.3 Driver Files and Examples

Driver Files

The following files are required to develop an **XL Driver Library** application.

File name	Description
`vxlapi.dll`	32 bit DLL for Windows 11
`vxlapi64.dll`	64 bit DLL for Windows 11
`vxlapi.h`	C header for C/C++ based applications
`vxlapi_NET.dll`	Wrapper for .NET bases applications (requires `vxlapi.dll`/`vxlapi64.dll`)
`vxlapi_NET.xml`	Wrapper documentation, used by IntelliSense function



Note
It is recommended to place all files in the folder of the application (.exe).



Note
It is not possible to initialize the **XL Driver Library** in a superior DLL within a DllMain function.

Examples

The **XL Driver Library** also contains a couple of examples (including the source code and already compiled projects) which show the handling for initialization, transmitting and receiving of messages.



Reference
Find the source code examples in sub folder `\samples`.
The according compiled examples can be found in sub folder `\exec`.



Note
The **XL Driver Library** can also be loaded dynamically. Please check the application example `xlCANcontrol` and the module `xlLoadlib.cpp` for further details.

2.4 System Requirements

Supported Vector devices

The **XL Driver Library** is compatible with the following Vector devices:

- VN0600 Interface Family
- VN1500 Interface Family
- VN1600 Interface Family
- VN5000 Interface Family
- VN7000 Interface Family
- VN8800 Interface Family
- VN8900 Interface Family
- VT6306(B)
- VX0312/VX1135/VX1161.41

Supported operating systems

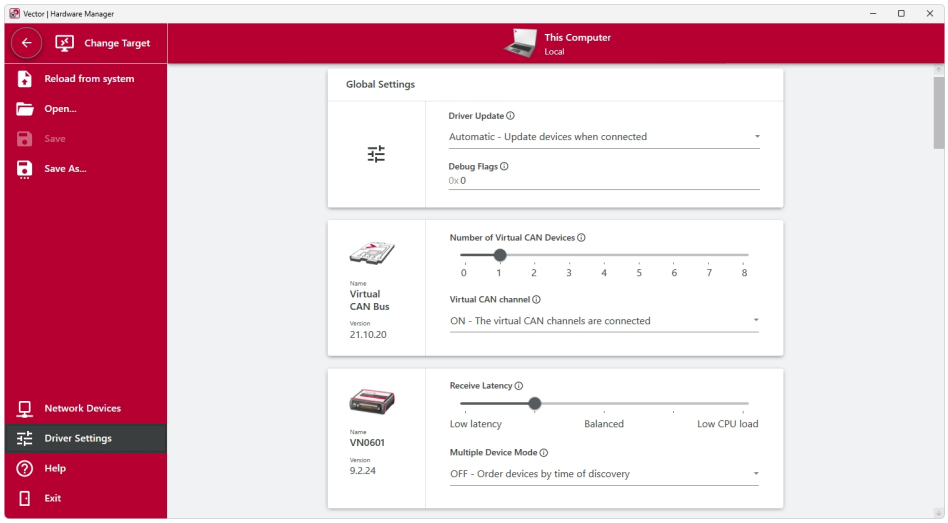
The **XL Driver Library** is compatible with the following operating systems:

- Windows 11 (64 bit)

2.5 Additional Information

Debug prints

The **XL Driver Library** supports debug prints which can be enabled in the Vector Hardware Manager (**Burger menu | Driver Settings | Global Settings | Debug Flags**).



Flags	Supported Bus Type
0x400000	CAN, LIN, DAIO
0x010000	FlexRay

Reference

The debug prints can be viewed with the freeware tool **DebugView** (download from Microsoft website: https://docs.microsoft.com/en-us/sysinternals/downloads/debugview).

2.6 License Management

2.6.1 General Information

Advanced libraries

While most features of the XL Driver Library are free, some features need a license to unlock. Currently, the only feature that needs a license is the FlexRay Advanced Library (see section Free Library and Advanced Library on page 261). A function that requires a license that has not been unlocked returns `XL_ERR_NO_LICENSE`.

License types

The following license types may be used to unlock a feature:

- **Old device licenses**
  Old device licenses are bound to a Vector network interface and listed in the old **Vector Hardware Config** tool. Those licenses automatically unlock features in the XL Driver Library.
- **Old Keyman licenses**
  Old Keyman licenses are bound to a Keyman hardware dongle and listed in the old **Vector Hardware Config** tool. To unlock a feature with an old Keyman license, first call `xlGetKeymanBoxes()` (see page 51) and subsequently call `xlGetKeymanInfo()` (see page 51).
- **New license model**
  Licenses of the new license model may be stored on a Vector network interface, a Keyman dongle or on the host PC. Those licenses are managed with the **Vector License Client**. All features licensed via the new model are unlocked by calling `xlGetKeymanBoxes()` (page 51).

3 Common Commands

In this chapter you find the following information:

3.1 Introduction **29**
3.2 Support for More Than 64 Channels **30**
3.3 Functions **31**
3.4 Structs **55**
3.5 Events **70**

3.1 Introduction

Description

The **XL Driver Library** offers bus independent functions which are required for driver initialization, for reading/writing hardware settings from/to the Vector Hardware Manager as well as to open or close ports (see section Principles of the XL Driver Library on page 20).



Reference
Please refer to the flowcharts at the beginning of each bus section to see which functions are required to set up the driver.

3.2 Support for More Than 64 Channels

Description

The XL API offers two options for initializing an application port. Depending on the option, more than 64 channels are supported.

- Via **ChannelMask** (classic option)
  The number of channels that can be used per port is limited to 64.
- Via **ChannelIndex**
  Use this option to initialize ports for more than 64 channels.

3.2.1 Flowchart for up to 64 Channels

Calling sequence

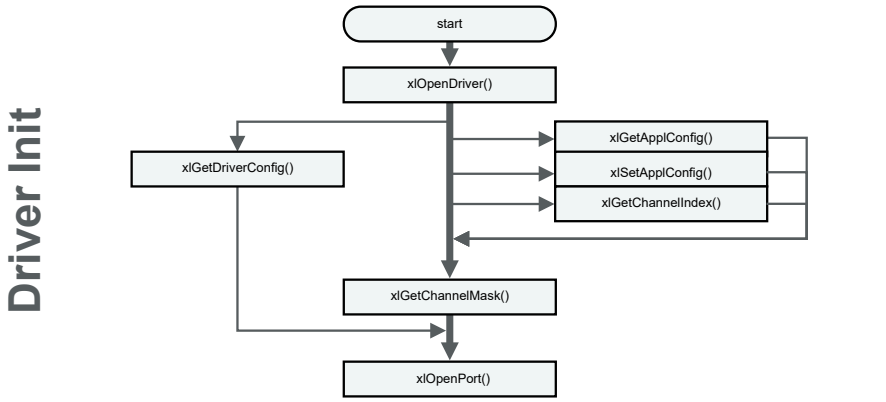


Figure 4: Port initialization via ChannelMask

3.2.2 Flowchart for More Than 64 Channels

Calling sequence

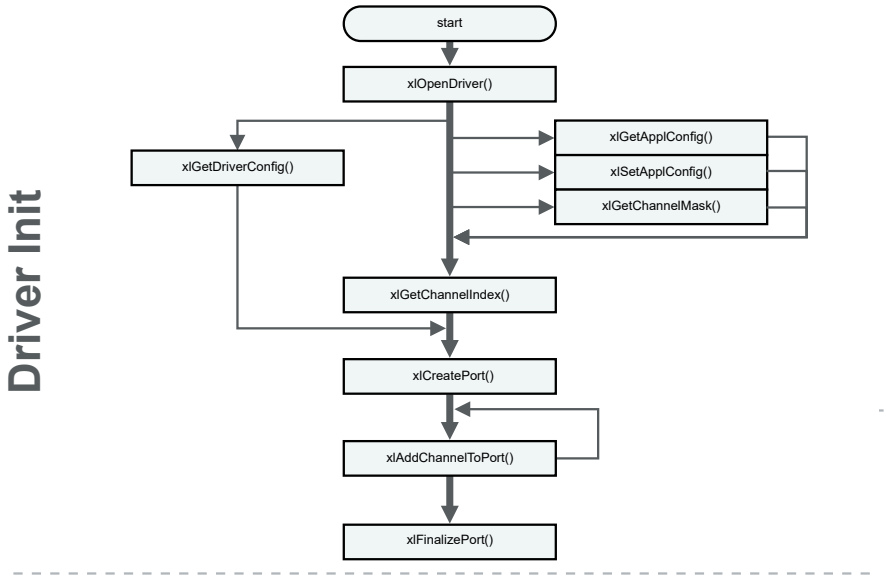


Figure 5: Port initialization via ChannelIndex

3.3 Functions

3.3.1 xlOpenDriver

Syntax

```
XLstatus xlOpenDriver(void)
```

Description

Each application must call this function to load the driver. If the function call is not successful (`XLStatus = 0`), no other API calls are possible.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.2 xlCloseDriver

Syntax

```
XLstatus xlCloseDriver(void)
```

Description

This function closes the driver.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.3 xlGetApplConfig

Syntax

```
XLstatus xlGetApplConfig(
  char          *appName
  unsigned int appChannel,
  unsigned int *pHwType,
  unsigned int *pHwIndex,
  unsigned int *pHwChannel,
  unsigned int busType)
```

Description

Retrieves the hardware settings for an application which are configured in the Vector Hardware Manager. The information can then be used to get the required channel mask (see section xlGetChannelMask on page 35). To open a port with multiple channels, the retrieved channel masks have to be combined before and then passed over to the open port function.

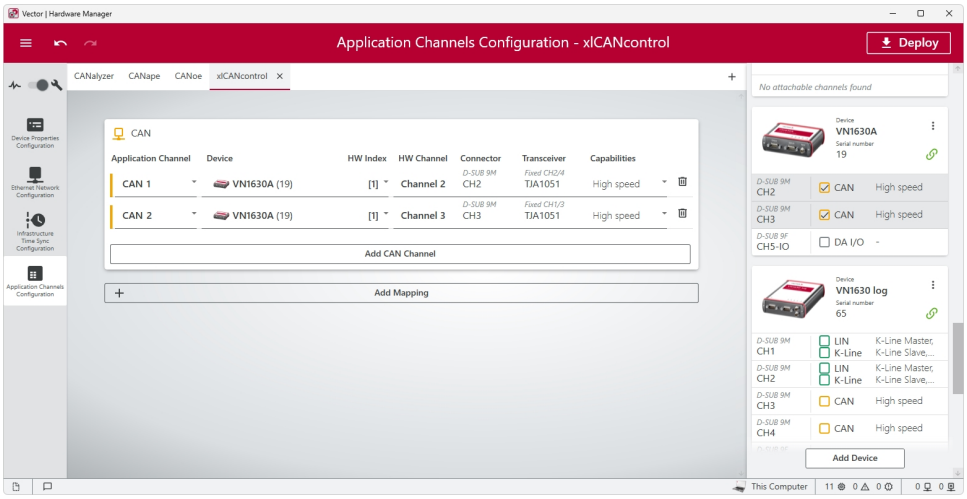


Figure 6: Example of hardware settings

Input parameters

- **appName**
  Name of the application to be read (e. g. "xlCANcontrol").
  Application names are listed in the Vector Hardware Manager.
- **appChannel**
  Selects the application channel (0,1, …). An application can offer several channels which are assigned to physical channels (e. g. “CANdemo CAN1” to VN1610 Channel 1 or “CANdemo CAN2” to VN1610 Channel 2). Such an assignment has to be configured with the Vector Hardware Manager.
- **busType**
  Specifies the bus type which is used by the application,
  e. g.:
  `XL_BUS_TYPE_CAN`
  `XL_BUS_TYPE_LIN`
  `XL_BUS_TYPE_DAIO`
  `XL_BUS_TYPE_FLEXRAY`

  Find further definitions in the `vxlapi.h` file.

Output parameters

- **pHwType**
  Hardware type is returned (see `vxlapi.h`),
  e. g. CANcardXL: `XL_HWTYPE_CANCARDXL`
- **pHwIndex**
  Index of same hardware types is returned (0,1, ...),
  e. g. for two CANcardXL on one system:
  - CANcardXL 01: `hwIndex = 0`
  - CANcardXL 02: `hwIndex = 1`
- **pHwChannel**
  Channel index of same hardware types is returned (0,1, ...),
  e. g. CANcardXL:
  Channel 1: `hwChannel = 0`
  Channel 2: `hwChannel = 1`

Return value

Returns an error code (see section Error Codes on page 360).

3.3.4 xlSetApplConfig

Syntax

```
XLstatus xlSetApplConfig(
  char         *appName,
  unsigned int appChannel,
  unsigned int hwType,
  unsigned int hwIndex,
  unsigned int hwChannel,
  unsigned int busType)
```

Description

Creates a new application in the Vector Hardware Manager or sets the channel configuration in an existing application. To set an application channel to "not assigned" state set `hwType`, `hwIndex` and `hwChannel`to 0.

Input parameters

- **appName**
  Name of the application to be set.
  Application names are listed in the Vector Hardware Manager.

- **appChannel**
  Application channel (0,1, …) to be accessed.
  If the channel number does not exist, it will be created.
- **hwType**
  Contains the hardware type (see `vxlapi.h`),
  e. g. CANcardXL:
  `XL_HWTYPE_CANCARDXL`
- **hwIndex**
  Index of same hardware types (0,1, ...),
  e. g. for two CANcardXL on one system:
  CANcardXL 01: `hwIndex = 0`
  CANcardXL 02: `hwIndex = 1`
- **hwChannel**
  Channel index on one physical device (0, 1, ...)
  e. g. CANcardXL with `hwIndex`=0:
  Channel 1: `hwChannel = 0`
  Channel 2: `hwChannel = 1`
- **busType**
  Specifies the bus type for the application, e. g.
  `XL_BUS_TYPE_CAN`
  `XL_BUS_TYPE_LIN`
  `XL_BUS_TYPE_DAIO`

Return value — Returns an error code (see section Error Codes on page 360).

3.3.5 xlGetDriverConfig

Syntax

```
XLstatus xlGetDriverConfig(XLdriverConfig *pDriverConfig)
```

Description — Gets detailed information on the hardware configuration. This function can be called at any time after a successfully xlOpenDriver() call. The result describes the current state of the driver configuration after each call.

Note
Applications that search for channels on which they can open a port of a specific bus type should check the corresponding `XL_BUS_ACTIVE_CAP_XXX` (see section XLchannelConfig on page 55)

Input parameters

- **XLdriverConfig**
  Points to the information structure that is returned by the driver (see section XLdriverConfig on page 55).

Return value — Returns an error code (see section Error Codes on page 360).

3.3.6 xlGetRemoteDriverConfig

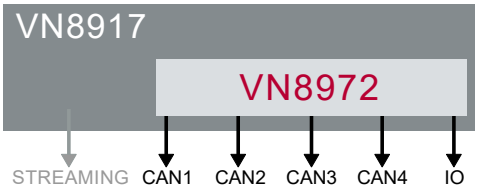
Syntax

```
XLstatus xlGetRemoteDriverConfig(XLdriverConfig *pDriverConfig)
```

Description — This function is similar to xlGetDriverConfig(), but returns the driver configuration of the installed slide-in module (client) in a VN8900 device.

Note
Applications that search for channels on which they can open a port of a specific bus type should check the corresponding `XL_BUS_ACTIVE_CAP_XXX` (see section XLchannelConfig on page 55)

See the following example below for the differences between both function calls (the returned structure is identical):



xlGetDriverConfig()

channelCount	6
STREAMING internal use	channelIndex = 0 hwType = XL_HWTYPE_VN8900 hwChannel = 0 hwIndex = 0
CAN1 VN8950	channelIndex = 1 hwType = XL_HWTYPE_VN8900 hwChannel = 1 hwIndex = 0
CAN2 VN8950	channelIndex = 2 hwType = XL_HWTYPE_VN8900 hwChannel = 2 hwIndex = 0
CAN3 VN8950	channelIndex = 3 hwType = XL_HWTYPE_VN8900 hwChannel = 3 hwIndex = 0
CAN4 VN8950	channelIndex = 4 hwType = XL_HWTYPE_VN8900 hwChannel = 4 hwIndex = 0
IO VN8950	channelIndex = 5 hwType = XL_HWTYPE_VN8900 hwChannel = 5 hwIndex = 0

Input parameters

- **XLdriverConfig**
  Points to the XLdriverConfig structure for the information which is returned by the driver.

Return value

Returns an error code (see section Error Codes on page 360).

Note
It is not possible to access the DLL version of the VN8900 device through the parameter `dllVersion`. This parameter always returns 0.

3.3.7 xlGetChannelIndex

Syntax

```
int xlGetChannelIndex (
```

```
  int hwType,
  int hwIndex,
  int hwChannel);
```

Description

Retrieves the channel index of a particular hardware channel.

Input parameters

- **hwType**
  Required to distinguish the different hardware types, e. g.
  `-1`
  `XL_HWTYPE_CANCARDXL`
  `XL_HWTYPE_CANBOARDXL`
  …
  Parameter `-1` can be used, if the hardware type does not matter.
- **hwIndex**
  Required to distinguish between two or more devices of the same hardware type (-1, 0, 1…). Parameter `-1` can be used to retrieve the first available hardware. The type depends on `hwType`.
- **hwChannel**
  Required to distinguish the hardware channel of the selected device (-1, 0, 1, …). Parameter `-1` can be used to retrieve the first available channel.

Return value

Returns the channel index.

3.3.8 xlGetChannelMask

Syntax

```
XLaccess xlGetChannelMask (
  int hwType,
  int hwIndex,
  int hwChannel);
```

Description

Retrieves the channel mask of a particular hardware channel. Typically, the parameters are directly read from the Vector Hardware Manager via xlGetApplConfig().

Input parameters

- **hwType**
  Required to distinguish the different hardware types, e. g.
  `-1`
  `XL_HWTYPE_CANCARDXL`
  `XL_HWTYPE_CANBOARDXL`
  …
  Parameter `-1` can be used if the hardware type does not matter.
- **hwIndex**
  Required to distinguish between two or more devices of the same hardware type (-1, 0, 1…). Parameter `-1` is used to retrieve the first available hardware. The type depends on `hwType`.
- **hwChannel**
  Required to distinguish the hardware channel of the selected device (-1, 0, 1, …). Parameter `-1` can be used to retrieve the first available channel.

Return value

Returns the channel mask.

Example

Selecting CANcardXL Channel 1

```
m_xlChannelMask = xlGetChannelMask(XL_HWTYPE_CANCARDXL,-1, 0);
if(!m_xlChannelMask) return XL_ERR_HW_NOT_PRESENT;
xlPermissionMask = m_xlChannelMask;
xlStatus = xlOpenPort(&m_XLportHandle,
                      "xlCANdemo",
                      m_xlChannelMask,
                      &xlPermissionMask,
                      1024,
                      XL_INTERFACE_VERSION,
                      XL_BUS_TYPE_CAN);
```

Example

Opening port with two channels and queue size of 256 events

```
// calculate the channelMask for both channel
m_xlChannelMask_both = m_xlChannelMask[MASTER] |
                       m_xlChannelMask[SLAVE];
xlPermissionMask = m_xlChannelMask_both;
xlStatus = xlOpenPort(&m_XLportHandle,
                                    "LIN Example",
                                    m_xlChannelMask_both,
                                    &xlPermissionMask,
                                    256,
                                    XL_INTERFACE_VERSION,
                                   XL_BUS_TYPE_LIN);
```

3.3.9 xlCreatePort

Syntax

```
XLstatus xlCreatePort(
  XlportHandle *portHandle,
  const char *appName,
  unsigned int rxQueueSize,
  unsigned int xlInterfaceVersion,
  unsigned __int64 busType)
```

Description

This function is an alternative to xlOpenPort() and creates a port handle with the specified properties.

Use this function if you want to use **more than 64** channels in your application (see section Flowchart for More Than 64 Channels on page 30).

Input parameters

- **appName**
  The name of the application that is displayed in the Vector Hardware Manager.
- **rxQueueSize**
  CAN, LIN, DAIO
  Size of the port receive queue, which is allocated by the driver. Specifies how many events can be stored in the queue. The value must be a power of two in the range 16...32768. The actual size is rxQueueSize-1.

  CAN FD, FlexRay, Ethernet
  Size of the port receive queue allocated by the driver in bytes.

- **xlInterfaceVersion**
  Current API version, e. g.:
  `XL_INTERFACE_VERSION` for CAN, LIN, DAIO.
  `XL_INTERFACE_VERSION_V4` for FlexRay, Ethernet and CAN FD.
- **busType**
  The bus type to be used with this port, e. g.:
  - `XL_BUS_TYPE_LIN` for LIN
  - `XL_BUS_TYPE_CAN` for CAN
  - `XL_BUS_TYPE_DAIO` for DAIO
  - `XL_BUS_TYPE_FLEXRAY` for FlexRay
  - `XL_BUS_TYPE_STATUS` for time sync status notifications

Output parameters

- **portHandle**
  Pointer to the `portHandle` variable. This handle must be used for all further calls to the port. If `-1` is returned, the port has not been created.

Return value

Returns the status of the call.

3.3.10 xlAddChannelToPort

Syntax

```
XLstatus xlAddChannelToPort(
  XlportHandle portHandle,
  XLaccess        channelIndex,
  unsigned int initAccess,
  unsigned int *permission,
  unsigned __int64 busType)
```

Description

Adds a single channel to the port handle already created. Before the function can be called, the port must be created with xlCreatePort().

The function checks the consistency of the channel and creates the driver handle. The function returns an error if the caller attempts to add a channel to a port more than once.

Input parameters

- **portHandle**
  Valid `portHandle` of the port to which the channel is to be added.The bus type of the port must match that of the channel.
- **channelIndex**
  Channel index of the channel to be added to the port. The channel index can be determined using the xlCreateDriverConfig() or xlGetChannelIndex() function.
- **initAccess**
  Specifies the required authorization of the channel.
  - `1`: initAccess (extended authorization).
  - `0`: no InitAccess.
- **busType**
  Specifies the bus type of the channel. The specified value must match the bus type of the port.

Output parameters

- **permission**
  Returns the authorization of the channel.
  - `1`: initAccess (extended authorization).

- `0`: no InitAccess.

Return value

Returns the status of the call.
The call may fail for the following reasons:

- The bus type of the port does not match the bus type of the channel.
- The channel does not grant init access for a bus type that requires it (e .g. LIN).
- The channel index is invalid.
- The port handle is invalid. If the user requires init access but the channel does not provide any, the port can be closed with xlClosePort().

3.3.11 xlFinalizePort

Syntax

```
XLstatus xlFinalizePort(
  XlportHandle portHandle)
```

Description

Completes the addition of channels to a port. Now the channels can be activated with xlActivatChannel(). A further xlAddChannelToPort() is no longer possible.

Input parameters

► **portHandle**
`portHandle`of the port that is to be completed.

Return value

Returns the status of the call.
The call may fail for the following reasons:

- The bus type of the port does not match the bus type of the channel.
- The channel does not grant init access for a bus type that requires it (e .g. LIN).
- The channel index is invalid.
- The port handle is invalid. If the user requires init access but the channel does not provide any, the port can be closed with xlClosePort().

3.3.12 xlOpenPort

Syntax

```
XLstatus xlOpenPort(
  XlportHandle *portHandle,
  char         *userName,
  XLaccess accessMask,
  XLaccess *permissionMask,
  unsigned int rxQueueSize,
  unsigned int xlInterfaceVersion,
  unsigned int busType)
```

Description

Opens a port for a bus type (e. g. CAN) and grants access to the different channels that are selected by the `accessMask`. It is possible to open more ports on a specific channel, but only the first one gets **init access**. The `permissionMask` returns the channels which get **init access**.

Use this function if you want to use **up to 64** channels in your application (see section Flowchart for up to 64 Channels on page 30). If you need **more than 64** channels in your application, please use xlCreatePort() instead.

Input parameters

► **userName**
The name of the application that is listed in the Vector Hardware Manager.

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

- **rxQueueSize**
  CAN, LIN, DAIO, K-Line
  Size of the port receive queue allocated by the driver. Specifies how many events can be stored in the queue. The value must be a power of 2 and within a range of 16…32768. The actual queue size is `rxQueueSize-1`. The rxQueuesize depends on the **busType** and queue version.
  - V3: queue size in events
  - V4: queue size in bytes

- **CAN FD**
  Size of the port receive queue allocated by the driver in bytes. The value must be a power of 2 and within a range of 8192…524288 bytes (0.5 MB).

  FlexRay
  Size of the port receive queue allocated by the driver in bytes. The value must be a power of 2 and within a range of 8192…1048576 bytes (1 MB).

  Ethernet
  Size of the port receive queue allocated by the driver in bytes. The value must be a power of 2 and within a range of 65536…8*1024*1024 bytes (8 MB).

  ARINC
  Size of the port receive queue allocated by the driver in bytes. The value must be a power of 2 and within a range of 8192…524288 bytes (0.5 MB).

- **xlInterfaceVersion**
  Current API version, e. g.:
  `XL_INTERFACE_VERSION` for CAN, LIN, DAIO, K-Line.
  `XL_INTERFACE_VERSION_V4` for CAN FD, Ethernet, FlexRay, ARINC429

- **busType**
  Bus type that should be activated, e. g.:
  `XL_BUS_TYPE_NONE`
  `XL_BUS_TYPE_CAN`
  `XL_BUS_TYPE_LIN`
  `XL_BUS_TYPE_KLINE`
  `XL_BUS_TYPE_FLEXRAY`
  `XL_BUS_TYPE_AFDX`
  `XL_BUS_TYPE_DAIO`
  `XL_BUS_TYPE_J1708`
  `XL_BUS_TYPE_ETHERNET`
  `XL_BUS_TYPE_A429`

Output parameters

- **portHandle**
  Pointer to a variable that receives the `portHandle`. This handle must be used for all further calls to the port. If `XL_INVALID_PORT_HANDLE` is returned, the port was neither created nor opened.

Input/Output parameters

- **permissionMask**
  on output
  Pointer to a variable that receives the mask for those channels that have **init access**.

  on input
  As input, this is the channel mask where **init access** is requested.

Return value

Returns an error code (see section Error Codes on page 360).

Note
For LIN (`busType = XL_BUS_TYPE_LIN`), **init access** is needed (see section Introduction on page 189). If the LIN channel gets no **init access** the function returns `XL_ERR_INVALID_ACCESS`.

3.3.13 xlClosePort

Syntax

```
XLstatus xlClosePort (XLportHandle portHandle)
```

Description

This function closes a port and deactivates its channels.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Return value

Returns an error code (see section Error Codes on page 360).

3.3.14 xlSetTimerRate

Syntax

```
XLstatus xlSetTimerRate (
  XLportHandle portHandle
  unsigned long timerRate)
```

Description

This call sets the rate for the port's cyclic timer events.
The resolution of `timeRate` is 10 µs, but the internal step width is 1000 µs. Values less than multiples of 1000 µs will be rounded down (truncated) to the next closest value.

Examples:
`timerRate = 105`: 1050 µs → 1000 µs
`timerRate = 140`: 1400 µs → 1000 µs
`timerRate = 240`: 2400 µs → 2000 µs
`timerRate = 250`: 2500 µs → 2000 µs

The minimum timer rate value is 1000 µs (`timerRate = 100`).
If more than one application uses the timer events the lowest value will be used for all.

Example:
Application 1 `timerRate = 150` (1000 µs)
Application 2 `timerRate = 350` (3000 µs)
Used timer rate → 1000 µs

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **timerRate**
  Value specifying the interval for cyclic timer events generated by a port.
  If 0 is passed, no cyclic timer events will be generated.

Return value — Returns an error code (see section Error Codes on page 360).

3.3.15 xlSetTimerRateAndChannel

Syntax

```
XLstatus xlSetTimerRateAndChannel (
  XLportHandle  portHandle
  XLaccess      *timerChannelMask
  unsigned long *timerRate)
```

Description — This call sets the rate for the port's cyclic timer events. The resolution is 10 µs (`timerRate` of 1 means 10 µs, a `timerRate` of 10 means 100 µs).

The minimum and maximum `timerRate` values depend on the hardware. If a value is outside of the allowable range the limit value is used. Only deterministic values according to the following list can be used. Other values will be rounded to the next faster timer rate.

- **CAN/LIN**
  Minimum `timerRate`: 250 µs
  Discrete `timerRate` values: 250 µs + x * 250 µs
- **Ethernet**
  Minimum `timerRate`: 250 µs
  Discrete `timerRate` values: 250 µs + x * 250 µs
- **FlexRay (USB)**
  Minimum `timerRate`: 250 µs
  Discrete `timerRate` values: 250 µs + x * 50 µs
- **FlexRay (PCI)**
  Minimum `timerRate`: 100 µs
  Discrete `timerRate` values: 100 µs + x * 50 µs

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **timerChannelMask**
  A mask specifying the channels, at which the timer events may be generated. Please note that the driver selects the best suitable (accurate) channel of the entire channel mask for timer event generation. This selected channel is returned in `timerChannelMask`.
- **timerRate**
  Value specifying the interval for cyclic timer events generated by a port. If 0 is passed, no cyclic timer events will be generated.

Return value — Returns an error code (see section Error Codes on page 360).

3.3.16 xlResetClock

Syntax

```
XLstatus xlResetClock (XLportHandle portHandle)
```

Description — Resets the time stamps (in nanoseconds) for the specified port.



Note
This function cannot be used with the new synchronization protocols (see section TimeSync Commands on page 73).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Return value

Returns an error code (see section Error Codes on page 360).

3.3.17 xlSetNotification

Syntax

```
XLstatus xlSetNotification (
  XLportHandle portHandle,
  XLhandle     *handle,
  int          queueLevel)
```

Description

The function sets the queue level for notifications on the receive queue of the given port and returns the notification handle for that queue. This notification handle is a handle to an auto-resetting Windows event and remains valid until the port is closed. The application may pass this handle to the Windows `WaitForSingleObject()` or `WaitForMultipleObjects()` functions to await incoming driver events, as demonstrated in the xlReceive example.

For each event written, the driver signals the Windows event if the resulting receive queue level is larger or equal to the queue level set by this function.

Whether the queue level is evaluated in bytes or number of events depends on the port as described for the `rxQueueSize` parameter of xlOpenPort. Passing `queueLevel=1` therefore instructs the driver to signal the event as soon as the receive queue is not empty anymore, which is what applications usually require.

Windows events have a signaled and a non-signaled state but are not counting. `WaitForSingleObject()` and `WaitForMultipleObjects()` block the calling thread until the Windows event reaches the signaled state and reset the event to the non-signaled state when the thread execution continues. Multiple driver events might have been inserted before the Windows event was reset. To ensure that all incoming driver events are eventually processed, the thread must consequently call the ports receive function (for example xlReceive, xlCanReceive, ...) in a loop until the receive function returns `XL_ERR_QUEUE_IS_EMPTY` before waiting again.



Note
There is no one-to-one relationship between driver events in the receive queue and the XL API events returned by the receive function.

Some driver events do not have a corresponding XL API event and therefore the receive function may return `XL_ERR_QUEUE_IS_EMPTY` although the Windows event was signaled.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **queueLevel**
  Queue level in number of events or number of bytes to set on the queue. For LIN, this is fixed to '1'. For other bus types, '1' is the recommended value.

Output parameters

- **handle**
  Pointer to a WIN32 event handle.

Return value

Returns an error code (see section Error Codes on page 360).



Note
Applications only need to call xlSetNotification **once** after opening the port.

3.3.18 xlFlushReceiveQueue

Syntax

```
XLstatus xlFlushReceiveQueue (XLportHandle portHandle)
```

Description

This function flushes the port's receive queue.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Return value

Returns an error code (see section Error Codes on page 360).

3.3.19 xlGetReceiveQueueLevel

Syntax

```
XLstatus xlGetReceiveQueueLevel (
  XLportHandle portHandle,
  int          *level)
```

Description

This function reads the number of events or number of bytes currently in use in a port's receive queue. Applications can use this value to compare the actual queue usage to the allocated size requested by the `rxQueueSize` parameter of xlOpenPort().

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Output parameters

- **level**
  Pointer to an int that receives the actual count of events or bytes. The value depends on the bus type (see section xlOpenPort on page 38).

Return value

Returns an error code (see section Error Codes on page 360).

Note
The driver events in the receive queue have a different format (e. g. different length) than the XL API events returned by the receive functions (for example xlReceive, xlCanReceive, ...) and there is no one-to-one relationship between driver and XL API events.

Therefore, applications cannot use xlGetReceiveQueueLevel to test how many events the receive function will return. In particular, a returned level larger zero does not guarantee that the receive function will return a XL API event, as some driver events do not have a corresponding XL API event.

The correct and efficient way to check that no more XL API events are available is to call the receive function and test for the `XL_ERR_QUEUE_IS_EMPTY` return value.

3.3.20 xlActivateChannel

Syntax

```
XLstatus xlActivateChannel(
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int busType,
  unsigned int flags)
```

Description

Goes ‚on bus' for the selected port and channels. At this point, the user can transmit and receive messages on the bus.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **busType**
  Bus type that has also been used for xlOpenPort().
- **flags**
  Additional flags for activating the channels:
  - `XL_ACTIVATE_RESET_CLOCK`
    Resets the internal clock (CAN/FlexRay/Ethernet) after activating the channel. This flag cannot be used with the new synchronization protocols (see section TimeSync Commands on page 73).
  - `XL_ACTIVATE_NONE`

Return value

Returns an error code (see section Error Codes on page 360).

Example

Channel Activation

```
xlStatus = xlActivateChannel(m_vPortHandle,
                             &m_vChannelMask[MASTER],
                             XL_BUS_TYPE_CAN,
                             XL_ACTIVATE_RESET_CLOCK);
```

3.3.21 xlReceive

Syntax

```
XLstatus xlReceive (
  XLportHandle portHandle,
  unsigned int *pEventCount,
  XLevent      *pEventList)
```

Description

Reads the received events from the message queue.
Supported bus types:

- CAN
- LIN
- K-Line
- DAIO

An application should read all available messages to be sure to re-enable the event. An overrun of the receive queue can be determined by the message flag `XL_EVENT_FLAG_OVERRUN` in `XLevent.flags`.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Input/output parameters

- **pEventCount**
  Pointer to an event counter. On input, the variable must be set to the size (in messages) of the received buffer. On output, the variable contains the number of received messages.
- **pEventList**
  Pointer to the application allocated receive event buffer (see section XLevent on page 70). The buffer must be large enough to hold the requested messages (`pEventCount`).

Return value

`XL_ERR_QUEUE_IS_EMPTY`: No event is available (see section Error Codes on page 360)

Example

Reading messages from queue

```
XLhandle     h;
unsigned int msgsrx = 1;
XLevent      xlEvent;
vErr = xlSetNotification(XLportHandle, &h, 1);


// Wait for event
while (g_RXThreadRun) {

  WaitForSingleObject(g_hMsgEvent,10);
  xlStatus = XL_SUCCESS;

  while (!xlStatus) {

    msgsrx = RECEIVE_EVENT_SIZE;
    xlStatus = xlReceive(g_xlPortHandle, &msgsrx, &xlEvent);
    if ( xlStatus!=XL_ERR_QUEUE_IS_EMPTY ) {
      if (!g_silent) {
        printf("%s\n", xlGetEventString(&xlEvent));
      }
    }
  }
}
```

3.3.22 xlGetEventString

Syntax

```
XLstringType xlGetEventString (XLevent *ev)
```

Description

Returns the textual description of the given event.
Supported bus types and events:

- CAN (not CAN FD/XL)
- LIN
- partly DAIO
- common events (e. g. TIMER events)

This function is **not thread-save**.

Input parameters

- **ev**
  Points to the event (see section XLevent on page 70).

Return value

Text string.



Example

Returned string

```
RX_MSG c=4,t=794034375, id=0004 l=8, 0000000000000000 TX tid=CC
```

Explanation:

- **RX_MSG**
  Rx message
- **c=4**
  On channel 4.
- **t=794034375**
  Time stamp of 794034375 ns.
- **id=004**
  ID is 4.
- **l=8**
  DLC of 8
- **0000000000000**
  D0 to D7 are set to 0.
- **TX tid=CC**
  Tx flag, message was transmitted successfully by the CAN controller.

3.3.23 xlGetErrorString

Syntax

```
const char *xlGetErrorString (XLstatus err)
```

Description

Returns the textual description of the given error code.

Input parameters

- **err**
  Error code (see section Error Codes on page 360)

Return value

Error code as plain text string.

3.3.24 xlGetSyncTime

Syntax

```
XLstatus xlGetSyncTime (
  XlportHandle portHandle,
  XLuint64     *time)
```

Description

Returns the current high precision PC time (in ns).

Note

- If the software time synchronization is active, the event time stamp is synchronized to the PC time. If the XL API function xlResetClock() was not called, the event time stamp can be compared to the time retrieved from xlGetSyncTime().
- Application can use this function to query the domain time when the time domain timescale is `XL_TS_TIMESCALE_PERFORMANCE_COUNTER`.
- If no time synchronization protocol is active on the domain, the function will still return the performance counter for backward compatibility.
- When used with new time synchronization protocols (see section TimeSync Commands on page 73), this function returns `XL_ERR_NOT_IMPLEMENTED`.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Output parameters

- **time**
  Points to a variable that receives the sync time.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.25 xlGetChannelTime

Syntax

```
xlGetChannelTime (
  XLportHandle portHandle,
  XLaccess     accessMask,
  XLuint64     *pChannelTime)
```

Description

This function is available only on VN8900 devices and returns the 64 bit PC-based card time.

Note
This function is still only available for the VN8900 interface mode, VX1135, VX1161, and VT6306 and provides the synchronized time stamps of the plug-in module. Available for channel-based access only. This function returns the synchronized time for VX1135, VX1161 and VT6306 when the device's measurement clock is synchronized by the Vector Time Synchronization Service.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

Output parameters

- **pChannelTime**
  64 bit PC-based card time.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.26 xlGenerateSyncPulse

Syntax

```
XLstatus xlGenerateSyncPulse (
  XlportHandle portHandle,
  XLaccess     accessMask)
```

Description

This function generates a sync pulse at the hardware synchronization line (hardware party line) with a maximum frequency of 10 Hz. It is only allowed to generate a sync pulse at **one channel** and at one device at the same time.



Note
This function fails if the Vector Time Synchronization Service based hardware synchronization protocol is enabled on the Vector device.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.27 xlPopupHwConfig

Syntax

```
XLstatus xlPopupHwConfig (
  char         *callSign,
  unsigned int waitForFinish)
```

Description

Call this function to pop up the hardware configuration.

Input parameters

- **callSign**
  May be NULL. If not NULL, `callSign` is appended to the command line. Therefore, `callSign` can contain whitespace separated arguments to Vector Hardware Config. The function prefers to call the Vector Hardware Manager (VHM). It will fall back to the old Vector Hardware Config (`vcanconf.exe`) if
  - a non-null and non-empty `callSign` is passed or
  - the VHM is not installed.
- **waitForFinish**
  Timeout (for the application) to wait for the user entry within **Vector Hardware Config** in milliseconds.
  `0`: The application does not wait.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.28 xlDeactivateChannel

Syntax

```
XLstatus xlDeactivateChannel (
  XlportHandle portHandle,
  XLaccess     accessMask)
```

Description

The selected channels go off the bus. The channels are deactivated if there is no further port that activates the channels.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.29 xlGetLicenseInfo

Syntax

```
XLstatus xlGetLicenseInfo (
  XLaccess      channelMask,
  XLlicenseInfo *pLicInfoArray,
  unsigned int  licInfoArraySize)
```

Description

This function returns an array (type of XLlicenseInfo) with all available licenses from the selected Vector device. The order of available licenses is always the same, since each element with its index is dedicated to a license. Whether a license is available or not can be checked within the related structure.

Input parameters

- **channelMask**
  The channel mask of the Vector device containing the licenses.
- **licInfoArraySize**
  Size of the array.

Output parameters

- **pLicInfoArray**
  Pointer to the array to be returned (see section XLlicenseInfo on page 62).

Return value

Returns an error code (see section Error Codes on page 360).

3.3.30 xlSetGlobalTimeSync

Syntax

```
XLstatus xlSetGlobalTimeSync (
  unsigned long newValue,
  unsigned long *previousValue
);
```

Description

Reads/sets the software synchronization setting in the Vector Hardware Manager. This setting is written to the registry and read every time when the driver is loaded. To reload the driver of a connected interface, disconnect and reconnect it (or reboot the PC).

Input parameters

- **newValue**
  `XL_SET_TIMESYNC_NO_CHANGE`
  Use this value to read the current setting which is stored in `previousValue`.

  `XL_SET_TIMESYNC_ON`
  Enables the software synchronization in the Vector Hardware Manager.

  `XL_SET_TIMESYNC_OFF`
  Disables the software synchronization in the Vector Hardware Manager.

Output parameters

- **previousValue**
  Buffer which stores the previous value.

Return value

Returns an error code (see section Error Codes on page 360).

3.3.31 xlGetKeymanBoxes

Syntax

```
XLstatus xlGetKeymanBoxes(unsigned int* boxCount);
```

Description

Returns the number of connected Keyman license dongles. In addition, all available library relevant licenses (new license model only, e. g. advanced FlexRay support) found on any connected Vector network interface are activated. The activation is required to use the advanced features of the XL API. For further details, refer to section License Management on page 27.

Output parameters

- **boxCount**
  Number of connected Keyman license dongles.

Return value

Returns `XL_SUCCESS` if returned `boxCount` is larger than 0 or if at least one library feature was unlocked via the new license model. Otherwise the function returns an error code (see section Error Codes on page 360).

3.3.32 xlGetKeymanInfo

Syntax

```
XLstatus xlGetKeymanInfo (
  unsigned int boxIndex,
  unsigned int* boxMask,
  unsigned int* boxSerial,
  XLuint64* licInfo);
```

Description

Returns the serial number and license info (license bits) of a selected Keyman license dongle. This function is also required to activate available licenses (old license model only), e. g. advanced FlexRay support. Otherwise function calls will return `XL_ERR_NO_LICENSE`. In order to activate single licenses of the old license model, get the count of licenses via xlGetKeymanBoxes() and then use the value range (count-1) for the indexs in xlGetKeymanInfo().

Input parameters

- **boxIndex**
  Index of the Keyman license dongle (zero based).

Output parameters

- **boxMask**
  Mask of the Keyman license dongle.
- **boxSerial**
  Serial of the Keyman license dongle.

- **licInfo**
  License Info (license bits in license array).
  The structure's size is 4*64 bits (see example below).

Return value

Returns an error code (see section Error Codes on page 360).



Example

```
XLstatus xlStatus = XL_ERROR;
unsigned int nbrOfBoxes;
unsigned int boxMask;
unsigned int boxSerial;
unsigned int i;
XLuint64 licInfo[4], tmpLicInfo[4];

memset(licInfo, 0, sizeof(licInfo));
xlStatus = xlGetKeymanBoxes(&nbrOfBoxes);

if (xlStatus == XL_SUCCESS) {
  sprintf(tmp, "xlGetKeymanBoxes: %d Keyman License Dongle(s) found!\n",
  nbrOfBoxes);
XLDEBUG(DEBUG_ADV, tmp);

  for (i = 0; i<nbrOfBoxes; i++) {
    memset(tmpLicInfo, 0, sizeof(tmpLicInfo));
    xlStatus = xlGetKeymanInfo(i, &boxMask, &boxSerial, tmpLicInfo);
    if (xlStatus == XL_SUCCESS) {
      sprintf(tmp, "xlGetKeymanInfo: Keyman Dongle (%d) with Seri-
alNumber:
      %d-%d\n", i, boxMask, boxSerial);
    XLDEBUG(DEBUG_ADV, tmp);

      licInfo[0] |= tmpLicInfo[0];
      licInfo[1] |= tmpLicInfo[1];
      licInfo[2] |= tmpLicInfo[2];
      licInfo[3] |= tmpLicInfo[3];
    }
  }

  sprintf(tmp, "xlGetKeymanInfo: licInfo[0]=0x%I64x, licInfo[1]=0x%I64x,
  licInfo[2]=0x%I64x, licInfo[3]=0x%I64x\n",
  licInfo[0], licInfo[1], licInfo[2], licInfo[3]);
  XLDEBUG(DEBUG_ADV, tmp);
}
```

3.3.33 xlCreateDriverConfig

Syntax

```
XLstatus xlCreateDriverConfig (
  XLIdriverConfigVersion version,
  struct XLIDriverConfig *pConfigInterface)
```

Description

This function allocates a structure that holds information on the hardware configuration. Compared to its predecessors xlGetDriverConfig() and xlGetRemoteDriverConfig(), it has the following advantages:

- Has a versioned interface, which will allow later XL API versions to add additional fields and structs.
- Provides information on networks, segments, measurement points and virtual ports (see section Network Based Access Mode on page 136).
- Logically separates devices from the channels on these devices.
- Is not limited to 64 channels.

- Combines the local and remote channel information in a common structure.

The application may call xlCreateDriverConfig at any time after a successful xlOpenDriver() call. The returned instance of the `XLIDriverConfig`structure holds the state of the driver configuration at the time of the call. The function pointers contained in the returned structure query this state. An application may hold multiple instances of `XLIDriverConfig` structures at the same time. Once the application does not need an instance anymore, it must release the instance with xlDestroyDriverConfig().

Input parameters

- **version**
  Requested version of `XLIDriverConfig`. Currently the only value is `XL_IDRIVER_CONFIG_VERSION_1`.

Output parameters

- **pConfigInterface**
  Pointer to a structure specified by the version parameter. For `XL_IDRIVER_CONFIG_VERSION_1`, the structure must have type XLapiIDriverConfigV1.

Return value

Returns an error code (see section Error Codes on page 360). If the current version of the DLL does not support the requested version of the `XLIDriverConfig`, `XL_ERR_NOT_IMPLEMENTED` is returned.



Example
The following program prints the list of channels grouped by device:

```
XLapiIDriverConfigV1 config;
XLdeviceDrvConfigListV1 devices;

XLstatus status = xlCreateDriverConfig(XL_IDRIVER_CONFIG_VERSION_
1,
                                       (XLIDriverConfig*)&config);

if (status != XL_SUCCESS) {
  return status;
}

status = config.fctGetDeviceConfig(config.configHandle,
&devices);

if (status == XL_SUCCESS) {
  for (unsigned int i = 0; i < devices.count; ++i) {
    const XLdeviceDrvConfigV1& device = devices.item[i];
    printf("%s\n", device.name);

    for (unsigned int j = 0; j < device.channelList.count; ++j) {
      const XLchannelDrvConfigV1& channel =
         device.channelList.item[j];

      printf("- [%u] %s\n",
             channel.channelIndex,
             channel.transceiver.name);
    }
  }
}

xlDestroyDriverConfig(config.configHandle);

return status;
```

3.3.34 xlDestroyDriverConfig

Syntax

```
XLstatus xlDestroyDriverConfig(
  XLdrvConfigHandle configHandle)
```

Description

This function releases the memory allocated for an instance of the `XLIDriver-Config` structure.

Input parameters

- **configHandle**
  Handle for the `XLIDriverConfig` instance to be released. The handle is returned by xlCreateDriverConfig() in a field of the allocated structure.

Return value

Returns an error code (see section Error Codes on page 360).



Note

The structure allocated by xlCreateDriverConfig() consists of multiple memory objects (for example strings and arrays) that are accessible via pointers. As xlDestroyDriverConfig releases all these memory objects, the application must be careful not to dereference any pointers of a released `XLIDriverConfig` structure.

3.4 Structs

3.4.1 XLdriverConfig

Syntax

```
typedef struct s_xl_driver_config {
unsigned int    dllVersion;
unsigned int    channelCount;
unsigned int    reserved[10];
XLchannelConfig channel[XL_CONFIG_MAX_CHANNELS];
} XLdriverConfig;
```

Description

The driver returns a structure containing the following information:

Parameters

- **dllVersion**
  The used dll version:
  `(DRIVER_VERSION_MAJOR<<24) |`
  `(DRIVER_VERSION_MINOR<<16) |`
  `DRIVER_VERSION_BUILD;`
- **channelCount**
  The number of available channels.
- **reserved**
  Reserved for future use.
- **channel**
  Structure containing channels information (see section XLchannelConfig on page 55). `XL_CONFIG_MAX_CHANNELS=64.`

3.4.2 XLchannelConfig

Syntax

```
typedef struct s_xl_channel_config {
char           name [XL_MAX_LENGTH + 1];
unsigned cha   hwType;
unsigned char  hwIndex;
unsigned char  hwChannel;
unsigned short transceiverType;
unsigned int   transceiverState;
unsigned char  channelIndex;
XLuint64       channelMask;
unsigned int   channelCapabilities;
unsigned int   channelBusCapabilities;
unsigned char  isOnBus;
unsigned int   connectedBusType;
XLbusParams    busParams;
unsigned int   driverVersion;
unsigned int   interfaceVersion;
unsigned int   raw_data[10];
unsigned int   serialNumber;
unsigned int   articleNumber;
char           transceiverName [XL_MAX_LENGTH + 1];
unsigned int   specialCabFlags;
unsigned int   dominantTimeout;
unsigned in    reserved[8];
} XLchannelConfig;
```

Description

This structure is used in **XLdriverConfig** (see section XLdriverConfig on page 55).

Parameters

- **name**
  The channel's name.
- **hwType**
  Contains the hardware types (see `vxlapi.h`),
  e. g. CANcardXL: `XL_HWTYPE_CANCARDXL`
- **hwIndex**
  Index of same hardware types (0, 1, ...),
  e. g. for two CANcardXL on one system:
  CANcardXL 01: `hwIndex = 0`
  CANcardXL 02: `hwIndex = 1`
- **hwChannel**
  Channel index on one physical device (0, 1, ...)
  e. g. CANcardXL with `hwIndex=0`:
  Channel 1: `hwChannel = 0`
  Channel 2: `hwChannel = 1`
- **transceiverType**
  Contains type of Cab or Piggyback,
  e. g. 251 Highspeed Cab: `XL_TRANSCEIVER_TYPE_CAN_251`
- **transceiverState**
  State of the transceiver.
- **channelIndex**
  Global channel index (0, 1, ...).
- **channelMask**
  Global channel mask (`1 << channelIndex`).
- **channelCapabilities**
  `XL_CHANNEL_FLAG_TIME_SYNC_RUNNING`
  `XL_CHANNEL_FLAG_NO_HWSYNC_SUPPORT`
  `XL_CHANNEL_FLAG_LOG_CAPABLE`
  `XL_CHANNEL_FLAG_SPDIF_CAPABLE`
  `XL_CHANNEL_FLAG_CANFD_BOSCH_SUPPORT`
  `XL_CHANNEL_FLAG_CANFD_ISO_SUPPORT`

- **channelBusCapabilities**
  Describes the channel and the current transceiver features. Applications that search for channels on which they can open a port of a specific bus type should check the corresponding `XL_BUS_ACTIVE_CAP`.

  The channel (hardware) supports the bus types:
  `XL_BUS_COMPATIBLE_CAN`
  `XL_BUS_COMPATIBLE_LIN`
  `XL_BUS_COMPATIBLE_FLEXRAY`
  `XL_BUS_COMPATIBLE_DAIO`
  `XL_BUS_COMPATIBLE_J1708`
  `XL_BUS_COMPATIBLE_ETHERNET`
  `XL_BUS_COMPATIBLE_A429`
  `XL_BUS_COMPATIBLE_KLINE`

  The connected Cab or Piggyback supports the bus type:
  `XL_BUS_ACTIVE_CAP_CAN`
  `XL_BUS_ACTIVE_CAP_LIN`
  `XL_BUS_ACTIVE_CAP_FLEXRAY`
  `XL_BUS_ACTIVE_CAP_DAIO`
  `XL_BUS_ACTIVE_CAP_J1708`
  `XL_BUS_ACTIVE_CAP_ETHERNET`
  `XL_BUS_ACTIVE_CAP_A429`
  `XL_BUS_ACTIVE_CAP_KLINE`

- **isOnBus**
  The flag specifies whether the channel is on bus (1) or off bus (0).
- **connectedBusType**
  The flag specifies to which bus type the channel is connected, e. g.
  `XL_BUS_TYPE_CAN`
  ...

  Note: The flag is only set when the channel is on bus.
- **busParams**
  Current bus parameters (see section XLbusParams on page 58).
- **driverVersion**
  Current driver version.
- **interfaceVersion**
  Current interface API version, e. g. `XL_INTERFACE_VERSION`
- **raw_data**
  Only for internal use.
- **serialNumber**
  Hardware serial number.
- **articleNumber**
  Hardware article number.
- **transceiverName**
  Name of the connected transceiver.
- **specialCabFlags**
  Only for internal use.

- **dominantTimeout**
  Only for internal use.
- **reserved**
  Reserved for future use.

3.4.3 XLbusParams

Syntax

```
typedef struct {
  unsigned int busType;
  union {
    struct {
      unsigned int  bitRate;
      unsigned char sjw;
      unsigned char tseg1;
      unsigned char tseg2;
      unsigned char sam; // 1 or 3
      unsigned char outputMode;
      unsigned char reserved[7];
      unsigned char canOpMode;
    } can;

    struct {
      unsigned int  arbitrationBitRate;
      unsigned char sjwAbr;
      unsigned char tseg1Abr;
      unsigned char tseg2Abr;
      unsigned char samAbr;
      unsigned char outputMode;
      unsigned char sjwDbr;
      unsigned char tseg1Dbr;
      unsigned char tseg2Dbr;
      unsigned int  dataBitRate;
      unsigned char canOpMode;
    } canFD;

    struct {
      unsigned int status;
      unsigned int cfgMode;
      unsigned int baudrate;
    } flexray;

    struct {
      unsigned char macAddr[6];
      unsigned char connector;
      unsigned char phy;
      unsigned char link;
      unsigned char speed;
      unsigned char clockMode;
      unsigned char bypass;
    } ethernet;

    struct {
      unsigned short channelDirection;
      unsigned short res1;
      union {
        struct {
          unsigned int bitrate;
          unsigned int parity;
          unsigned int minGap;
        } tx;

        struct {
          unsigned int bitrate;
          unsigned int minBitrate;
```

```
          unsigned int maxBitrate;
          unsigned int parity;
          unsigned int minGap;
          unsigned int autoBaudrate;
        } rx;
        unsigned char raw[24];
      } dir;
    } a429;

    unsigned char raw[28];
  } data;
} XLbusParams;
```

Description

Structure used in XLchannelConfig.

Parameters

- **busType**
  Specifies the bus type for the application.

CAN

- **bitRate**
  This value specifies the real bitrate (e. g. 125000).
- **sjw**
  Bus timing value sample jump width.
- **tseg1**
  Bus timing value tseg1.
- **tseg2**
  Bus timing value tseg2.
- **sam**
  Bus timing value sam. Samples may be 1 or 3.
- **outputMode**
  Actual output mode of the CAN chip.
- **reserved**
  For future use.
- **canOpMode**
  CAN 2.0: `XL_BUS_PARAMS_CANOPMODE_CAN20`
  CAN FD: `XL_BUS_PARAMS_CANOPMODE_CANFD`
  CAN FD NO ISO: `XL_BUS_PARAMS_CANOPMODE_CANFD_NO_ISO`

CAN FD

- **arbitrationBitRate**
  CAN bus timing for nominal/arbitration bitrate.
- **sjwAbr**
  Bus timing value sample jump width (arbitration).
- **tseg1Abr**
  Bus timing value tseg1 (arbitration).
- **tseg2Abr**
  Bus timing value tseg2 (arbitration).
- **samAbr**
  Bus timing value sam (arbitration).
- **outputMode**
  Actual output mode of the CAN chip.

- **sjwDbr**
  CAN bus timing for data bitrate.
- **tseg1Dbr**
  Bus timing value tseg1.
- **tseg2Dbr**
  Bus timing value tseg1.
- **dataBitRate**
  Data bitrate.
- **canOpMode**
  CAN 2.0: `XL_BUS_PARAMS_CANOPMODE_CAN20`
  CAN FD: `XL_BUS_PARAMS_CANOPMODE_CANFD`

FlexRay

- **status**
  `XL_FR_CHANNEL_CFG_STATUS_INIT_APP_PRESENT`
  `XL_FR_CHANNEL_CFG_STATUS_CHANNEL_ACTIVATED`
  `XL_FR_CHANNEL_CFG_STATUS_VALID_CLUSTER_CF`
  `XL_FR_CHANNEL_CFG_STATUS_VALID_CFG_MODE`
- **cfgMode**
  `XL_FR_CHANNEL_CFG_MODE_SYNCHRONOUS`
  `XL_FR_CHANNEL_CFG_MODE_COMBINED`
  `XL_FR_CHANNEL_CFG_MODE_ASYNCHRONOUS`
- **baudrate**
  FlexRay baud rate in kBaud.

Ethernet

- **macAddr**
  The MAC address starting with the MSB. This field is only defined on VN5610 (A) and VN5640. In network-based mode, MAC addresses are available via xlNetRequestMACAddress().
- **connector**
  The interface connector currently assigned to the MAC:
  `XL_ETH_STATUS_CONNECTOR_RJ45`
  `XL_ETH_STATUS_CONNECTOR_DSUB`
- **phy**
  The currently active transmitter (physical interface):
  `XL_ETH_STATUS_PHY_UNKNOWN`
  `XL_ETH_STATUS_PHY_802_3`
  `XL_ETH_STATUS_PHY_BROADR_REACH`
- **link**
  Link state:
  `XL_ETH_STATUS_LINK_UNKNOWN`
  `XL_ETH_STATUS_LINK_DOWN`
  `XL_ETH_STATUS_LINK_UP`
  `XL_ETH_STATUS_LINK_ERROR`

- **speed**
  Current Ethernet connection speed:

  `XL_ETH_STATUS_SPEED_UNKNOWN`

  `XL_ETH_STATUS_SPEED_10`
  10 Mbit/s operation.

  `XL_ETH_STATUS_SPEED_100`
  100 Mbit/s operation.

  `XL_ETH_STATUS_SPEED_1000`
  1000 Mbit/s operation.

- **clockMode**
  Clock mode setting of the connection:

  `XL_ETH_STATUS_CLOCK_DONT_CARE`
  Reported for IEEE 802.3.

  `XL_ETH_STATUS_CLOCK_MASTER`
  `XL_ETH_STATUS_CLOCK_SLAVE`

- **bypass**
  `XL_ETH_BYPASS_INACTIVE` (Default)
  `XL_ETH_BYPASS_PHY`
  `XL_ETH_BYPASS_MACCORE`

ARINC 429

- **channelDirection**
  See XL_A429_PARAMS.
- **res1**
  Reserved for future use.
- **bitrate**
  See XL_A429_PARAMS.
- **parity**
  See XL_A429_PARAMS.
- **minGap**
  See XL_A429_PARAMS.
- **bitrate**
  See XL_A429_PARAMS.
- **minBitrate**
  See XL_A429_PARAMS.
- **maxBitrate**
  See XL_A429_PARAMS.
- **parity**
  See XL_A429_PARAMS.
- **minGap**
  See XL_A429_PARAMS.

- **autoBaudrate**
  See XL_A429_PARAMS.
- **raw**
  See XL_A429_PARAMS.
- **dir**

3.4.4 XLlicenseInfo

Syntax

```
typedef struct s_xl_license_info {
unsigned char bAvailable;
char          licName[65];
} XLlicenseInfo;
```

Parameters

- **bAvailable**
  0: license not available
  1: license available
- **licName**
  Name of the license.



Example

Retrieving licenses, check if available

```
XLstatus xlStatus;
char          licAvail[2048];
char          strtmp[512];
XLlicenseInfo licenseArray[1024];
unsigned int licArraySize = 1024;

xlStatus = xlGetLicenseInfo(m_xlChannelMask m_xlCh,
                            licenseArray,
                            licArraySize);

if (xlStatus == XL_SUCCESS) {
  strcpy(licAvail, "Licenses found:\n\n");
  for (unsigned int i = 0; i < licArraySize; i++) {
    if (licenseArray[i].bAvailable) {
      sprintf(strtmp,"ID 0x%03x: %s\n", i,licenseArray
[i].licName);

      if ((strlen(licAvail) + strlen(strtmp)) < sizeof(licAvail))
{
        strcat(licAvail, strtmp);
      }

      else {
        sprintf(licAvail, "Error: String size too small!");
        xlStatus = XL_ERROR;
      }
    }
  }
}
else {
  sprintf(licAvail, "Error: %d", xlStatus);
}
```

3.4.5 XLapiIDriverConfigV1

Syntax

```
typedef struct s_xlapi_driver_config_v1 {
  XLdrvConfigHandle configHandle;
  TP_FCT_XLAPI_GET_DEVICE_CONFIG_V1  fctGetDeviceConfig;
  TP_FCT_XLAPI_GET_CHANNEL_CONFIG_V1 fctGetChannelConfig;
  TP_FCT_XLAPI_GET_NETWORK_CONFIG_V1 fctGetNetworkConfig;
  TP_FCT_XLAPI_GET_SWITCH_CONFIG_V1 fctGetSwitchConfig;

  TP_FCT_XLAPI_GET_VIRTUAL_PORT_CONFIG_V1
    fctGetVirtualPortConfig;

  TP_FCT_XLAPI_GET_MEASUREMENT_POINT_CONFIG_V1
    fctGetMeasurementPointConfig;

TP_FCT_XLAPI_GET_DLL_CONFIG_V1 fctGetDllConfig;
} XLapiIDriverConfigV1, *pXLapiIDriverConfigV1;
```

Description

XLAPI driver configuration interface structure version 1.

Parameters

- **configHandle**
  Handle for this XLIDriverConfig instance. It is the first argument to the function pointers in this structure.
- **fctGetDeviceConfig**
  Gets the list of local devices, see XLdeviceDrvConfigV1.
- **fctGetChannelConfig**
  Gets the list of local channels, see XLchannelDrvConfigV1.
- **fctGetNetworkConfig**
  Gets the list of networks, see XLnetworkDrvConfigV1.
- **fctGetSwitchConfig**
  Gets the list of switches, see XLswitchDrvConfigV1.
- **fctGetVirtualPortConfig**
  Gets the list of virtual ports, see XLvirtualportDrvConfigV1.
- **fctGetMeasurementPointConfig**
  Gets the list of measurement points, see XLmeasurementpointDrvConfigV1.
- **fctGetDllConfig**
  Gets information on the loaded XL API DLL, see XLdllDrvConfigV1.

3.4.6 XLchannelDrvConfigV1

Syntax

```
typedef struct s_xl_channel_drv_config_v1 {
  unsigned int  hwChannel;
  unsigned int  channelIndex;
  unsigned int  deviceIndex;
  unsigned int  interfaceVersion;
  unsigned int  isOnBus;
  XLuint64      channelCapabilities;
  XLuint64      channelCapabilities2;
  XLuint64      channelBusCapabilities;
  XLuint64      channelBusActiveCapabilities;
  XLuint64      connectedBusType;
  unsigned      intcurrentlyAvailableTimestamps;
  XLbusParams   busParams;

  struct {
    const char*   name;
```

```
    unsigned int  type;
    unsigned int  configError;
  } transceiver;

  const struct s_xl_channel_drv_config_v1 *remoteChannel;
} XLchannelDrvConfigV1, *pXLchannelDrvConfigV1;
```

Description

This structure contains information on a channel of a local or remote connected device.

All channels, that are directly connected to the host, are local channels. The sub devices of a remote host, for example a VN8900 device, have remote channels and local channel counterparts. A remote channel is a channel from the perspective of the remote host. Its local channel counterpart is the same channel but from perspective of the host PC.

Previously connected devices maintain their position in the list even if they are disconnected after the launch of the application. In consequence there might be unused device indexes in the channel list.

Parameters

- **hwChannel**
  Index of this channel relative to its device.
- **channelIndex**
  Index of the local channel in the channel list returned by fctGetChannelConfig. This index uniquely identifies a local channel in the current application. Remote channels have the same `channelIndex` than their local counterpart. The `channelMask` can be computed as `(XLaccess)1 << channelIndex`.
- **deviceIndex**
  Index of the channel's device in the device list returned by fctGetDeviceConfig or `XL_INVALID_DEVICE_INDEX`. The index of remote devices exceeds the item count (number of local devices) set by fctGetDeviceConfig().
- **interfaceVersion**
  Interface version supported by this channel, currently either `XL_INTERFACE_VERSION_V3` or `XL_INTERFACE_VERSION_V4`.
- **isOnBus**
  Is 1 while at least one application activated this channel ("is on bus") and 0 otherwise.
- **channelCapabilities**
  Bitwise combination of the values below. Note the difference between the `XL_CHANNEL_FLAG_EX1` defines and the `XL_CHANNEL_FLAG` defines used in the channelCapablities field of XLchannelConfig.
  - `XL_CHANNEL_FLAG_EX1_TIME_SYNC_RUNNING`
    The channel is synchronized with legacy software time synchronization.
  - `XL_CHANNEL_FLAG_EX1_HWSYNC_SUPPORT`
    The channel supports hardware synchronization, see section XL Sync Pulse on page 71.
  - `XL_CHANNEL_FLAG_EX1_CANFD_ISO_SUPPORT`
    The channel supports ISO-compliant CAN FD operation.
  - `XL_CHANNEL_FLAG_EX1_CANFD_BOSCH_SUPPORT`
    The channel supports the non-ISO CAN FD checksum mode, see also XLcanFdConf.

- XL_CHANNEL_FLAG_EX1_NET_ETH_SUPPORT
  The Ethernet device operates in network-based instead of channel-based mode, see section Switching Access Mode on page 134.
- XL_CHANNEL_FLAG_EX1_TIME_SYNC_SERVICE_PROTOCOL_RUNNING
  A time synchronization protocol from the Vector Time Synchronization Service is running. Use the TimeSync commands.

- **channelCapabilities2**
  For future use.
- **channelBusCapabilities**
  Bitwise combination of XL_BUS_TYPE_* values that defines the set of bus types that the hardware can support on this channel if equipped with an appropriate transceiver.
- **channelBusActiveCapabilities**
  Bitwise combination of XL_BUS_TYPE_* values that the hardware can support on this channel using the currently equipped transceiver. This serves the same purpose as the XL_BUS_ACTIVE_CAP values of the channelBusCapabilities field in XLchannelConfig. Applications that search for channels on which they can open a port of a specific bus type should check channelBusActiveCapabilities.
- **connectedBusType**
  While an application in the system has opened and activated a port on this channel, connectedBusType is the XL_BUS_TYPE_* value for the bustype of the open port. Otherwise this is XL_BUS_TYPE_NONE.
- **currentlyAvailableTimestamps**
  Reserved.
- **busParams**
  Current bus parameters, see section XLbusParams on page 58.
- **transceiver.name**
  Name of the transceiver on this channel, for example “On board CAN 1051cap (Highspeed)” as null-terminated UTF-8 encoded string.
- **transceiver.type**
  The XL_TRANSCEIVER_TYPE_* value for the transceiver on this channel.
- **transceiver.configError**
  Non-zero values indicate problems with the transceiver, for example an unsupported combination of piggies or a transceiver that is not supported on this channel at all.
- **remoteChannel**
  Pointer to the remote channel counterpart of this channel or NULL if there is no remote channel counterpart.

3.4.7 XLdeviceDrvConfigV1

Syntax

```
typedef struct s_xl_device_drv_config_v1 {
  const char* name;
  unsigned int hwType;
  unsigned int hwIndex;
  unsigned int serialNumber;
  unsigned int articleNumber;
  XLuint64     driverVersion;
```

```
  unsigned int connectionInfo;
  unsigned int isRemoteDevice;

  struct {
    const struct s_xl_device_drv_config_v1 *item;
    unsigned int count;
  } remoteDeviceList;

XLchannelDrvConfigListV1 channelList;
} XLdeviceDrvConfigV1, *pXLdeviceDrvConfigV1;
```

Description

This structure contains information on a local or remote connected device.

Parameters

- **name**
  Name of the device, for example “VN1630A”, as null-terminated UTF-8 string.
- **hwType**
  The `XL_HWTYPE_*` value for this device.
- **hwIndex**
  Index to differentiate between multiple connected devices of the same `hwType`.
- **serialNumber**
  Serial number of this device.
- **articleNumber**
  Article number of this device.
- **driverVersion**
  Version of the driver for this device, encoded as `major << 56 | minor << 48 | revision << 32`. The lower 32 bit are reserved.
- **connectionInfo**
  A 32 bit value that specifies the host connection of the device. The value is splitted as follows:
  - `XL_CONNECTION_INFO_FAMILY_MASK`
    The upper 8 bit define the family.
  - `XL_CONNECTION_INFO_DETAIL_MASK`
    The lower 24 bit provide additional information. Interpretation depends on the family.

  Following values for the family are defined:
  - `XL_CONNECTION_INFO_FAMILY_USB`
    The device is either connected via USB or the driver version is older than 10.8.
  - `XL_CONNECTION_INFO_FAMILY_NETWORK`
    The device is connected via the network.
  - `XL_CONNECTION_INFO_FAMILY_PCIE`
    The device is connected via PCI express.

  The detail Information for the USB family reports the speed of the active USB connection:
  - `XL_CONNECTION_INFO_USB_UNKNOWN`
  - `XL_CONNECTION_INFO_USB_FULLSPEED`
  - `XL_CONNECTION_INFO_USB_HIGHSPEED`
  - `XL_CONNECTION_INFO_USB_SUPERSPEED`
- **isRemoteDevice**
  1 if this is a remote device, 0 if it is a local device.

- **remoteDeviceList**
  List of remote devices connected to this device.
- **channelList**
  List of channels that this device provides, see XLchannelDrvConfigV1.



Note
The host connection of VN8900 devices can be detected via the `hwType` `XL_HWTYPE_VN8900` (USB) versus `XL_HWTYPE_IPCLIENT` (network).

The device list contains the Vector Timesync Service (`XL_HWTYPE_VTSSERVICE`) but it is not possible to interact with this pseudo device using the XL API.

3.4.8 XLnetworkDrvConfigV1

Syntax

```
typedef struct s_xl_network_drv_config_v1 {
  const char*             networkName;
  unsigned int            statusCode;
  const char*             statusErrorString;
  XLnetworkType           networkType;
  XLswitchDrvConfigListV1 switchList;
} XLnetworkDrvConfigV1, *pXLnetworkDrvConfigV1;
```

Description

This structure contains information on a network that is configured on at least one connected device (see section Network Based Access Mode on page 136).

Parameters

- **networkName**
  Name of the network as null-terminated UTF-8 string.
- **statusCode**
  A code that describes a configuration error.
  - `XL_NET_CFG_STAT_OK`
    The network is OK and the application can open it.
  - `XL_NET_CFG_DUPLICATE_SEGMENT_NAME`
    The network contains two segments with the same name.
  - `XL_NET_CFG_DUPLICATE_VP_NAME`
    The network contains two virtual ports with the same name.
  - `XL_NET_CFG_DUPLICATE_MP_NAME`
    The network contains two measurement points with the same name.
- **statusErrorString**
  A null-terminated UTF-8 string that contains an English error message that explains the statusCode. In case of a name-duplication, the message contains the duplicate name. If no error message is available, the string is NULL.
- **networkType**
  Currently always `XL_ETH_NETWORK`.
- **switchList**
  List of switches that are part of this network, see XLswitchDrvConfigV1.

3.4.9 XLswitchDrvConfigV1

Syntax

```
typedef struct s_xl_switch_drv_config_v1 {
  const char                      *switchName;
```

```
  XLswitchId switchId;
  unsigned int networkIdx;
  const XLdeviceDrvConfigV1 *device;
  unsigned int switchCapability;
  XLvirtualportDrvConfigListV1 vpList;
  XLmeasurementpointDrvConfigListV1 mpList;
} XLswitchDrvConfigV1, *pXLswitchDrvConfigV1;
```

Description

This structure contains information on a switch that is part of a network. Note that the term “switch” in the driver configuration refers to any kind of segment. For the definition of segment, see section Network Based Access Mode on page 136.

Parameters

- **switchName**
  Name of the switch as null-terminated UTF-8 string.
- **switchId**
  ID of the switch in the network. Switches in different networks may have the same switch ID.
- **networkIdx**
  Index of the network in the network list returned by fctGetNetworkConfig.
- **device**
  Pointer to the device that this switch resides on.
- **switchCapability**
  Defines what kind of segment this “switch” is:
  - `XL_NET_ETH_SWITCH_CAP_REALSWITCH`
    This is a switch segment (MAC address learning is on).
  - `XL_NET_ETH_SWITCH_CAP_DIRECTCONN`
    This segment is a direct connection.
  - `XL_NET_ETH_SWITCH_CAP_TAP_LINK`
    This segment is a tap (test access point).
- **vpList**
  List of virtual ports that are connected to this switch, see XLvirtualportDrvConfigV1. This list includes both statically configured and dynamically added virtual ports.
- **mpList**
  List of measurement points that are connected to this switch, see XLmeasurementpointDrvConfigV1.

3.4.10 XLvirtualportDrvConfigV1

Syntax

```
typedef struct s_xl_virtual_port_drv_config_v1 {
  const char *virtualPortName;
  unsigned int networkIdx;
  XLswitchId switchId;
} XLvirtualportDrvConfigV1, *pXLvirtualportDrvConfigV1;
```

Description

This structure contains information on a virtual port that is connected to a switch.

Parameters

- **virtualPortName**
  Name of the virtual port as null-terminated UTF-8 string.
- **networkIdx**
  Index of the network in the network list returned by fctGetNetworkConfig.

- **switchId**
  ID of the switch that this virtual port is connected to.

3.4.11 XLmeasurementpointDrvConfigV1

Syntax

```
typedef struct s_xl_measurement_point_drv_config_v1 {
  const char *measurementPointName;
  unsigned int networkIdx;
  XLswitchId switchId;
  const XLchannelDrvConfigV1 *channel;
} XLmeasurementpointDrvConfigV1, *pXLmeasurementpointDrvConfigV1;
```

Description

This structure contains information on a measurement point that is connected to a switch.

Parameters

- **measurementPointName**
  Name of the measurement point as null-terminated UTF-8 string.
- **networkIdx**
  Index of the network in the network list returned by fctGetNetworkConfig.
- **switchId**
  ID of the switch that this measurement point is connected to.
- **channel**
  Pointer to the channel associated with the measurement point or NULL if no such channel exists.

3.4.12 XLdllDrvConfigV1

Syntax

```
typedef struct s_xl_dll_drv_config_v1 {
  XLuint64 dllVersion;
} XLdllDrvConfigV1, *pXLdllDrvConfigV1;
```

Description

This structure contains information on the `vxlapi.dll/vxlapi64.dll` instance loaded by this application.

Parameters

- **dllVersion**
  Version of the DLL, encoded as `major << 56 | minor << 48 | revision << 32`. The lower 32 bit are reserved.

3.5 Events

3.5.1 XLevent

Syntax

```
struct s_xl_event {
  XLeventTag          tag;
  unsigned char       chanIndex;
  unsigned short      transId;
  unsigned short      portHandle;
  unsigned char       flags;
  unsigned char       reserved;
  XLuint64            timeStamp;
  union s_xl_tag_data tagData;
};
```

Parameters

- **tag**
  Common and CAN events
  `XL_RECEIVE_MSG`
  `XL_CHIP_STATE`
  `XL_TRANSCEIVER`
  `XL_TIMER`
  `XL_TRANSMIT_MSG`
  `XL_SYNC_PULSE`

  Special LIN events
  `XL_LIN_MSG`
  `XL_LIN_ERRMSG`
  `XL_LIN_SYNCERR`
  `XL_LIN_NOANS`
  `XL_LIN_WAKEUP`
  `XL_LIN_SLEEP`
  `XL_LIN_CRCINFO`

  All K-Line events
  `XL_KLINE_MSG`

  Special DAIO events
  `XL_RECEIVE_DAIO_DATA`
- **chanIndex**
  Channel on which the event occurs.
- **transId**
  Internal use only.
- **portHandle**
  Internal use only.
- **flags**
  e. g. `XL_EVENT_FLAG_OVERRUN`
- **reserved**
  Reserved for future use. Set to 0.
- **time stamp**
  Actual time stamp generated by the hardware with 8 µs resolution. Value is in nanoseconds.

- **tagData**
  Union for the different events.

3.5.2 XL Tag Data

Syntax

```
union s_xl_tag_data {
  struct s_xl_can_msg         msg;
  struct s_xl_chip_state      chipState;
  union s_xl_lin_msg_api      linMsgApi;
  struct s_xl_sync_pulse      syncPulse;
  struct s_xl_transceiver     transceiver;
  struct s_xl_daio_data daioData;
  struct s_xl_daio_piggy_data daioPiggyData;
  struct s_xl_kline_data      klineData;
};
```

Parameters

- **msg**
  Union for all CAN events.
- **chipState**
  Structure for all CHIPSTATE events.
- **linMsgApi**
  Union for all LIN events.
- **syncPulse**
  Structure for all SYNC_PULSE events.
- **transceiver**
  Structure for all TRANSCEIVER events.
- **daioData**
  Structure for all DAIO data.
- **daioPiggyData**
  Structure for all DAIO Piggy data.
- **klineData**
  Structure for all K-Line events.

3.5.3 XL Sync Pulse

Syntax

```
struct s_xl_sync_pulse {
  unsigned char pulseCode;
  XLuint64      time;
} XL_SYNC_PULSE_EV;
```

Description

This event is generated on all channels of the device when a sync pulse is received. A sync pulse can be triggered by xlGenerateSyncPulse().

Use the `timeStamp` element of the general event structure for time calculation. The structure element `time` is reserved and shall not be used on devices other than the XL Family.

Tag

`XL_SYNC_PULSE` (see section XLevent on page 70).

Parameters

- **pulseCode**
  `XL_SYNC_PULSE_EXTERNAL`
  The sync event comes from an external device.

  `XL_SYNC_PULSE_OUR`
  The sync pulse event occurs after an xlGenerateSyncPulse().

  `XL_SYNC_PULSE_OUR_SHARED`
  The sync pulse comes from the same hardware but from another channel.
- **time**
  This element is only used in XL Family devices. It is not used for all other Vector devices.

3.5.4 XL Transceiver

Syntax

```
struct s_xl_transceiver {
  unsigned char event_reason;
  unsigned char is_present;
};
```

Tag

`XL_TRANSCEIVER` (see section XLevent on page 70).

Parameters

- **event_reason**
  Reason for occurred event.
- **is_present**
  Always valid transceiver.

3.5.5 XL Timer

Description

A timer event can be generated cyclically by the driver to keep the application alive. The timer event occurs after initialization with xlSetTimerRate().

Tag

`XL_TIMER` (see section XLevent on page 70).

4 TimeSync Commands

In this chapter you find the following information:

4.1 Introduction **74**
4.2 Flowchart Time Sync Status (Channel Based) **77**
4.3 Flowchart Time Sync Status (Network Based) **78**
4.4 Flowchart Application Clock **79**
4.5 Application Examples **80**
4.6 Functions **83**
4.7 Structs **88**

4.1 Introduction

4.1.1 General Information

The Vector Time Synchronization Service is part of the driver installation package and is installed on the system together with the Vector device drivers. The service provides several synchronization methods. The XL Driver Library allows applications to access the features of the Vector Time Synchronization Service and provides functions to interact with the time synchronization mechanism, e. g. you can perform status queries or clock resets.

4.1.2 Time Sync Basics

Synchronization of Vector devices

A synchronization protocol is a method for synchronizing the clocks on Vector devices. Each Vector device implements its own local clock for event generation. A clock always starts at zero seconds after a cold start if no synchronization is present.

Clocks on Vector devices can have different timescales. A timescale describes the type of time reference (source) of each clock (see section XltsTimeScale on page 88). For example, if the time reference is another free-running Vector device, the timescale is **Arbitrary**.

Precision

Synchronization protocols can be combined and have different precisions; remember to use high-precision synchronization protocols as a source for lower-precision synchronization protocols. For more information on synchronization protocols, refer to the device manuals.

Application clock

The Vector Time Synchronization Service provides its own clock (application clock). This application clock is independent of ports or networks. The application clock can be synchronized in two different ways:

- via software sync (e. g. by a Vector device that provides software sync)
- via external PTP timeTransmitter (within Ethernet networks)

This has to be set up in the Vector Hardware Manager.

See also section Flowchart Application Clock on page 79 to learn how to use the clock and its synchronized time in your own application.

4.1.3 Time Sync Example

Clocks

The following setup consists of three different Vector devices, each with its own (hardware) clock. In addition, there is the Vector Time Sync Service with its application clock (embedded in the XL Default Time Domain). All (hardware) clocks are synchronized with each other via the synchronization configured in the Vector Hardware Manager. In this example, device A synchronizes all other devices. The application clock uses the available software synchronization sent by device C.

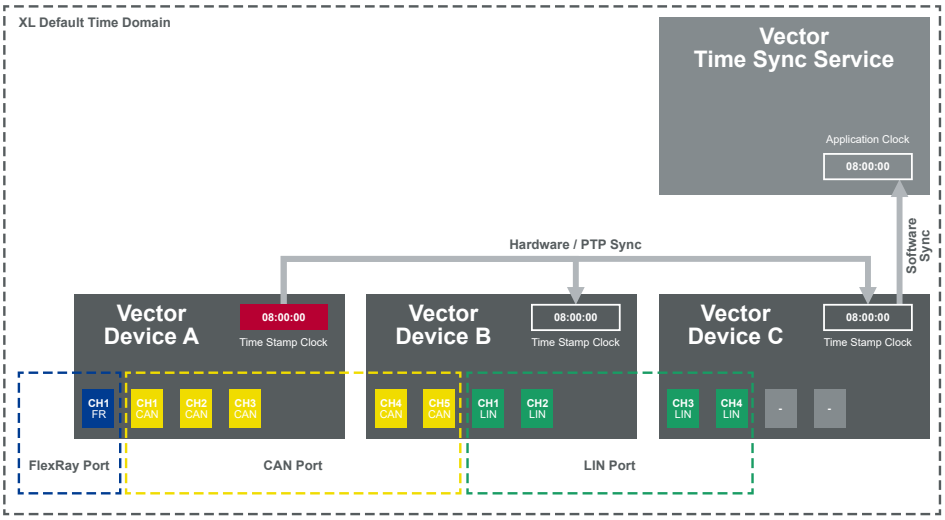


Figure 7: Setup example

Note

The XL API time sync functions are bound to port handles. In order to use the time sync functions, you must have already opened a CAN, a LIN and a FlexRay port to get the corresponding port handles.

Also note that a port is created for all available Vector devices that support a requested bus type, not just for a single device. For example, the CAN port includes the CAN channels of both device A and device B.

Time stamps

After opening the three corresponding ports and initializing the application clock, four time axes are available for the bus events. The bus events provide the time stamps. Without resetting the time sync clock of the ports, the time stamps are generated with respect to the current (synchronized) device time.

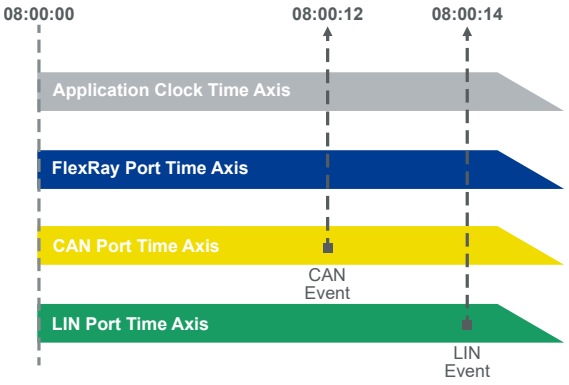


Figure 8: Time stamps from current device time

Resetting clocks

Resetting the clock shifts the event time stamp offset to a specific time, such as 0. This is especially useful if you want to start a measurement at zero seconds. Each bus type can be reset separately at any time.

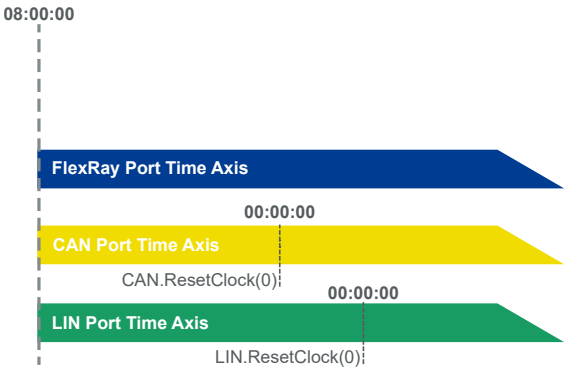


Figure 9: Device time no longer valid after reset

The following figure shows that the resets for CAN and LIN result in different time axes for each port. This can lead to unwanted effects.

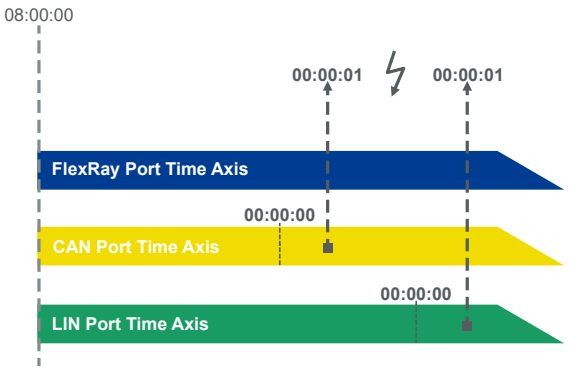


Figure 10: Identical time stamps occur

A common use case would be to reset all ports to use the same time axes. This is achieved by first resetting one clock to 0 (e. g. CAN) and then resetting the remaining clocks of the other ports.

Note

If you reset a clock to 0, the corresponding offset value is immediately written back in the function call. You can use this value for further resets.

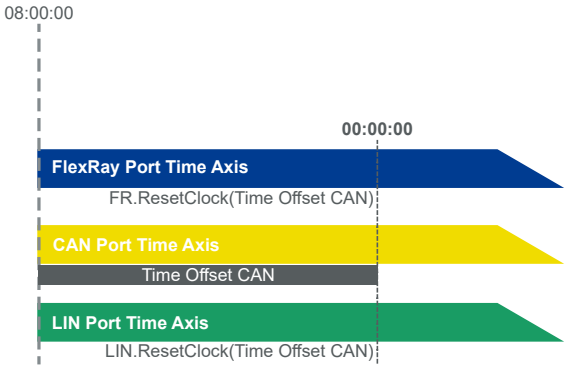


Figure 11: Aligned reset

4.2 Flowchart Time Sync Status (Channel Based)

Calling sequence

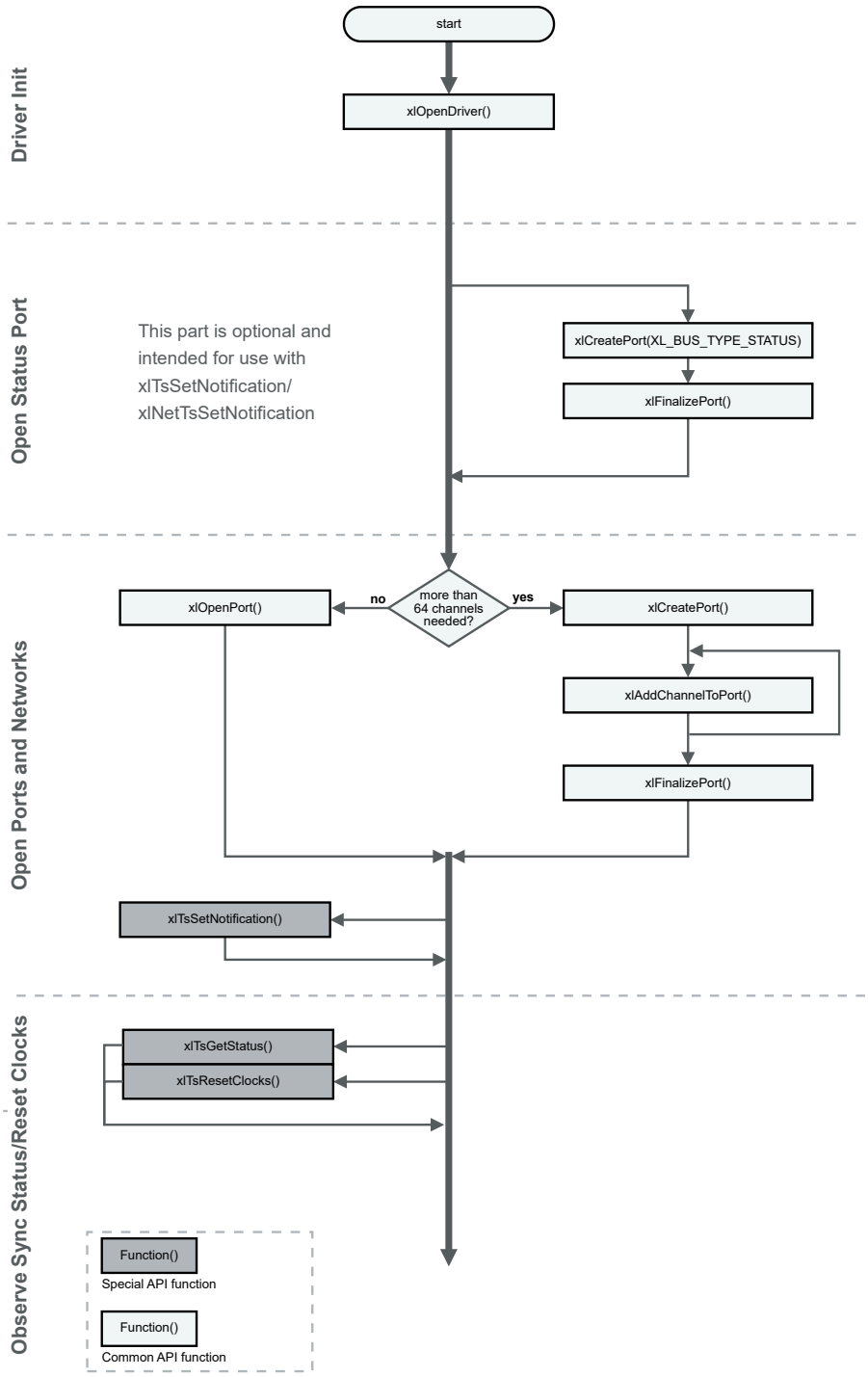


Figure 12: Function calls

4.3 Flowchart Time Sync Status (Network Based)

Calling sequence

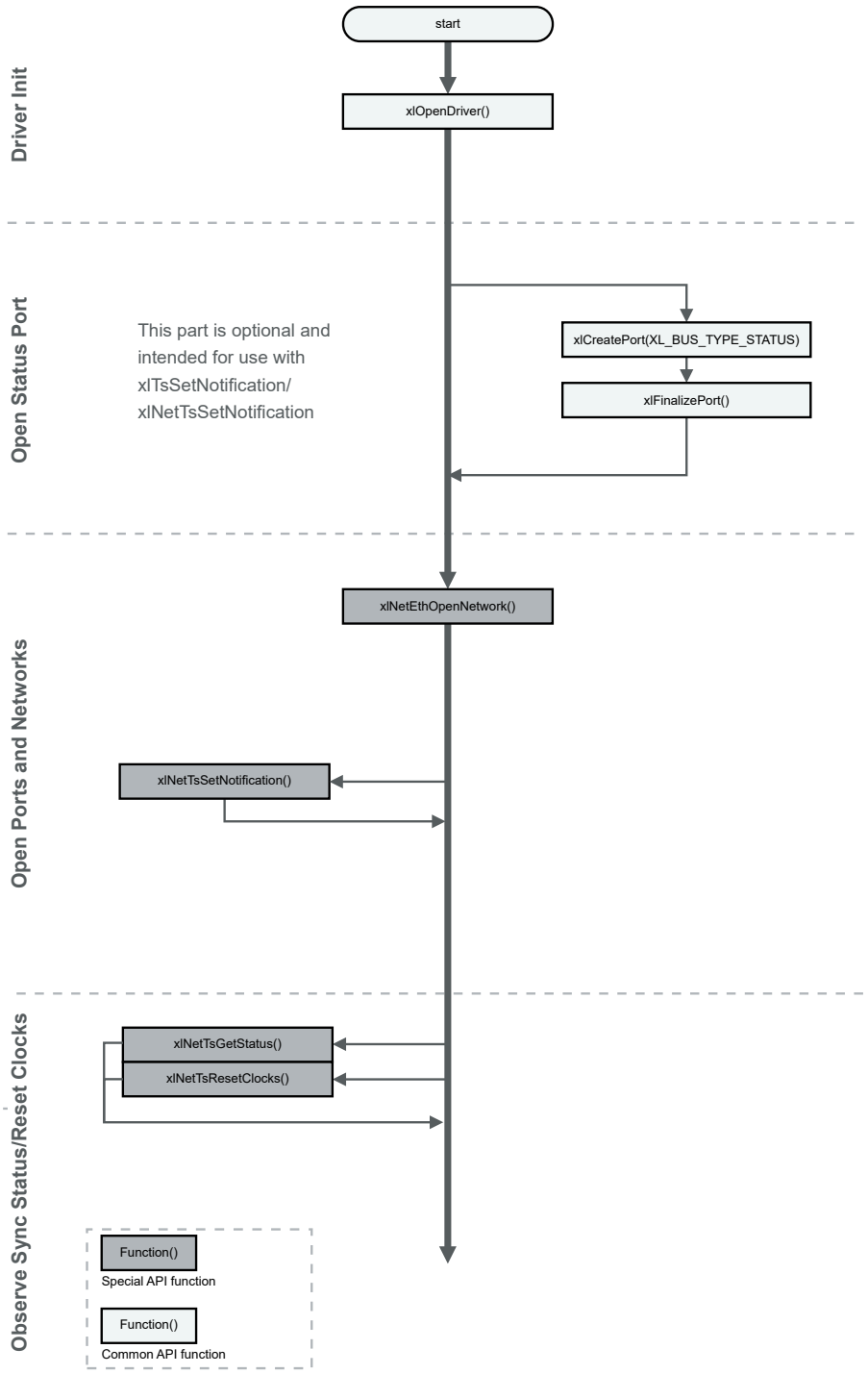


Figure 13: Function calls

4.4 Flowchart Application Clock

Calling sequence

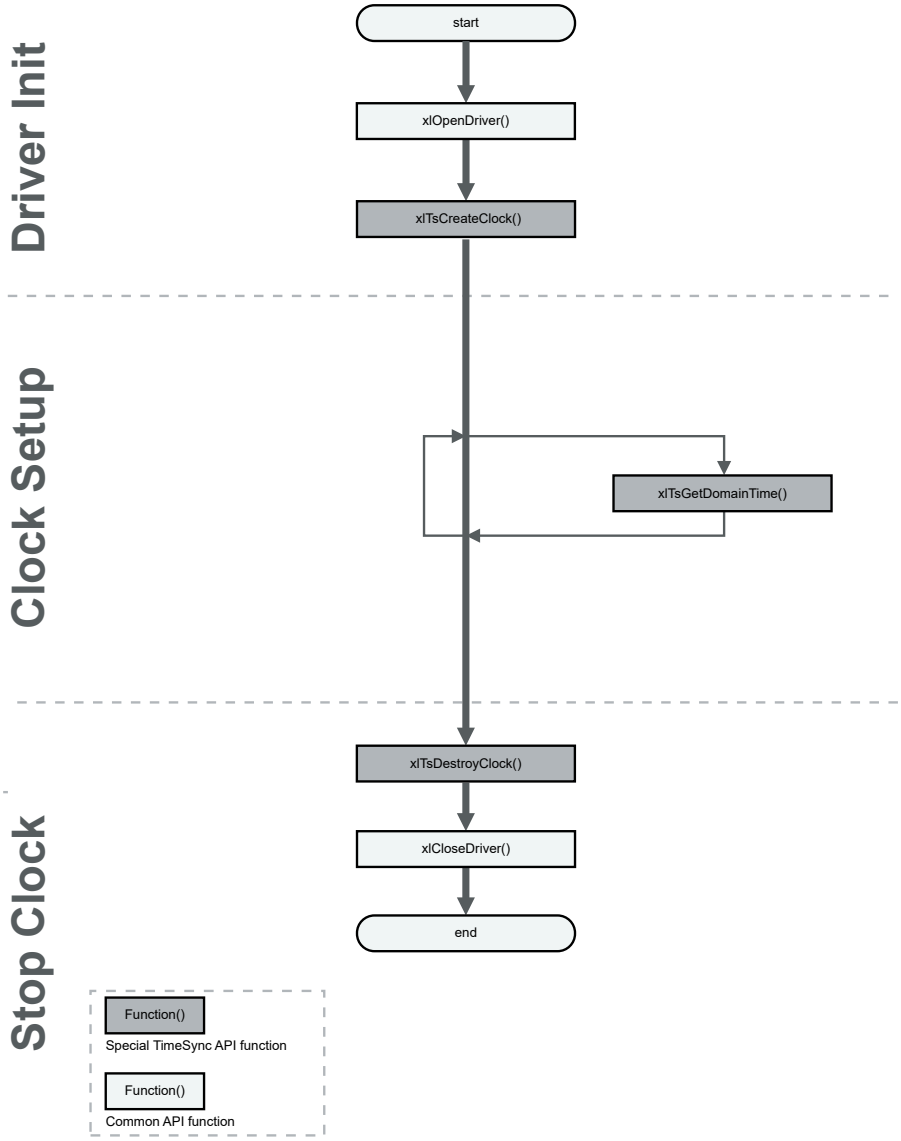


Figure 14: Function calls for application clock

4.5 Application Examples

4.5.1 xlVTssDemo

4.5.1.1 General Information

Description

This example demonstrates the basic handling of time sync commands. xlVTssDemo shows how to perform status queries or clock resets on clocks. The application fetches available CAN channels or Ethernet networks for its actions. See a possible Vector Hardware Manager configuration in the images below.

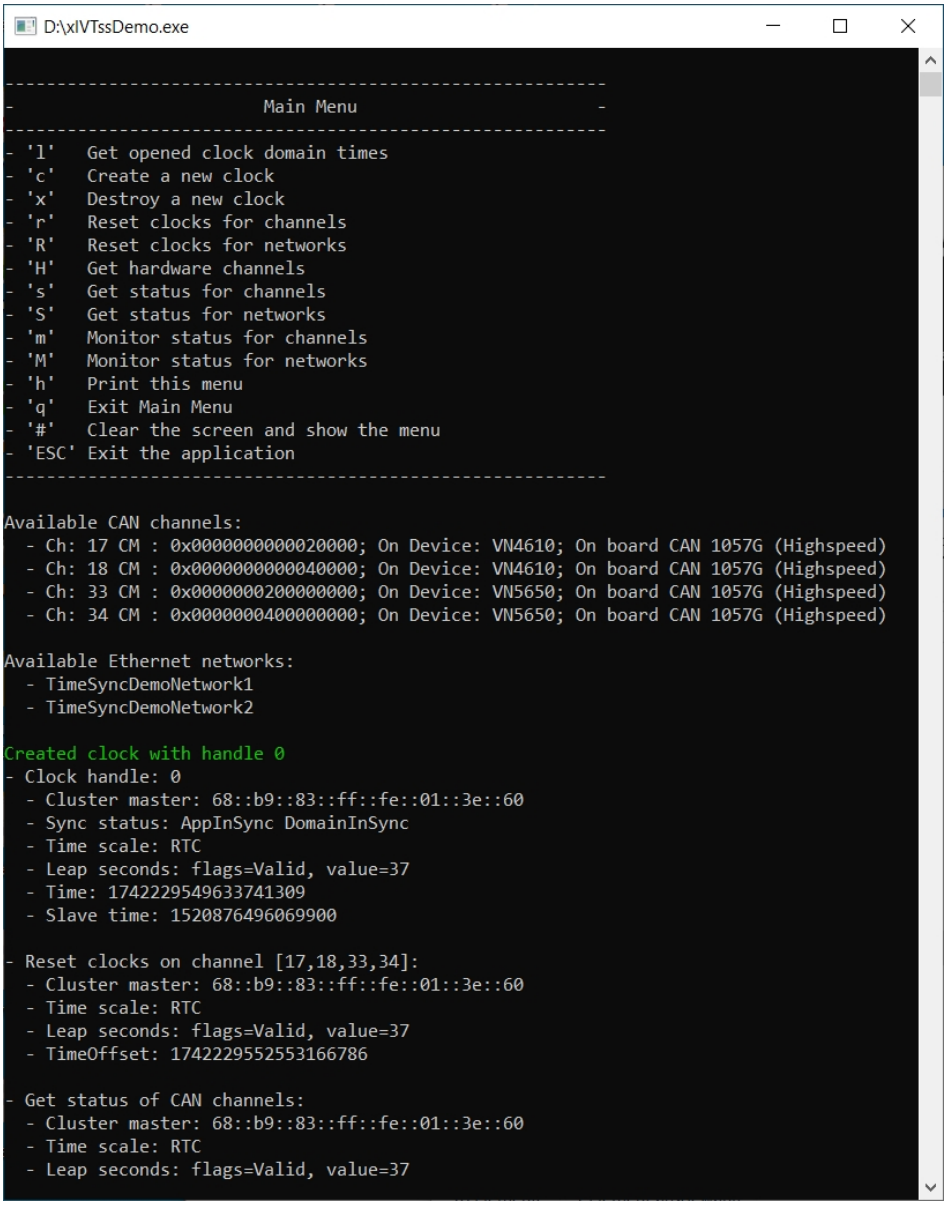


Figure 15: Running xlVTssDemo

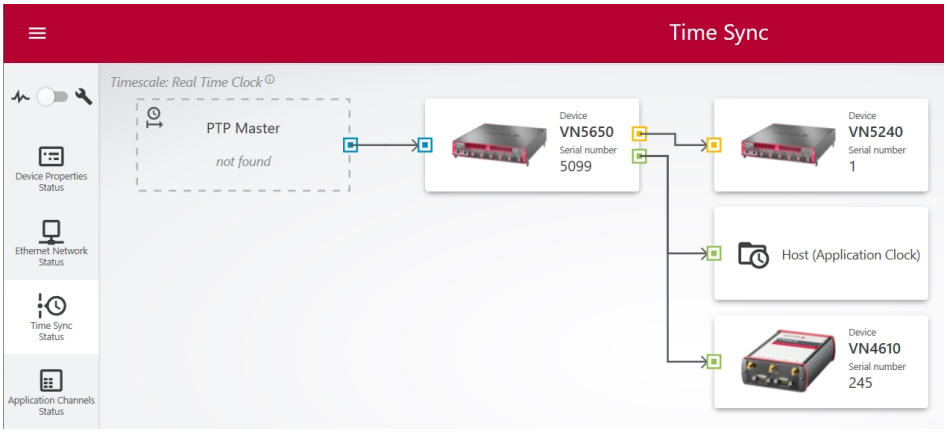


Figure 16: Vector Hardware Manager time sync config (example)

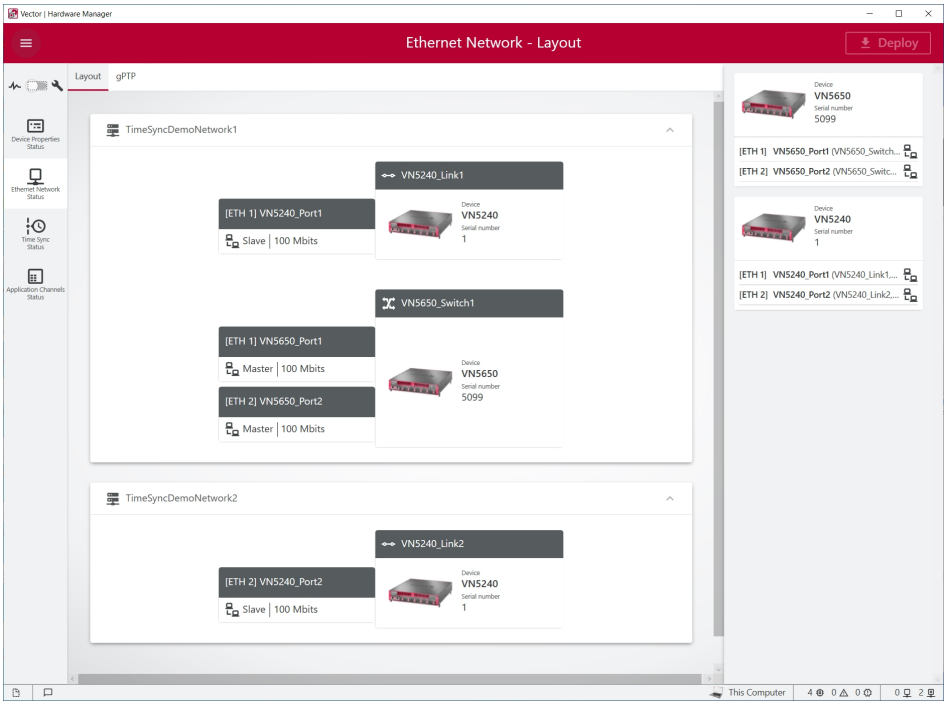


Figure 17: Vector Hardware Manager network config (example)

4.5.1.2 Keyboard Commands

The running application can be controlled via the following keyboard commands:

Key	Command
<l>	Get opened clock domain times
<c>	Create a new clock
<x>	Destroy new clock
<r>	Reset clocks for channels
<R>	Reset clocks for networks
<H>	Get hardware channels
<s>	Get status for channels
<S>	Get status for networks
<m>	Monitor status for channels
<M>	Get status for networks
<h>	Monitor status for channels

Key	Command
<q>	Monitor status for networks
<#>	Print this menu
<ESC>	Exit the application

4.6 Functions

4.6.1 xlTsCreateClock

Syntax

```
XLstatus xlTsCreateClock (
  XLtsClockHandle *pClockHandle,
  const char *timeDomainName,
  XLtsClkExternalType clkType,
  XLtsInterfaceVersion apiVersion)
```

Description

Allocates a new clock context or opens an existing clock. This function is called to open an existing domain clock and get the current time.

Input parameters

- **timeDomainName**
  The name of the time domain to which the allocated clock context should be bound. Supported names:
  - `XL_TS_DEFAULT_TIMEDOMAIN_NAME`.
- **clkType**
  The type of clock to allocate or open, defined by `XLtsClkExternalType`. Supported types:
  - `XL_TS_CLK_EXTTYPE_DOMAIN`
- **apiVersion**
  The version of the TimeSync API used by the application. See also `XLtsInterfaceVersion`. Supported versions:
  - `XL_TS_INTERFACE_VERSION_1`

Output parameters

- **pClockHandle**
  Handle to access the new clock context.

Return value

Returns an error code (see section Error Codes on page 360).

4.6.2 xlTsDestroyClock

Syntax

```
XLstatus xlTsDestroyClock (
  XLtsClockHandle clockHandle)
```

Description

Destroys the clock context that was allocated with xlTsCreateClock(). The context becomes invalid.

Input parameters

- **pClockHandle**
  The handle retrieved by xlTsCreateClock().

Return value

Returns an error code (see section Error Codes on page 360).

4.6.3 xlTsGetDomainTime

Syntax

```
XLstatus xlTsGetDomainTime (
  XLtsClockHandle clockHandle,
  XLtsDomainTime *domainTime,
  XLuint64 *slaveTime)
```

Description

Reads the current time from the Vector Time Synchronization Service internal soft-

ware synchronized clock for this domain. The timestamp is not taken directly from the network interface.

The cluster master must be unique or all network interfaces in the domain must have the same absolute time scale (e. g. TAI). One of the interfaces connected to the computer must be of uplink type USB or PCIe, or the PTP client of the computer must be enabled.

This API function replaces xlGetSyncTime() and xlGetChannelTime().

Input parameters

- **pClockHandle**
  The handle retrieved by xlTsCreateClock().
- **domainTime**
  Converted 64-bit timestamp of the domain clock in nanoseconds and the synchronization status of the domain clock.
- **slaveTime**
  64-bit time stamp of the local computer's performance counter taken in Vector Time Synchronization Service.

Return value

Returns an error code (see section Error Codes on page 360).

4.6.4 xlTsResetClocks

Syntax

```
XLstatus xlTsResetClocks (
  XLportHandle    portHandle,
  XLtsTimeScale   *timescale,
  XLtsLeapSeconds *leapSeconds,
  XLtsClkUuid     *gmUuid,
  XLuint64        *timeOffset)
```

Description

Checks the synchronization status of the channels and reset the time stamps. The channel is part of the port. A channel is said to be in synchronous state when the device's measurement clock is in synchronous state.

The entire port is in synchronous state when:

- The timescale is arbitrary, all channels are in synchronous state and all clocks use the same grandmaster's reference time.
- The timescale is TAI, all channels and networks are in synchronous state and all participants have the same leap second information.
- All synchronization protocols are disabled.
- The cluster master is a GNSS synchronized device without time fix.

Input parameters

- **portHandle**
  The port handle retrieved by xlCreatePort() or xlOpenPort().

Output parameters

- **timescale**
  The actual timescale of the channel timestamps.
- **leapSeconds**
  If timescale equals TAI, these are the leap seconds on the protocol.
- **gmUuid**
  This is the XL API clock identifier of the grandmaster clock. When receiving time from GNSS and the receiving device is a Vector device, this is the identifier of the device's local clock.

Input/output parameters

- **timeOffset**
  If this function is called with `*timeOffset=0`, the driver resets the clocks and returns the offset of the channel time stamp.If `*timeOffset!=0`, the driver uses this value as time offset for the channels. This mode of the function can be used to re-synchronize a channel to an existing time domain (e .g. after reconnecting a device).

Return value

Returns an error code (see section Error Codes on page 360).

4.6.5 xlNetTsResetClocks

Syntax

```
XLstatus xlNetTsResetClocks (
  XLnetworkHandle networkHandle,
  XLtsTimeScale   *timescale,
  XLtsLeapSeconds *leapSeconds,
  XLtsClkUuid     *gmUuid,
  XLuint64        *timeOffset)
```

Description

Similar to xlTsResetClocks() but for use with network handles.

Input parameters

- **networkHandle**
  The handle retrieved by xlNetEthOpenNetwork().

For more parameters see xlTsResetClocks().

Return value

Returns an error code (see section Error Codes on page 360).

4.6.6 xlTsSetNotification

Syntax

```
XLstatus xlTsSetNotification (
  XLportHandle statusPortHandle,
  XLportHandle portHandle,
  XLhandle     *pHandle)
```

Description

Sets up an event to notify the application when the synchronization status of a clock associated with this port changes.

Input parameters

- **statusPortHandle**
  The status port handle retrieved by xlCreatePort() with type `XL_BUS_TYPE_STATUS`.
- **portHandle**
  The port handle retrieved by xlCreatePort() or xlOpenPort().

Output parameters

- **pHandle**
  Handle for the created event object.

Return value

Returns an error code (see section Error Codes on page 360).

4.6.7 xlNetTsSetNotification

Syntax

```
XLstatus xlNetTsSetNotification (
  XLportHandle     statusPortHandle,
  XLnetworkHandle networkHandle,
```

```
   XLhandle        *pHandle)
```

Description

Sets up an event to notify the application when the synchronization status of a clock related to this port changes. When using the notification event, first register the event, then call xlTsGetStatus().

Input parameters

- **statusPortHandle**
  The status port handle retrieved by xlCreatePort() with type `XL_BUS_TYPE_STATUS`.
- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().

Output parameters

- **pHandle**
  Handle for the created event object.

Return value

Returns an error code (see section Error Codes on page 360).

4.6.8 xlTsGetStatus

Syntax

```
XLstatus xlTsGetStatus (
XLportHandle    portHandle,
XLaccess        channelSelector,
XLtsTimeScale   *timescale,
XLtsLeapSeconds *leapSeconds,
XLtsClkUuid      *gmUuid,
XLtsClkUuid     *localUuid)
```

Description

Checks the synchronization status of the channels and reset the time stamps. The channel is part of the port. A channel is said to be in synchronous state when the device's measurement clock is in synchronous state.

The entire port is in synchronous state when:

- The timescale is arbitrary, all channels are in synchronous state and all clocks use the same grandmaster's reference time.
- The timescale is TAI, all channels and networks are in synchronous state and all participants have the same leap second information.
- All synchronization protocols are disabled.
- The cluster master is a GNSS synchronized device without time fix.

Input parameters

- **portHandle**
  The port handle retrieved by xlCreatePort() or xlOpenPort().
- **channelSelector**
  Selects the channel for which to get the time synchronization status. Use `XL_USE_ALL_CHANNELS` to get the time synchronization status of the entire port.

Output parameters

- **timescale**
  The actual timescale of the channel timestamps.
- **leapSeconds**
  If timescale equals TAI, these are the leap seconds on the protocol.
- **gmUuid**
  This is the XL API clock identifier of the grandmaster clock. When receiving time from GNSS and the receiving device is a Vector device, this is the identifier of the device's local clock.

- **localUuid**
  The unique time source identifier for the channel when `channelSelector != XL_USE_ALL_CHANNELS`.
  This parameter is optional and can be set to null.

Return value — Returns an error code (see section Error Codes on page 360).

4.6.9 xlNetTsGetStatus

Syntax

```
XLstatus xlNetTsGetStatus (
  XLnetworkHandle networkHandle,
  XLethPortHandle ethPortHandle,
  XLtsTimeScale   *timescale,
  XLtsLeapSeconds *leapSeconds,
  XLtsClkUuid     *gmUuid,
  XLtsClkUuid     *localUuid)
```

Description — Similar to xlTsGetStatus() but for use with network handles.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **ethPortHandle**
  Reserved for future use. Set to `XL_INVALID_ETHPORTHANDLE`.

For more parameters see xlTsGetStatus().

Return value — Returns an error code (see section Error Codes on page 360).

4.7 Structs

4.7.1 XLtsClkUuid

Syntax

```
struct s_xl_timesync_clock_uuid_ {

  XLtsClkUuidFormat uuidFormat;

  union {
    struct {
      unsigned int articleNumber;
      unsigned int serialNumber;
      unsigned int clkId;
    } vectorDevUuid;

    struct {
      unsigned char oui[3];
      unsigned char extensionId[5];
    } eui64Uuid;

      unsigned int raw[6];
  } uuid;

  unsigned int reserved[4];

} XLtsClkUuid;
```

Parameters

- **uuidFormat**
  Specifies which identifier format is used.
- **vectorDevUuid**
  Identifier is composed of the article and serial number of the Vector device and an index per device of the clock. This format is used when PTP is disabled.
- **eui64Uuid**
  The 64-bit extended unique identifier according to the IEC 61588:2009 specification.

4.7.2 XltsTimeScale

Syntax

```
enum e_xl_timesync_time_scale {
  XL_TS_TIMESCALE_UNDEFINED,
  XL_TS_TIMESCALE_UTC,
  XL_TS_TIMESCALE_TAI,
  XL_TS_TIMESCALE_RTC,
  XL_TS_TIMESCALE_PERFORMANCE_COUNTER,
  XL_TS_TIMESCALE_ARBITRARY
} XLtsTimeScale;
```

Parameters

- **XL_TS_TIMESCALE_UNDEFINED**
  The time scale of this clock or domain is undefined due to synchronization errors or ambiguous time scales in the domain.
- **XL_TS_TIMESCALE_UTC**
  The time scale for event time stamps is UTC. The 64-bit time stamp counts the number of nanoseconds since 01-01-1970, including leap seconds.
- **XL_TS_TIMESCALE_TAI**
  Timescale is TAI according to IEC 61588:2009 specification.

- **XL_TS_TIMESCALE_RTC**
  Like TAI, but the clock source is RTC and the second is not SI.
- **XL_TS_TIMESCALE_PERFORMANCE_COUNTER**
  Performance counter of the local computer. Start time of the computer.
- **XL_TS_TIMESCALE_ARBITRARY**
  Time scale of a free-running clock on a Vector device.

4.7.3 XLtsLeapSeconds

Syntax

```
struct s_xl_timesync_leap_seconds {
  unsigned int leapSecondsFlags;
  int          leapSecondsValue;
} XLtsLeapSeconds;
```

Parameters

- **leapSecondsFlags**
  - XL_TS_LEAP_SECONDS_FLAGS_VALID
- **leapSecondsValue**
  Leap seconds.

5 CAN Commands

In this chapter you find the following information:

5.1 Introduction **91**
5.2 Flowchart **92**
5.3 Functions **93**
5.4 Structs **104**
5.5 Events **105**
5.6 Application Examples **108**

5.1 Introduction

Description

The **XL Driver Library** enables the development of CAN applications for supported Vector devices (see section System Requirements on page 25). Multiple CAN applications can use a common physical CAN channel at the same time.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel parameters can be changed/configured
- CAN messages can be transmitted on the channel
- CAN messages can be received on the channel

Without init access

- CAN messages can be transmitted on the channel
- CAN messages can be received on the channel



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

5.2 Flowchart

Calling sequence

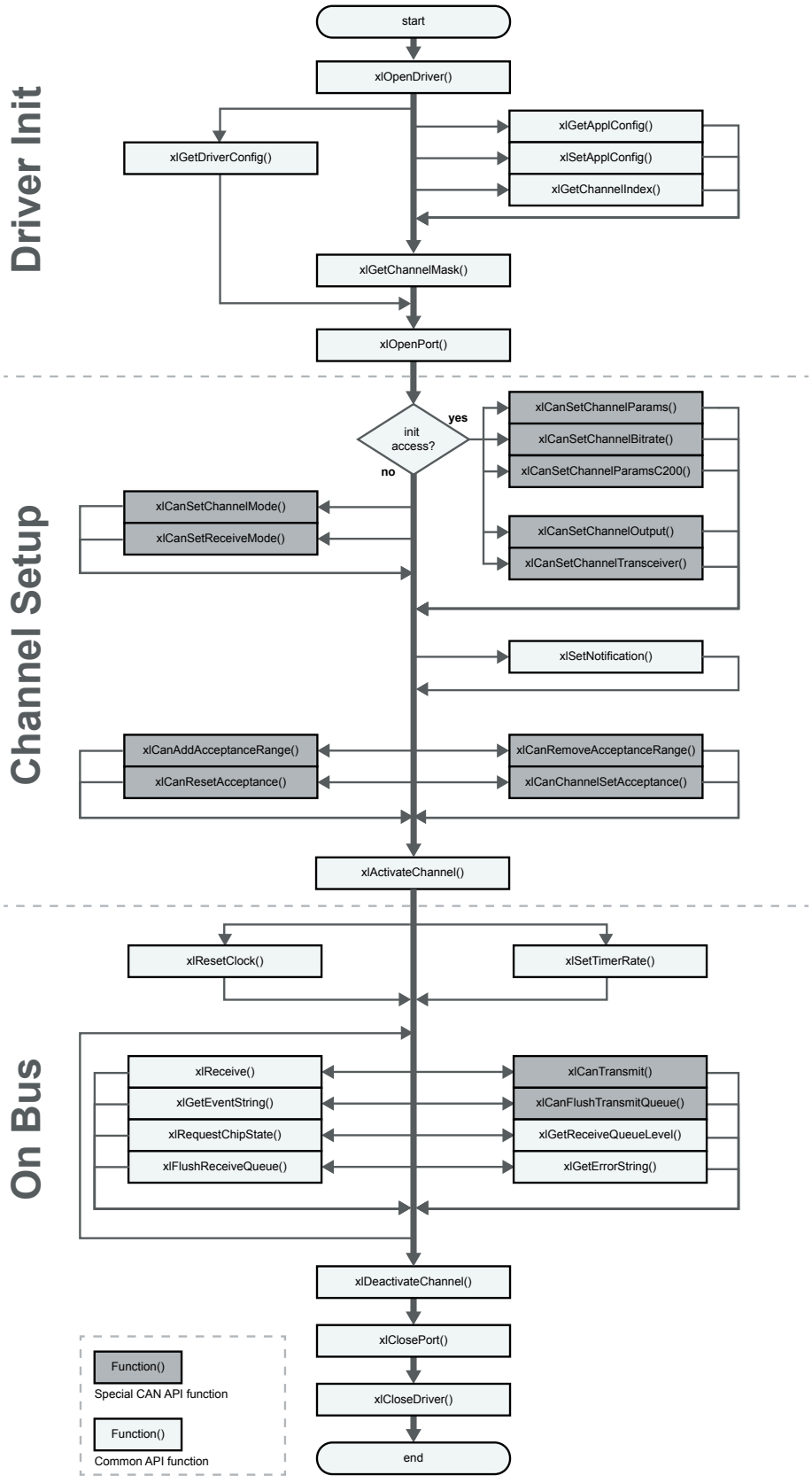


Figure 18: Function calls for CAN applications

5.3 Functions

5.3.1 xlCanSetChannelMode

Syntax

```
Xlstatus xlCanSetChannelMode (
  XLportHandle portHandle,
  XLaccess     accessMask,
  int          tx,
  int txrq)
```

Description

This function specifies whether the caller will get a Tx and/or a TxRq receipt for transmitted messages (for CAN channels defined by `accessMask`). The default is TxRq deactivated and Tx activated.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **tx**
  A flag specifying whether the channel should generate receipts when a message is transmitted by the CAN chip.
  - '1' = generate receipts
  - '0' = deactivated.
  Sets the `XL_CAN_MSG_FLAG_TX_COMPLETED` flag.
- **txrq**
  A flag specifying whether the channel should generate receipts when a message is ready for transmission by the CAN chip.
  - '1' = generate receipts,
  - '0' = deactivated.
  Sets the `XL_CAN_MSG_FLAG_TX_REQUEST` flag.

Return value

Returns an error code (see section Error Codes on page 360).

5.3.2 xlCanSetChannelOutput

Syntax

```
Xlstatus xlCanSetChannelOutput (
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned char mode)
```

Description

If mode is `XL_OUTPUT_MODE_SILENT` the CAN chip will not generate any acknowledges when a CAN message is received. It is not possible to transmit messages, but they can be received in the silent mode. Normal mode is the default mode if this function is not called.



Note
To call this function, the port must have **init access** (see section xlOpenPort on page 38) for the specified channels, and the channels must be deactivated.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **mode**
  Specifies the output mode of the CAN chip.

  `XL_OUTPUT_MODE_SILENT`
  No acknowledge will be generated on receive (silent mode).

  Note: With driver version V5.5, the silent mode has been changed. The Tx pin is switched off now (the 'SJA1000 silent mode' is not used anymore).

  `XL_OUTPUT_MODE_NORMAL`
  Acknowledge (normal mode)

Return value

Returns an error code (see section Error Codes on page 360).

5.3.3 xlCanSetReceiveMode

Syntax

```
XLstatus xlCanSetReceiveMode (
  XLportHandle  Port,
  unsigned char ErrorFrame,
  unsigned char ChipState)
```

Description

Suppresses error frames and chipstate events with '1', but allows those with '0'. Error frames and chipstate events are allowed by default.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **ErrorFrame**
  Suppresses error frames.
- **ChipState**
  Suppresses chipstate events.

Return value

Returns an error code (see section Error Codes on page 360).

5.3.4 xlCanSetChannelTransceiver

Syntax

```
XLstatus xlCanSetChannelTransceiver(
  XLportHandle portHandle,
  XLaccess     accessMask,
  int          type,
  int          lineMode,
  int          resNet)
```

Description

This function is used to set the transceiver modes. The possible transceiver modes depend on the transceiver type connected to the hardware. The port must have **init access** (see section xlOpenPort on page 38) to the channels.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **type**
  Lowspeed (252/1053/1054)
  `XL_TRANSCEIVER_TYPE_CAN_252`

  Highspeed (1041 and 1041opto)
  `XL_TRANSCEIVER_TYPE_CAN_1041`
  `XL_TRANSCEIVER_TYPE_CAN_1041_opto`

  Single Wire (AU5790)
  `XL_TRANSCEIVER_TYPE_CAN_SWC`
  `XL_TRANSCEIVER_TYPE_CAN_SWC_OPTO`
  `XL_TRANSCEIVER_TYPE_CAN_SWC_PROTO`

  Truck & Trailer
  `XL_TRANSCEIVER_TYPE_CAN_B10011S`
  `XL_TRANSCEIVER_TYPE_PB_CAN_TT_OPTO`



Reference
Find further definitions in the header file `vxlapi.h`.

- **lineMode**
  Lowspeed (252/1053/1054)
  `XL_TRANSCEIVER_LINEMODE_SLEEP`
  Puts CANcab into sleep mode.

  `XL_TRANSCEIVER_LINEMODE_NORMAL`
  Enables normal operation.

  Highspeed (1041 and 1041opto)
  `XL_TRANSCEIVER_LINEMODE_SLEEP`
  Puts CANcab into sleep mode.

  `XL_TRANSCEIVER_LINEMODE_NORMAL`
  Enables normal operation.

  Single Wire (AU5790)
  `XL_TRANSCEIVER_LINEMODE_SWC_WAKEUP`
  Enables the sending of high voltage messages (used to wake up sleeping nodes on the bus).

  `XL_TRANSCEIVER_LINEMODE_SWC_SLEEP`
  Switches to sleep mode.

  `XL_TRANSCEIVER_LINEMODE_SWC_NORMAL`
  Switches to normal operation.

  `XL_TRANSCEIVER_LINEMODE_SWC_FAST`
  Switches transceiver to fast mode.

  Truck & Trailer
  `XL_TRANSCEIVER_LINEMODE_NORMAL`
  Normal operation on CAN High and CAN Low.

  `XL_TRANSCEIVER_LINEMODE_TT_CAN_H`
  Switches the transceiver to one-wire-mode on CAN High.

  `XL_TRANSCEIVER_LINEMODE_TT_CAN_L`
  Switches the transceiver to one-wire-mode on CAN Low.

- **resNet**
  Reserved for future use. Set to 0.

Return value — Returns an error code (see section Error Codes on page 360).

5.3.5 xlCanSetChannelParams

Syntax

```
XLstatus xlCanSetChannelParams (
  XLportHandle  portHandle,
  XLaccess      accessMask,
  XLchipParams  *pChipParams)
```

Description — This function initializes the channels defined by `accessMask` with the given parameters. In order to call this function the port must have **init access** (see section xlOpenPort on page 38), and the selected channels must be deactivated.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pChipParams**
  Pointer to an array of chip parameters (see section XLchipParams on page 104).

Return value

Returns an error code (see section Error Codes on page 360).

5.3.6 xlCanSetChannelParamsC200

Syntax

```
XLstatus xlCanSetChannelParamsC200 (
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned char btr0,
  unsigned char btr1)
```

Description

This function initializes the channels defined by `accessMask` with the given parameters. In order to call this function, the port must have **init access** (see section xlOpenPort on page 38), and the selected channels must be deactivated.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **btr0**
  BTRO value for a C200 or 527 compatible controllers.
- **btr1**
  BTR1 value for a C200 or 527 compatible controllers.

Return value

Returns an error code (see section Error Codes on page 360).

5.3.7 xlCanSetChannelBitrate

Syntax

```
XLstatus xlCanSetChannelBitrate (
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned long bitrate)
```

Description

This function provides a simple way to specify the bitrate. The sample point is about 69 % (SJW=1, samples=1).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **bitrate**
  Bit rate in BPS. May be in the range 15000 … 1000000.

Return value

Returns an error code (see section Error Codes on page 360).

5.3.8 xlCanSetChannelAcceptance

Syntax

```
XLstatus xlCanSetChannelAcceptance(
  XlportHandle  portHandle,
  XLaccess      accessMask,
  unsigned long code,
  unsigned long mask,
  unsigned int  idRange)
```

Description

A filter lets pass messages. Different ports may have different filters for a channel. If the CAN hardware cannot implement the filter, the driver virtualizes filtering.

However, in some configurations with multiple ports, the application will receive messages although it has installed a filter blocking those message IDs.

Accept `if ((id ^ code) & mask) == 0)`.



Note
By default, all IDs are accepted after xlOpenPort() . Generally, modern computers are fast enough to receive all CAN messages. Therefore, it is recommended that the application implements filtering with its own logic. For standard IDs, xlCanAddAcceptanceRange()/ xlCanRemoveAcceptanceRange() provide a more flexible interface to configure filters.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **code**
  The acceptance code for id filtering.
- **mask**
  The acceptance mask for id filtering, bit = 1 means relevant.
- **idRange**
  To distinguish whether the filter is for standard or extended identifiers:
  `XL_CAN_STD`
  `XL_CAN_EXT`

Return value

Returns an error code (see section Error Codes on page 360).

Example
Several acceptance filter settings.

	IDs	mask	code	idRange
Std.	Open for all IDs	0x000	0x000	XL_CAN_STD
	Open for ID 1, ID=0x001	0x7FF	0x001	XL_CAN_STD
	Close for all IDs	0xFFF	0xFFF	XL_CAN_STD
Ext.	Open for all IDs	0x000	0x000	XL_CAN_EXT
	Open for ID 1, ID=0x80000001	0x1FFFFFFF	0x001	XL_CAN_EXT
	Close for all IDs	0xFFFFFFFF	0xFFFFFFFF	XL_CAN_EXT

Example

Open filter for all standard message IDs

```
xlStatus = xlCanSetChannelAcceptance(m_XLportHandle,
                                     m_xlChannelMask,
                                      0x000,
                                      0x000,
                                      XL_CAN_STD);
```

5.3.9 xlCanAddAcceptanceRange

Syntax

```
XLstatus xlCanAddAcceptanceRange(
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned long first_id,
  unsigned long last_id)
```

Description

This function sets the filter for accepted **standard IDs** and can be called several times to open multiple ID windows. Different ports may have different filters for a channel. If the CAN hardware cannot implement the filter, the driver virtualizes filtering.

However, in some configurations with multiple ports, the application will receive messages although it has installed a filter blocking those message IDs.

Note
By default, all **standard IDs** are accepted after xlOpenPort() . To receive only a specific ID range, the acceptance filter must be removed before. Generally, modern computers are fast enough to receive all CAN messages. Therefore, it is recommended that the application implements filtering with its own logic.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

- **first_id**
  First ID to pass acceptance filter.
- **last_id**
  Last ID to pass acceptance filter.

Return value

Returns an error code (see section Error Codes on page 360).



Example

Receiving IDs between 10…17 and 22…33

```
xlStatus = xlCanAddAcceptanceRange(XLportHandle,
                                   xlChannelMask,
                                   10,
                                   17);

xlStatus = xlCanAddAcceptanceRange(XLportHandle,
                                    xlChannelMask,
                                   22,
                                   33);
```

5.3.10 xlCanRemoveAcceptanceRange

Syntax

```
XLstatus xlCanRemoveAcceptanceRange(
  XLportHandle  portHandle,
  XLaccess       accessMask,
  unsigned long first_id,
  unsigned long last_id)
```

Description

The specified IDs will not pass the acceptance filter. The range of the acceptance filter can be removed several times. Different ports may have different filters for a channel. If the CAN hardware cannot implement the filter, the driver virtualizes filtering.

However, in some configurations with multiple ports, the application will receive messages although it has installed a filter blocking those message IDs.



Note
By default, all **standard IDs** are accepted after xlOpenPort() .
This function is for **standard IDs** only. Generally, modern computers are fast enough to receive all CAN messages. Therefore, it is recommended that the application implements filtering with its own logic.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **first_id**
  First ID to remove.
- **last_id**
  Last ID to remove (inclusive).

Return value

Returns an error code (see section Error Codes on page 360).



Example

Removing range between 10…13 and 27…30

```
xlStatus = xlCanRemoveAcceptanceRange(XLportHandle,
                                      xlChannelMask,
                                      10,
                                      13);

xlStatus = xlCanRemoveAcceptanceRange(XLportHandle,
                                    xlChannelMask,
                                      27,
                                      30);
```

5.3.11 xlCanResetAcceptance

Syntax

```
XLstatus xlCanResetAcceptance (
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int idRange)
```

Description

Resets the acceptance filter. The selected filters (depending on the `idRange` flag) are open.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **idRange**
  In order to distinguish whether the filter is reset for standard or extended identifiers.

  `XL_CAN_STD`
  Opens the filter for standard message IDs.

  `XL_CAN_EXT`
  Opens the filter for extended message IDs.

Return value

Returns an error code (see section Error Codes on page 360).



Example

Opening filter for all messages with extended IDs

```
xlStatus = xlCanResetAcceptance(XLportHandle,
                                xlChannelMask,
                                XL_CAN_EXT);
```

5.3.12 xlCanRequestChipState

Syntax

```
XLstatus xlCanRequestChipState (
```

```
  XlportHandle portHandle,
  XLaccess     accessMask)
```

Description

This function requests a CAN controller chipstate for all selected channels. For each channel an XL_CHIPSTATE event can be received by calling xlReceive().

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

Return value

Returns an error code (see section Error Codes on page 360).

5.3.13 xlCanTransmit

Syntax

```
XLstatus xlCanTransmit (
  XLportHandle portHandle,
  Xlaccess     accessMask,
  unsigned int *messageCount,
  void         *pMessages)
```

Description

This function transmits CAN messages on the selected channels. It is possible to transmit more messages with only one function call (see example below).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **messageCount**
  Points to the amount of messages to be transmitted
  or returns the number of transmitted messages.
- **pMessages**
  Points to a user buffer with messages to be transmitted,
  e. g. `XLevent xlEvent[100];`
  At least the buffer must have the size of `messageCount`.

Note
Each `xlEvent` has to be initialized to zero before calling `xlCanTransmit`,
e. g.: `memset(xlEvent, 0, sizeof(xlEvent));`

Output parameters

- **pMessages**
  Returns the number of successfully transmitted messages.

Return value

Returns `XL_SUCCESS` if all requested messages have been successfully transmitted. If no message or not all requested messages have been transmitted because the internal transmit queue is full, `XL_ERR_QUEUE_IS_FULL` is returned (see section Error Codes on page 360)



Example

Transmitting 100 CAN messages with the ID = 4

```
XLevent xlEvent[100];
memset(xlEvent, 0, sizeof(xlEvent)); // required init.
int nCount = 100;

for (i=0; i<nCount;i++) {
  xlEvent[i].tag                 = XL_TRANSMIT_MSG;
  xlEvent[i].tagData.msg.id      = 0x04;
  xlEvent[i].tagData.msg.flags   = 0;
  xlEvent[i].tagData.msg.data[0] = 1;
  xlEvent[i].tagData.msg.data[1] = 2;
  xlEvent[i].tagData.msg.data[2] = 3;
  xlEvent[i].tagData.msg.data[3] = 4;
  xlEvent[i].tagData.msg.data[4] = 5;
  xlEvent[i].tagData.msg.data[5] = 6;
  xlEvent[i].tagData.msg.data[6] = 7;
  xlEvent[i].tagData.msg.data[7] = 8;
  xlEvent[i].tagData.msg.dlc     = 8;
}
xlStatus = xlCanTransmit(portHandle, accessMask,&nCount,
xlEvent);
```

5.3.14 xlCanFlushTransmitQueue

Syntax

```
XLstatus xlCanFlushTransmitQueue(
  XLportHandle  portHandle,
  XLaccess      accessMask)
```

Description

The function flushes the transmit queues of the selected channels.

This function is only supported by XL family devices. On newer devices, this function performs no operation and returns XL_SUCCESS.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Return value

Returns an error code (see section Error Codes on page 360).

5.4 Structs

5.4.1 XLchipParams

Syntax

```
struct {
  unsigned long bitRate;
  unsigned char sjw;
  unsigned char tseg1;
  unsigned char tseg2;
  unsigned char sam;
};
```

Parameters

- **bitRate**
  This value specifies the real bitrate. (e. g. 125000)
- **sjw**
  Bus timing value sample jump width.
- **tseg1**
  Bus timing value tseg1.
- **tseg2**
  Bus timing value tseg2.
- **sam**
  Bus timing value. Samples may be 1 or 3.

Note
For more information on the bit timing of CAN controller please refer to the CAN literature or CAN controller data sheets.

Example

Calculation of baudrate
Baudrate = f/(2*presc*(1+tseg1+tseg2))

presc: CAN-Prescaler [1..64] (will be conformed autom.)
sjw: CAN-Synchronization-Jump-Width [1..4]
tseg1: CAN-Time-Segment-1 [1..16]
tseg2: CAN-Time-Segment-2 [1..8]
sam: CAN-Sample-Mode 1:3 Sample
f: crystal frequency is 16 MHz

Presc	sjw	tseg1	tseg2	sam	Baudrate
1	1	4	3	1	1 MBd
1	1	8	7	1	500 kBd
4	4	12	7	3	100 kBd
32	4	16	8	3	10 kBd

5.5 Events

5.5.1 XL CAN Message

Syntax

```
struct s_xl_can_msg {
  unsigned long      id;
  unsigned short     flags;
  unsigned short     dlc;
  XLuint64           res1;
  unsigned char      data[MAX_MSG_LEN];
  XLuint64           res2;
};
```

Description

This structure is used for received CAN events as well as for CAN messages to be transmitted.

Tag

- `XL_RECEIVE_MSG`
  Tag indicating CAN receive events, retrieved via xlReceive().
- `XL_TRANSMIT_MSG`
  Tag to be set for CAN messages to be transmitted, i. e. before calling xlCanTransmit().

For an event tag overview refer to section XLevent on page 70.

Parameters

- **id**
  The CAN identifier of the message. If the MSB of the id is set, it is an extended identifier (see `XL_CAN_EXT_MSG_ID`).

  flags
  `XL_CAN_MSG_FLAG_ERROR_FRAME`
  The event is an error frame (Rx*).

  `XL_CAN_MSG_FLAG_OVERRUN`
  An overrun occurred, events have been lost (Rx, Tx*).

  `XL_CAN_MSG_FLAG_REMOTE_FRAME`
  The event is a remote frame (Rx, Tx*).

  `XL_CAN_MSG_FLAG_TX_COMPLETED`
  Notification for successful message transmission (Rx*).

  `XL_CAN_MSG_FLAG_TX_REQUEST`
  Request notification for message transmission (Rx*).

  `XL_CAN_MSG_FLAG_NERR`
  The transceiver reported an error while the message was received (Rx*).

  `XL_CAN_MSG_FLAG_WAKEUP`
  High voltage message for Single Wire (Rx, Tx*).
  To flush the queue and transmit a high voltage message, combine the flags `XL_CAN_MSG_FLAG_WAKEUP` and `XL_CAN_MSG_FLAG_OVERRUN` by a binary OR.

  `XL_CAN_MSG_FLAG_SRR_BIT_DOM`
  SSR (Substitute Remote Request) bit in CAN message is set (Rx, Tx*).
  Only available with extended CAN identifiers.
  *: “Rx” indicates that the flag can be set by the driver for an event with tag `XL_RECEIVE_MSG`. “Tx” indicates that the flag can be set by the application for an event with tag `XL_TRANSMIT_MSG`.
- **dlc**
  Length of the data in bytes (0…8).
- **res1**
  Reserved for future use. Set to 0.
- **data**
  Array containing the data.
- **res2**
  Reserved for future use. Set to 0.

5.5.2 XL Chip State

Syntax

```
struct s_xl_chip_state {
  unsigned char busStatus;
  unsigned char txErrorCounter;
  unsigned char rxErrorCounter;
};
```

Description

This event occurs after calling xlCanRequestChipState().

Tag

`XL_CHIP_STATE` (see section XLevent on page 70).

Parameters

- **busStatus**
  Returns the state of the CAN controller. The following codes are possible:

  `XL_CHIPSTAT_BUSOFF`
  The bus is offline.

  `XL_CHIPSTAT_ERROR_PASSIVE`
  One of the error counters has reached the error level.

  `XL_CHIPSTAT_ERROR_WARNING`
  One of the error counters has reached the warning level.

  `XL_CHIPSTAT_ERROR_ACTIVE`
  The bus is online.

- **txErrorCounter**
  Error counter for the transmit section of the CAN controller.

- **rxErrorCounter**
  Error counter for the receive section of the CAN controller.

5.6 Application Examples

5.6.1 xlCANdemo

5.6.1.1 General Information

Description

This example demonstrates the basic handling of CAN and CAN FD. The program contains a command line interface:

xlCANdemo <Baudrate> <ApplicationName> <Identifier>

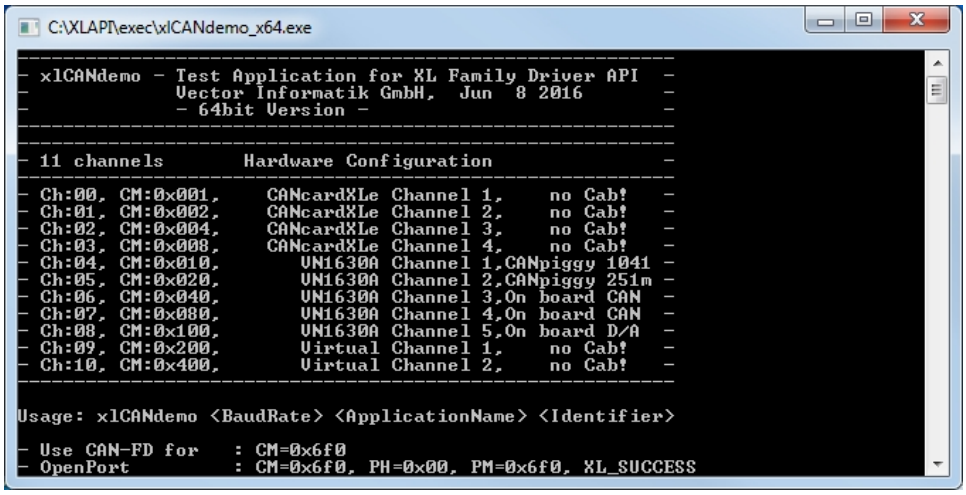


Figure 19: Running xlCANdemo

5.6.1.2 Keyboard Commands

The running application can be controlled via the following keyboard commands:

Key	Command
<t>	Transmit a message
<B>	Transmit a message burst
<M>	Transmit a remote message
<G>	Request chip state
<S>	Start/stop
<R>	Reset clock
<+>	Select channel (up)
<->	Select channel (down)
<i>	Select transmit Id (up)
<I>	Select transmit Id (down)
<X>	Toggle extended/standard Id
<O>	Toggle output mode
<A>	Toggle timer
<V>	Toggle logging to screen
<P>	Show hardware configuration
<H>	Help
<ESC>	Exit

5.6.1.3 Functions

Description

The source file `xlCANdemo.c` contains all needed functions:

- **demoInitDriver()**
  This function opens the driver and reads the actual hardware configuration. A valid `channelMask` is calculated and **one** port is opened afterwards.
- **demoInitDriver()**
  In order to read the driver message queue a thread is generated.

5.6.2 xlCANcontrol

5.6.2.1 General Information

Description

This example demonstrates the basic CAN handling with the **XL Driver Library** and a simple graphical user interface. The application needs two CAN channels to run and searches for Vector devices on the very first start. Two CAN are then automatically assigned to the application which is also added to the Vector Hardware Manager.

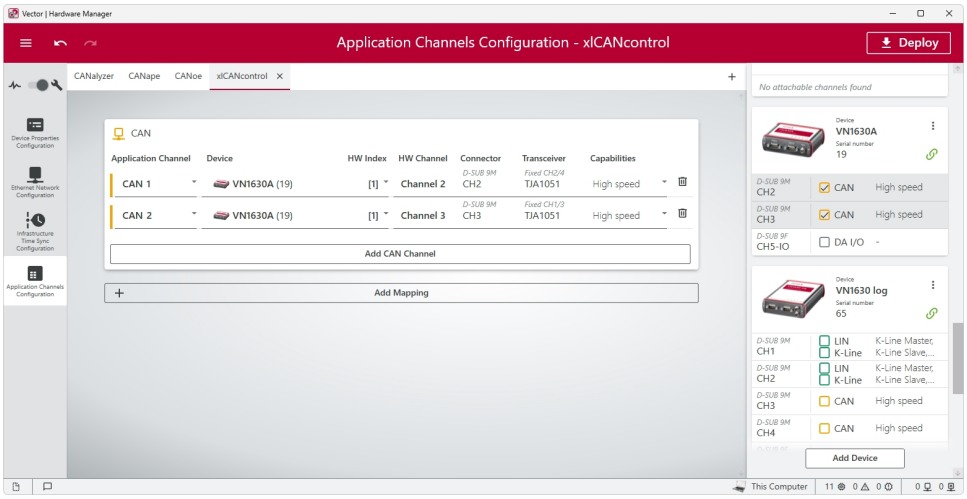


Figure 20: Example of hardware settings

Note

If you want to use other CAN channels, close the application and change the assignments in the Vector Hardware Manager. Execute the application again.

The assigned channels are displayed in the Hardware box. After pressing the **[Go OnBus]** button, both CAN channels are initialized with the selected baud rate.

In order to transmit a CAN message, set up the desired ID (standard or extended), DLC, databytes and press the **[Send]** button. The transmitted CAN message is displayed in the window (there is a Tx complete message from the transmit channel, and the received message on the second channel per default).

During the measurement the acceptance filter range can be changed with the **[Set filter]** or **[Reset filter]** button.

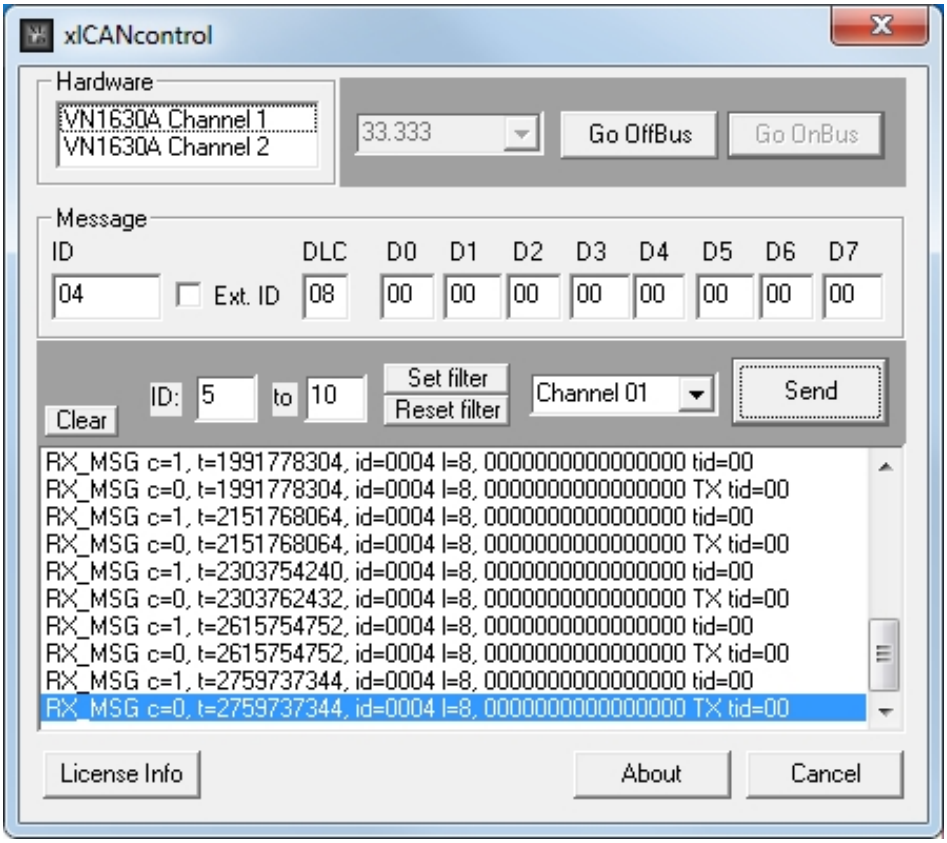


Figure 21: xlCANcontrol accessing VN1630A (CH1/CH2)

5.6.2.2 Classes

Description

The example has the following class structure:

- **CaboutDlg**
  About box.
- **CXLCANcontrolApp**
  Main MFC class → `xlCANcontrol.cpp`
- **CXLCANcontrolDlg**
  The 'main' dialog box → `xlCANcontrollDlg.cpp`
- **CCANFunctions**
  Contains all functions for the LIN access → `xlCANFunctions.cpp`

5.6.2.3 Functions

Description

- **CANInit**
  This function is called on application start to get the valid `channelmasks` (access masks). Afterwards, one port is opened for the two channels and a thread is created to read the message queue.
- **CANGoOnBus**
  After pressing the **[Go OnBus]** button, the CAN parameters are set and both channels are activated.
- **CANGoOffBus**
  After pressing the **[Go OffBus]** button, the channels will be deactivated.

- **CANSend**
  Transmits the CAN message with xlCANtransmit().
- **CANResetFilter**
  Resets (open) the acceptance filter.
- **CANSetFilter**
  Sets the acceptance filter range. It is needed to close the acceptance filter for every ID before.
- **canGetChannelMask**
  This function looks for assigned channels in the Vector Hardware Manager with xlGetApplConfig(). If there is no application registered, the application searchs for available CAN channels and assigns them in the Vector Hardware Manager with xlSetApplConfig(). The function fails if there are no valid channels found.
- **canInit**
  Opens one port with both channels (see section xlOpenPort on page 38).
- **canCreateRxThread**
  In order to readout the driver message queue, the application uses a thread (RxThread). An event is created and set up with xlSetNotification() to notify the thread.

6 CAN FD Commands

In this chapter you find the following information:

6.1 Introduction **114**
6.2 Flowchart **115**
6.3 Functions **116**
6.4 Structs **118**
6.5 Events **121**

6.1 Introduction

Description

The **XL Driver Library** enables the development of CAN FD applications for supported Vector devices (see section System Requirements on page 25). Multiple CAN applications can use a common physical CAN FD channel at the same time.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel configuration can be changed
- CAN FD messages can be transmitted on the channel
- CAN FD messages can be received on the channel

Without init access

- CAN FD messages can be transmitted on the channel
- CAN FD messages can be received on the channel

Reference
See the flowchart on the next page for all available functions and the according calling sequence.

6.2 Flowchart

Calling sequence

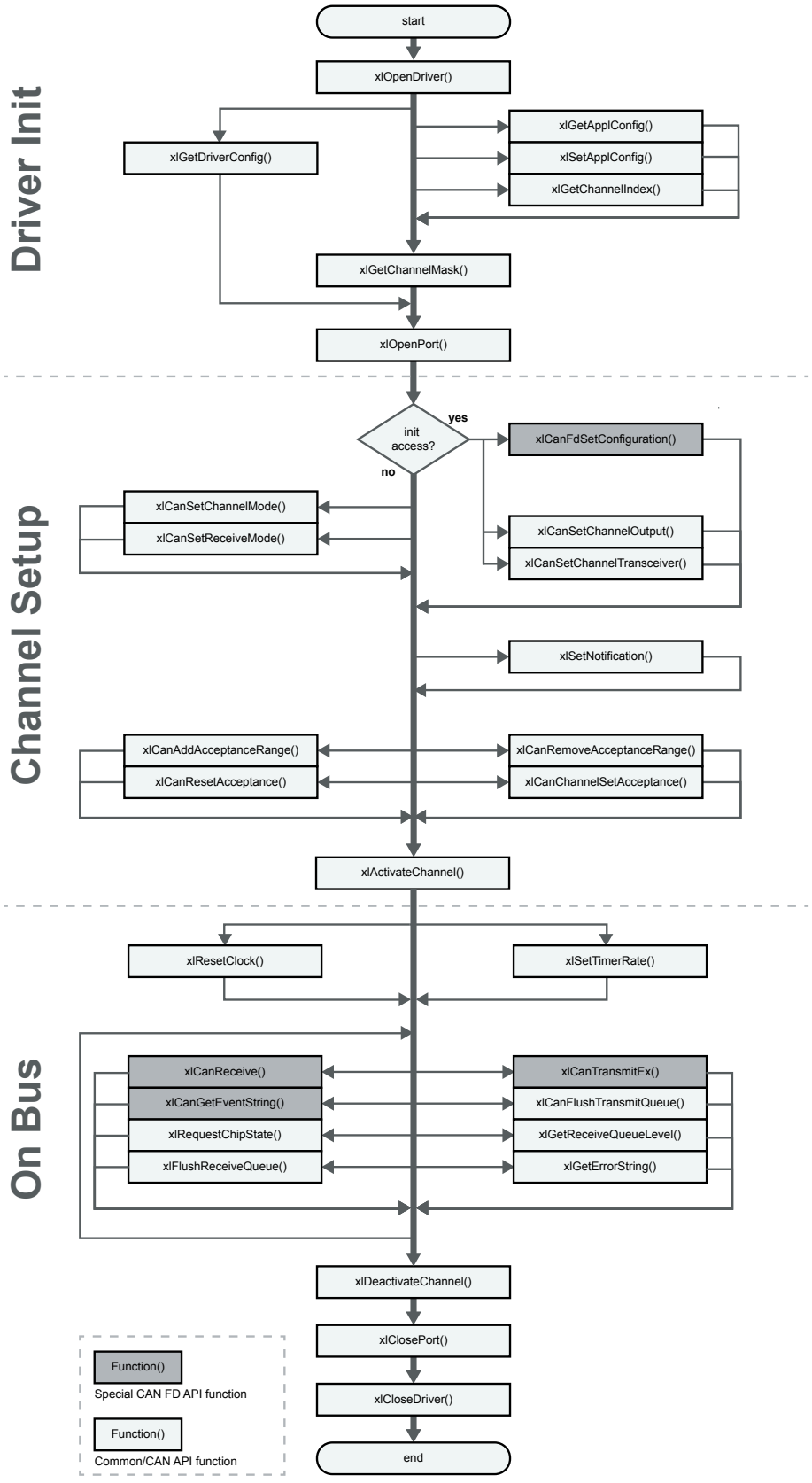


Figure 22: Function calls for CAN FD applications

6.3 Functions

6.3.1 xlCanFdSetConfiguration

Syntax

```
XLstatus xlCanFdSetConfiguration (
  XLportHandle portHandle,
  Xlaccess     accessMask,
  XLcanFdConf *pCanFdConf)
```

Description

Sets up a CAN FD channel. The structure differs between the arbitration part and the data part of a CAN message.



Note
To call this function the port must have **init access** (see section xlOpenPort on page 38) for the specified channels.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pCanFdConf**
  Points to the CAN FD configuration structure to set up a CAN FD channel (see section XLcanFdConf on page 118).

Return value

Returns an error code (see section Error Codes on page 360).

6.3.2 xlCanTransmitEx

Syntax

```
XLstatus xlCanTransmitEx (
  XLportHandle portHandle,
  Xlaccess     accessMask,
  unsigned int msgCnt,
  unsigned int *pMsgCntSent,
  XLcanTxEvent *pXlCanTxEvt)
```

Description

The function transmits CAN FD messages on the selected channels. It is possible to send multiple messages in a row (with a single call).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **msgCnt**
  Amount of messages to be transmitted by the user.

- **pMsgCntSent**
  Amount of messages which were transmitted.
- **pXlCanTxEvt**
  Points to a user buffer with messages to be transmitted (see section XLcanTxEvent on page 121).
  At least the buffer must have the size of `msgCnt`.

Return value

Returns `XL_SUCCESS` if all requested messages have been successfully transmitted. If no message or not all requested messages have been transmitted because the internal transmit queue is full, `XL_ERR_QUEUE_IS_FULL` is returned (see section Error Codes on page 360)

6.3.3 xlCanReceive

Syntax

```
XLstatus xlCanReceive (
  XLportHandle portHandle,
  XLcanRxEvent *pXlCanRxEvt)
```

Description

The function receives the CAN FD messages on the selected port.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Input/output parameters

- **pXLCanRxEvt**
  Pointer to the application allocated receive event buffer (see section XLcanRxEvent on page 122).

Return value

`XL_ERR_QUEUE_IS_EMPTY`: No event is available (see section Error Codes on page 360)

6.3.4 xlCanGetEventString

Syntax

```
XLstringType xlCanGetEventString (
  XLcanRxEvent *pEv
)
```

Description

This function returns a string based on the passed CAN Rx event data.

Input parameters

- **pEv**
  Points the CAN Rx event buffer to be parsed (see section XLcanRxEvent on page 122).

Return value

Returns an error code (see section Error Codes on page 360).

6.4 Structs

6.4.1 XLcanFdConf

Syntax

```
typedef struct {
  unsigned int arbitrationBitRate;
  unsigned int sjwAbr;
  unsigned int tseg1Abr;
  unsigned int tseg2Abr;
  unsigned int dataBitRate;
  unsigned int sjwDbr;
  unsigned int tseg1Dbr;
  unsigned int tseg2Dbr;
  unsigned char reserved;
  unsigned char options;
  unsigned char reserved1[2];
  unsigned int reserved2;
} XLcanFdConf;
```

Parameters

- **arbitrationBitRate**
  Arbitration CAN bus timing for nominal / arbitration bitrate in bit/s.
- **sjwAbr**
  Arbitration CAN bus timing value (sample jump width).
  Range: 0 < sjwAbr <= min(tseg2Abr, 128).
- **tseg1Abr**
  Arbitration CAN bus timing tseg1.
  Range: 1 < tseg1Abr < 255.
- **tseg2Abr**
  Arbitration CAN bus timing tseg2.
  Range: 1 < tseg2Abr < 255.
- **dataBitRate**
  CAN bus timing for data bitrate in bit/s.
  Range: dataBitRate >= max(arbitrationBitRate, 25000).
- **sjwDbr**
  Data phase CAN bus timing value (sample jump width).
  Range: 0 < sjwDbr <= min(tseg2Dbr, 64).
- **tseg1Dbr**
  Data phase CAN bus timing for data tseg1.
  Range: 1 < tseg1Dbr < 127.
- **tseg2Dbr**
  Data phase CAN bus timing for data tseg2.
  Range: 1 < tseg2Dbr < 127.
- **reserved**
  Reserved for future use. Set to 0.
- **options**
  CANFD-BOSCH
  `CANFD_CONFOPT_NO_ISO`
- **reserved1[2]**
  Reserved for future use. Set to 0.

- **reserved2**
  Reserved for future use. Set to 0.

Example

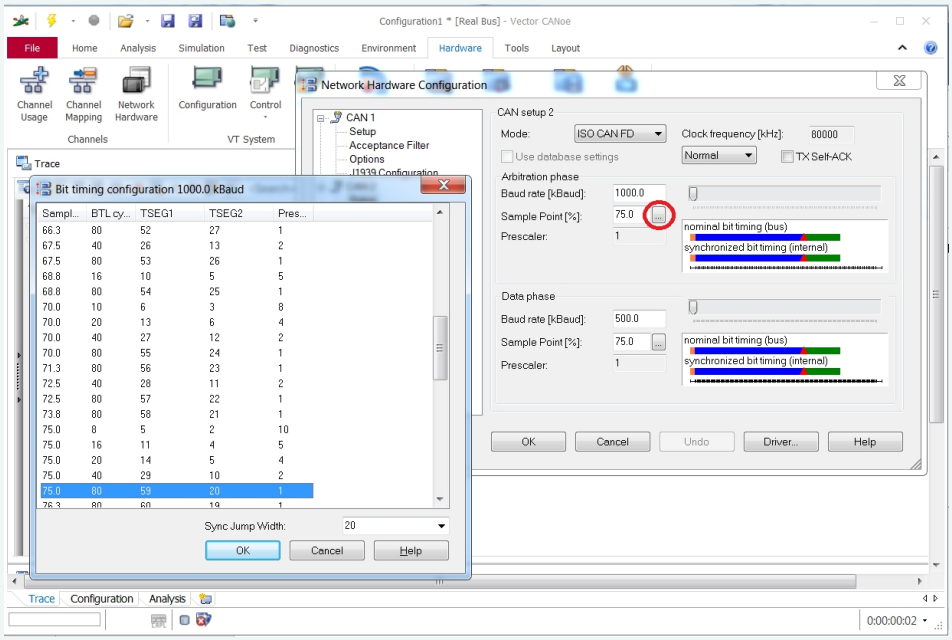
Deriving tseg1 and tseg2 for a given bitrate and sample point.

The ratio (tseg1+1)/(tseg1+tseg2+1) specifies the sample point. The constraint is that (tseg1+tseg2+1) must evenly divide the 80 MHz CAN clock at the desired bitrate. More precisely, the hardware attempts to determine a prescaler >= 1, such that (tseg1+tseg2+1)*actualBitrate*prescaler = 80MHz. Where actualBitrate differs by less than 1:256 from the requested bitrate.

Arbitration Phase					Data Phase				
Bitrate	Sample Point	Tseg1	Tseg2	Prescaler	Bitrate	Sample Point	Tseg1	Tseg2	Prescaler
20k	75%	149	50	20	100k	75%	119	40	5
125k	67,5%	53	26	8	1M	67,5%	53	26	1
500k	87,5%	27	4	5	2M	87,5%	34	5	1
1M	75%	59	20	1	8M	70%	6	3	1

Note

Software shall assume an 80 MHz CAN clock, although the hardware may internally use a different clock. If you have CANoe, you may calculate the tseg1 and tseg2 values with the **Network Hardware Configuration** dialog. The BTL-cycles equal tseg1+tseg2+1.



6.5 Events

6.5.1 XLcanTxEvent

Syntax

```
typedef struct {
  unsigned short tag;
  unsigned short transId;
  unsigned char  channelIndex;
  unsigned char  reserved[3];

  union {
    XL_CAN_TX_MSG canMsg;
  } tagData;
} XLcanTxEvent;
```

Description

This structure is used for CAN FD events that are transmitted by the application.

Parameters

- **tag**
  Event type. Set to `XL_CAN_EV_TAG_TX_MSG`.
- **transId**
  Internal use.
- **channelIndex**
  Internal use. The **accessMask** parameter of xlCanTransmitEx() specifies which channels send the message.
- **reserved**.
  Internal use.
- **tagData**
  Tag Data (see section XL_CAN_TX_MSG on page 121).

6.5.2 XL_CAN_TX_MSG

Syntax

```
typedef struct {
  unsigned int  canId;
  unsigned int  msgFlags;
  unsigned char dlc;
  unsigned char reserved[7];
  unsigned char data[XL_CAN_MAX_DATA_LEN];
} XL_CAN_TX_MSG;
```

Tag

XL_CAN_EV_TAG_TX_MSG

Parameters

- **canId**
  CAN ID (11 or 29 bits).
  For extended IDs: `canID = (XL_CAN_EXT_MSG_ID | id)`.

- **msgFlags**
  Set to 0 to transmit a CAN 2.0 frame.

  `XL_CAN_TXMSG_FLAG_BRS`
  Baudrate switch.

  `XL_CAN_TXMSG_FLAG_HIGHPRIO`
  High priority message. Clears all send buffers then transmits.

  `XL_CAN_TXMSG_FLAG_WAKEUP`
  Generates a wake up message.

  `XL_CAN_TXMSG_FLAG_EDL`
  This flag is used to indicate an extended CAN FD data length according to the table below.

  `XL_CAN_TXMSG_FLAG_RTR`
  This flag is used for Remote-Transmission-Request.
  Only useable for Standard CAN messages.

- **dlc**
  4-bit data length code.

DLC	Number of Data Bytes CAN 2.0	Number of Data Bytes CAN FD
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	8	12
10	8	16
11	8	20
12	8	24
13	8	32
14	8	48
15	8	64

- **reserved**
  Internal use.
- **data**
  Data to be transmitted.

6.5.3 XLcanRxEvent

Syntax

```
typedef struct {
  unsigned int    size;
```

```
  unsigned short  tag;
  unsigned char   channelIndex;
  unsigned char   reserved;
  unsigned int    userHandle;
  unsigned short  flagsChip;
  unsigned short  reserved0;
  XLuint64        reserved1;
  XLuint64        timeStampSync;

  union {
  XL_CAN_EV_RX_MSG     canRxOkMsg;
  XL_CAN_EV_RX_MSG     canTxOkMsg;
  XL_CAN_EV_TX_REQUEST canTxRequest;
  XL_CAN_EV_ERROR      canError;
  XL_CAN_EV_CHIP_STATE canChipState;
  XL_CAN_EV_SYNC_PULSE canSyncPulse;
  } tagData;
} XLcanRxEvent;
```

Description

This structure is used for CAN FD events that are received by the application.

Parameters

- **size**
  Overall size of the complete event.
- **tag**
  `XL_CAN_EV_TAG_RX_OK`
  `XL_CAN_EV_TAG_RX_ERROR`
  `XL_CAN_EV_TAG_TX_ERROR`
  `XL_CAN_EV_TAG_TX_REQUEST`
  `XL_CAN_EV_TAG_TX_OK`
  `XL_CAN_EV_TAG_CHIP_STATE`
  `XL_SYNC_PULSE`
- **channelIndex**
  Channel index of the hardware (see section xlGetChannelIndex on page 34).
- **reserved**
  Internal use.
- **userHandle**
  Internal use.
- **flagsChip**
  Queue overflow (upper 8bit), `XL_CAN_QUEUE_OVERFLOW`.
- **reserved0**
  Internal use.
- **reserved1**
  Internal use.
- **timeStampSync**
  Timestamp which is synchronized by the driver.
- **tagData**
  Tag Data. See the following sections for further details.

6.5.4 XL_CAN_EV_RX_MSG

Syntax

```
typedef struct {
  unsigned int   canId;
  unsigned int   msgFlags;
```

```
  unsigned int   crc;
  unsigned char  reserved1[12];
  unsigned short totalBitCnt;
  unsigned char  dlc;
  unsigned char  reserved[5];
  unsigned char  data[XL_CAN_MAX_DATA_LEN];
} XL_CAN_EV_RX_MSG;
```

Tag

XL_CAN_EV_TAG_RX_OK, XL_CAN_EV_TAG_TX_OK

Parameters

- **canId**
  CAN ID.
- **msgFlags**
  `XL_CAN_RXMSG_FLAG_EDL`
  Extended data length.

  `XL_CAN_RXMSG_FLAG_BRS`
  Baud rate switch.

  `XL_CAN_RXMSG_FLAG_ESI`
  Error state indicator.

  `XL_CAN_RXMSG_FLAG_EF`
  Error frame.

  `XL_CAN_RXMSG_FLAG_ARB_LOST`
  Arbitration lost.

  `XL_CAN_RXMSG_FLAG_RTR`
  Remote frame.

  `XL_CAN_RXMSG_FLAG_WAKEUP`
  High voltage message on single wire CAN.

  `XL_CAN_RXMSG_FLAG_TE`
  1: transceiver error detected.
- **crc**
  Crc of the CAN message.
- **totalBitCnt**
  Number of received bits including stuff bit.
- **dlc**
  4-bit data length code.
- **reserved**
  Internal use.
- **data**
  Data that was received.

6.5.5 XL_CAN_EV_ERROR

Syntax

```
typedef struct {
  unsigned char errorCode;
  unsigned char reserved[95];
} XL_CAN_EV_ERROR;
```

Tag XL_CAN_EV_TAG_RX_ERROR, XL_CAN_EV_TAG_TX_ERROR

Parameters

- **errorCode**
  XL_CAN_ERRC_BIT_ERROR
  XL_CAN_ERRC_FORM_ERROR
  XL_CAN_ERRC_STUFF_ERROR
  XL_CAN_ERRC_OTHER_ERROR
  XL_CAN_ERRC_CRC_ERROR
  XL_CAN_ERRC_ACK_ERROR
  XL_CAN_ERRC_NACK_ERROR
  XL_CAN_ERRC_OVLD_ERROR
  XL_CAN_ERRC_EXCPT_ERROR
- **reserved**
  Internal use.

6.5.6 XL_CAN_EV_CHIP_STATE

Syntax

```
typedef struct {
  unsigned char busStatus;
  unsigned char txErrorCounter;
  unsigned char rxErrorCounter;
  unsigned char reserved;
  unsigned int  reserved0;
} XL_CAN_EV_CHIP_STATE;
```

Tag XL_CAN_EV_TAG_CHIP_STATE

Parameters

- **busStatus**
  Returns the state of the CAN controller. The following codes are possible:

  XL_CHIPSTAT_BUSOFF
  The bus is offline.

  XL_CHIPSTAT_ERROR_PASSIVE
  One of the error counters has reached the error level.

  XL_CHIPSTAT_ERROR_WARNING
  One of the error counters has reached the warning level.

  XL_CHIPSTAT_ERROR_ACTIVE
  The bus is online.
- **txErrorCounter**
  Error counter for the transmit section of the CAN controller.
- **rxErrorCounter**
  Error counter for the receive section of the CAN controller.
- **reserved**
  Internal use.
- **reserved0**
  Internal use.

6.5.7 XL_CAN_EV_TX_REQUEST

Syntax

```
typedef struct {
  unsigned int    canId;
  unsigned int    msgFlags;
  unsigned char   dlc;
  unsigned char  txAttemptConf;
  unsigned short  reserved;
  unsigned char   data[XL_CAN_MAX_DATA_LEN];
} XL_CAN_EV_TX_REQUEST;
```

Tag

XL_CAN_EV_TAG_TX_REQUEST

Parameters

- **canId**
  CAN ID.
- **msgFlags**
  `XL_CAN_RXMSG_FLAG_EDL`
  Extended data length.

  `XL_CAN_RXMSG_FLAG_BRS`
  Baud rate switch.

  `XL_CAN_RXMSG_FLAG_ESI`
  Error state indicator.

  `XL_CAN_RXMSG_FLAG_EF`
  Error frame.

  `XL_CAN_RXMSG_FLAG_ARB_LOST`
  Arbitration lost.
- **dlc**
  4-bit data length code.
- **txAttemptConf**
  Reserved.
- **reserved**
  Internal use.
- **data**
  Data that was receive.

6.5.8 XL_SYNC_PULSE_EV

Syntax

```
typedef XL_SYNC_PULSE_EV XL_CAN_EV_SYNC_PULSE;

typedef struct s_xl_sync_pulse_ev {
  unsigned int  triggerSource;
  unsigned int  reserved;
  XLuint64      time;
} XL_SYNC_PULSE_EV;
```

Tag

XL_SYNC_PULSE

Parameters

- **triggerSource**
  `XL_SYNC_PULSE_EXTERNAL`
  The sync event comes from an external device.

  `XL_SYNC_PULSE_OUR`
  The sync pulse event occurs after an xlGenerateSyncPulse().

  `XL_SYNC_PULSE_OUR_SHARED`
  The sync pulse comes from the same hardware but from another channel.
- **reserved**
  Internal use.
- **time**
  Internally generated time stamp.

7 Ethernet Commands

In this chapter you find the following information:

7.1 Introduction **129**

7.2 Network Based Access Mode **136**

7.3 Channel Based Access Mode **163**

7.1 Introduction

7.1.1 General Information

For access to the Ethernet network, a distinction is made between two API variants:

- **channel-based access**,
  which was introduced with the VN5610(A) and VN5640 interface generation, and
- **network-based access**,
  which was introduced with the VN5620 and VN5430 interface generation.

From version V11.2 of the Windows Device Driver for the Vector Ethernet network interfaces, network-based access is supported.

Products that supported channel-based access until then also supports network-based access (see section Device Support on page 133). However, the new interface generation VN5620 and VN5430 and following only support network-based access to Ethernet networks.

Note
The channel-based access is only supported by the old Vector Hardware Configuration tool and is not recommended for new projects. Please use the network-based access in combination with the Vector Hardware Manager.

Note
The standard API and therefore the recommended API for accessing Ethernet networks is the **network-based access mode**.

Vector tools such as CANoe from Version 12.0 SP4 or CANape from Version 13.0 SP2 primarily rely on the network-based access mode.

The Vector Hardware Manager only supports the **network-based access mode**.

The access mode can be switched for supporting device drivers (see section Switching Access Mode on page 134).

The channel-based API will continue to be supported as a legacy API on devices that previously supported it.

7.1.2 Network-Based API vs. Channel-Based API

Restrictions of channel-based API

The channel-based access was developed up to driver version 11.2 and is an access concept based on the physical hardware channel (sending/receiving data on a specific hardware channel). The topology of the participants connected to the interface is only conditionally considered by the API or the configuration of the interface.

For larger setups and access to larger networks, a topology-oriented access method is more advantageous. For this purpose, the network-based access concept was designed.

Conceptual comparison

An exemplary topology (shown below) is used to illustrate the differences between the two access methods. The gray nodes are the real available nodes that are connected to the interface. The dotted nodes are virtual participants that communicate via the API.

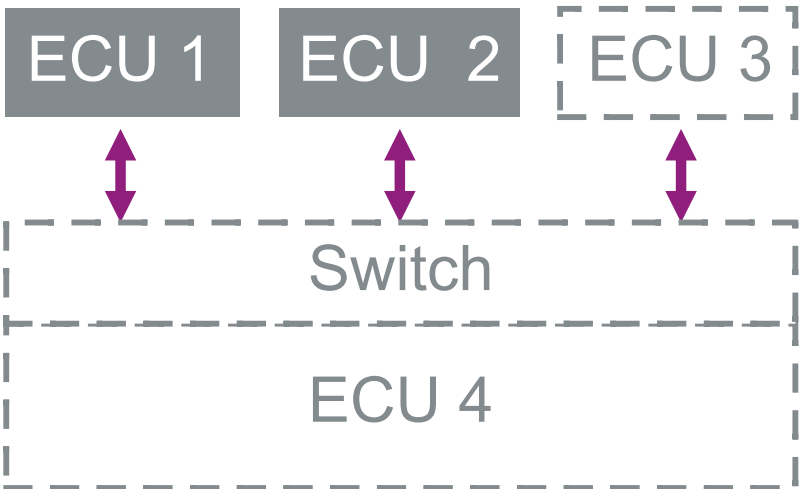


Figure 23: Example topology

Channel-based access

With the channel-based access method, the hardware port is the focus of the API. Messages are received or sent on a specific channel.

Messages must be addressed to the corresponding hardware port. During interaction between all participants (real connected ECUs and simulated participants), the topology is potentially changed (as shown in the example before: ECU4 contains a switch).

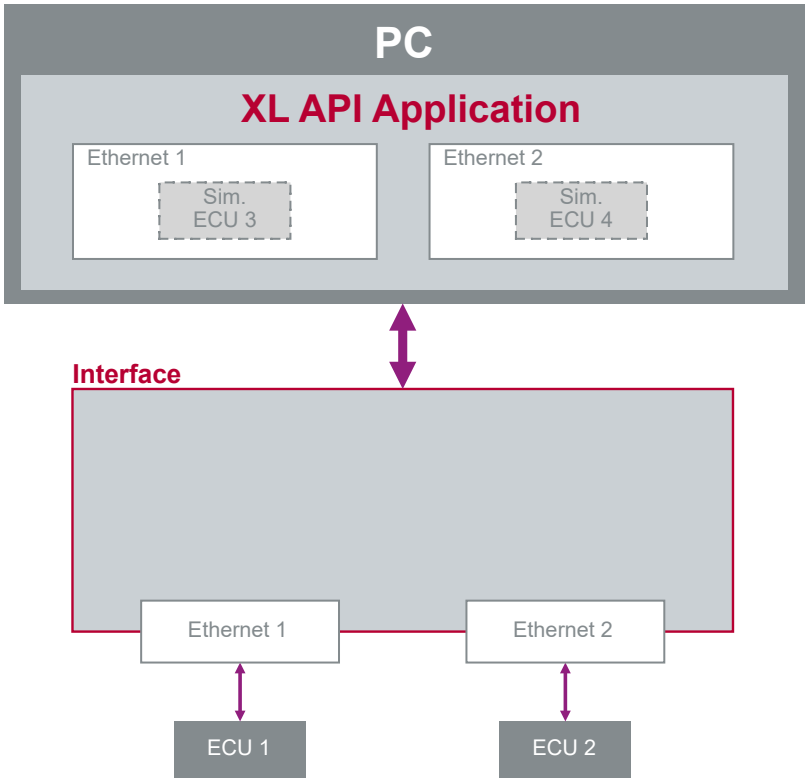


Figure 24: Example of channel-based access

Network-based access

For network-based access, the configuration of the interface hardware defines how all participants communicate with each other and what the communication paths are. The configuration can be used to decide how close it should be to the original topology.

By defining which ports (see definition of terms in Definitions on page 137) are located within a network, the application does not have to care about which physical port the target hardware is connected to.

By appropriate segmentation of the physical ports on the hardware, all participants in the network can be reached. The application can open virtual ports on the switches and thus, depending on the test requirements, a fast access to the network or a topology-compliant structure can be implemented.

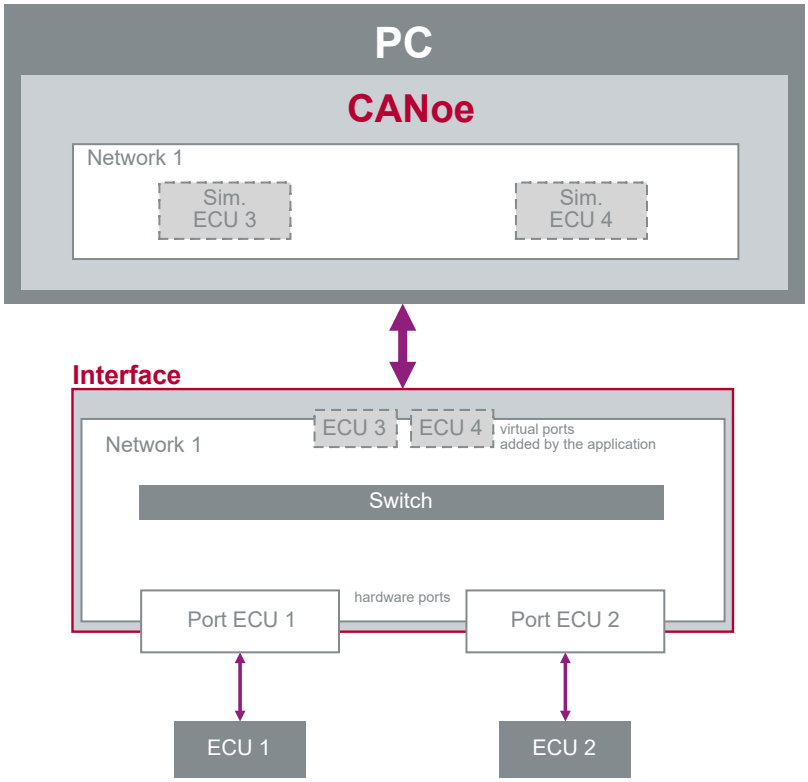


Figure 25: Example of network-based access

7.1.3 Device Support

Device	channel-based access mode	network-based access mode
VN5240	-	X
VN5430	-	X
VN5610A	X	X
VN5611	-	X
VN5612	-	X
VN5614	-	X
VN5620	-	X
VN5620A	-	X
VN5640	X	X
VN5650	-	X
VN7640	X	-
VT6306B	-	X
VX0312	X	-

7.1.4 Switching Access Mode

The Vector devices VN5640 and VN5610(A) can be switched between network-based access mode and channel-based access mode via the old Vector Hardware Config tool.

Note
To detect with help of the XL Driver Library API if a device works in network or channel-based access mode, the bit `XL_CHANNEL_FLAG_EX1_NET_ETH_SUPPORT` in the mask `channelCapabilities` of the structure XLchannelDrvConfig can be used which is returned by the function xlCreateDriverConfig ().

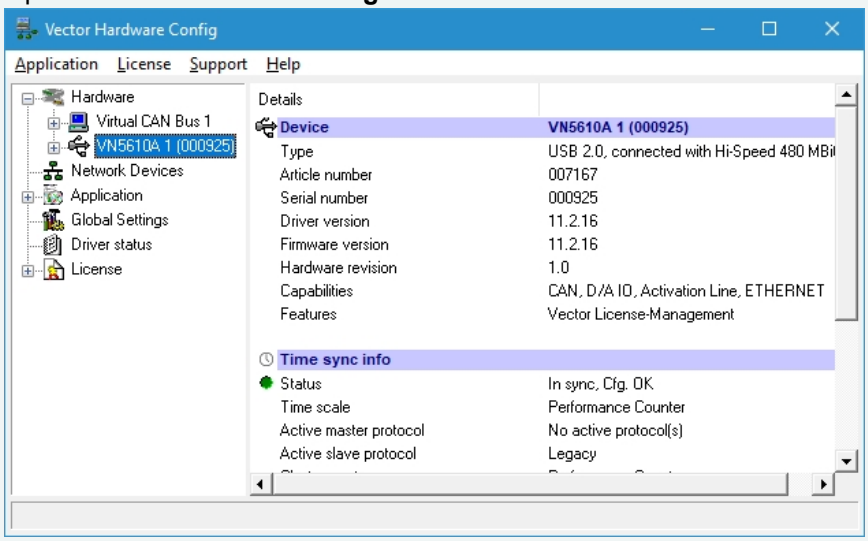
Modes

With driver version 11.2, the new **network-based** and the old **channel-based** Ethernet configuration are available. For new projects, the **network-based** Ethernet configuration is recommended. For this, the mode of the VN5000 interface must be switched once. Either via the old Vector Hardware Config tool or via CANoe V12 SP4 or later. The following steps show how to switch the mode in the old Vector Hardware Config tool.

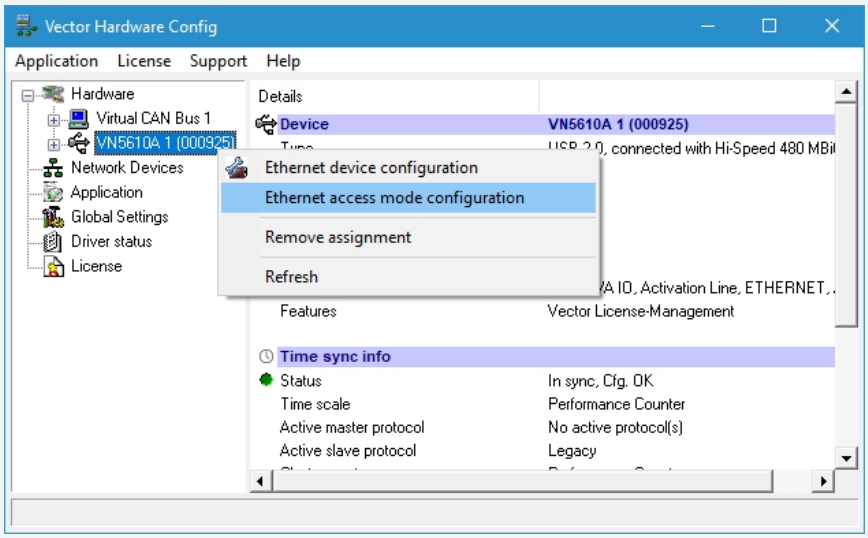
Note
Only the VN5610(A) and VN5640 support both modes. The new interfaces only support network-based Ethernet configuration. Accordingly, this step-by-step guide applies to VN5610(A)/VN5640 only.

Step by Step Procedure

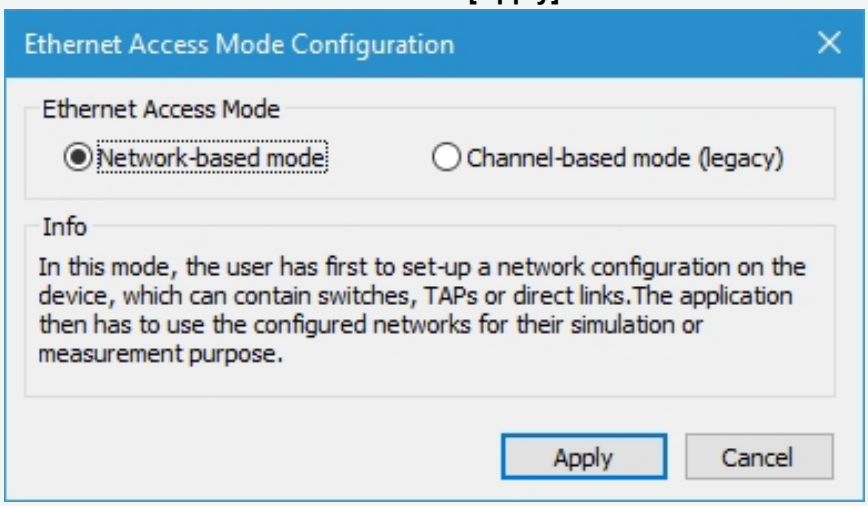
1. Open **Vector Hardware Config**.



2. With a right-click, select an installed VN5000 interface and click **Ethernet access mode configuration** in the context menu.



3. Select **Network-based mode** and click **[Apply]**.



4. Wait until mode programming has finished.

7.2 Network Based Access Mode

7.2.1 Basic Concept

With effect from device driver version 11.2 of the Vector Ethernet Network Interfaces, Vector introduces a new way of Ethernet configuration for all Vector Ethernet network interfaces.

The device firmware before version 11.2 allows a maximum of one switch segment per Ethernet interface. Therefore, two Ethernet interfaces are required. The Ethernet interfaces are connected to CANoe via two application channels (ETH1 and ETH2). This results in two Ethernet networks in the simulation setup of CANoe:

Channel-based access mode

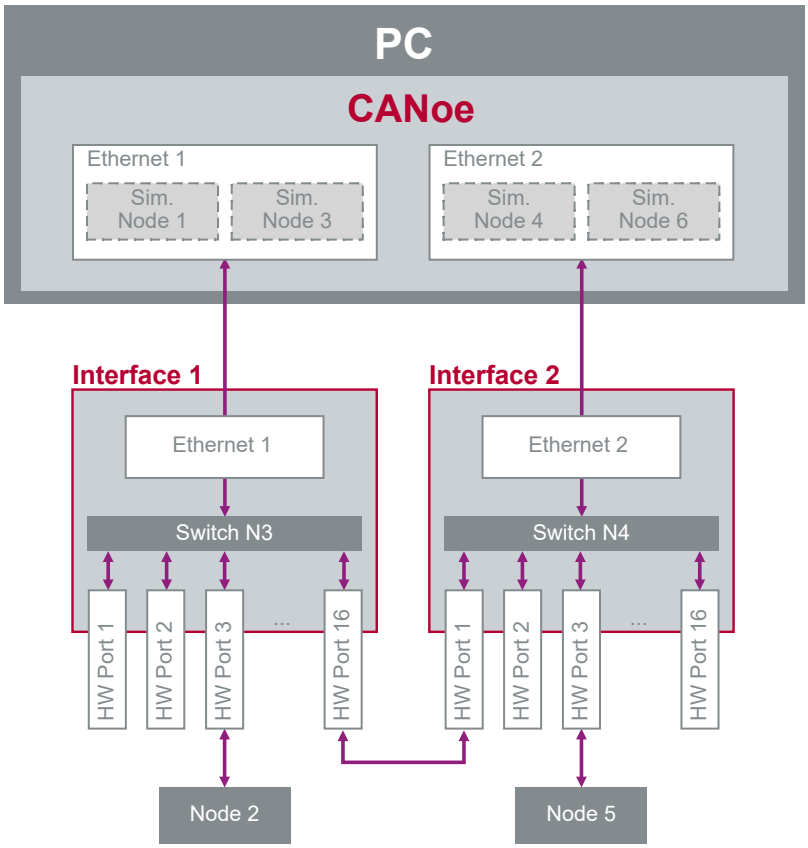


Figure 26: Simulation setup of the device firmware before version 11.2

The device firmware from version 11.2 allows free segmentation. Therefore, two switch segments with the associated ports can be defined with this version. Both segments are assigned to the same network **Network 1**. In CANoe, in this setup only one Ethernet network is required in the Simulation setup (network name = **Network 1**).

Network-based access mode

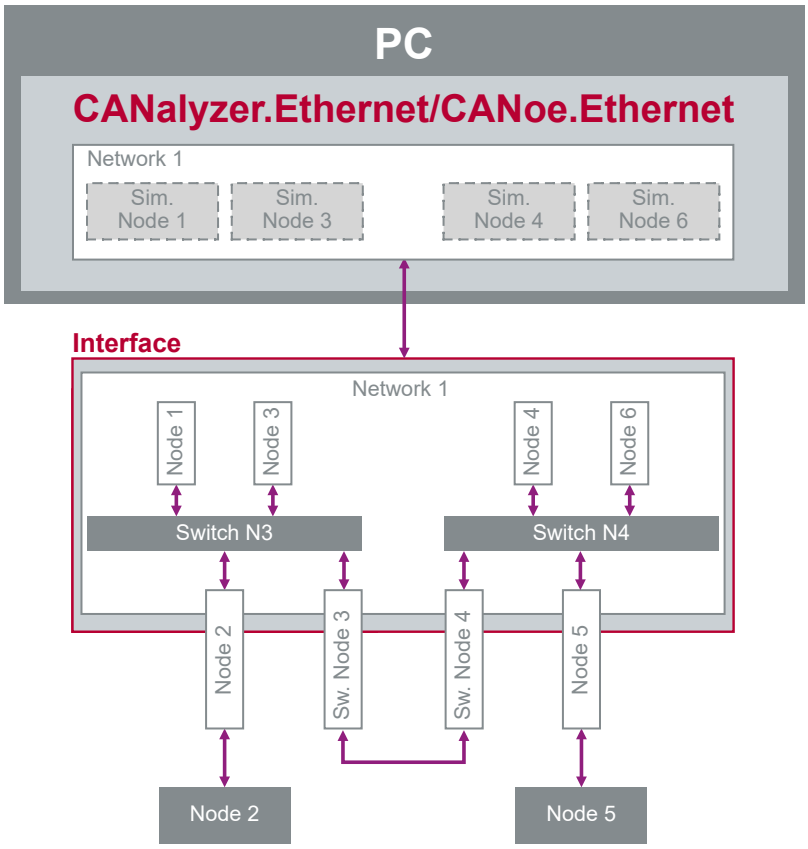


Figure 27: Simulation setup of the device firmware from version 11.2

7.2.2 Definitions

Terms

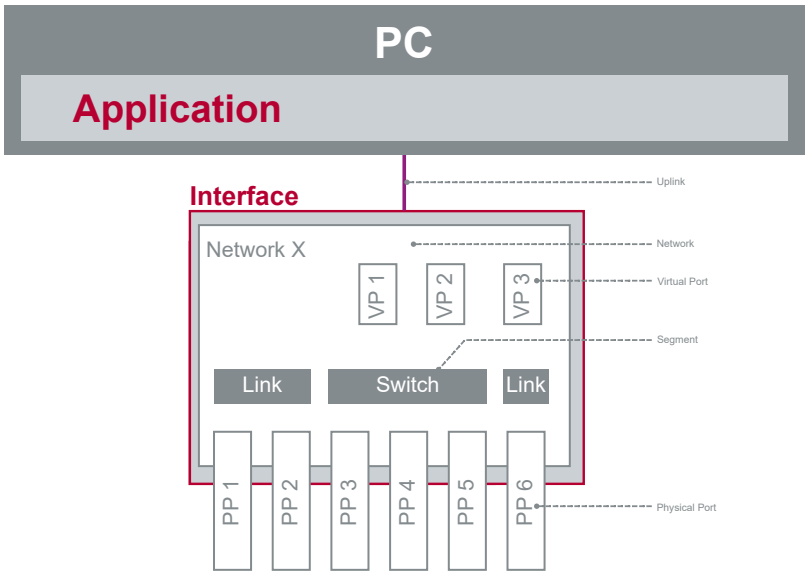


Figure 28: Simulation setup of the device driver from version 11.2

- Port
  A port is an access point for an application like CANoe, CANape or a real device like an ECU. A distinction is made between a physical port and a virtual port. Each port has a unique name and is assigned to exactly one segment.
- Physical Port
  Each Vector Ethernet network interface provides a defined number of physical connections. Exactly one physical port is assigned to each physical connection at one point in time. An application can configure physical layer properties by means of the physical port. An application has only read access to a physical port, apart from stress sending.



Note
Assigning a physical port to a segment enables the usage of a physical connection. Unassigned physical ports will disable the physical connection and no communication can take place.

- Virtual Port
  A virtual port does not represent any physical connection. Therefore, no physical layer properties can be configured. Several applications can have read access to a virtual port. Only one application can have write access to a virtual port.

Note
Simulation nodes (e. g. CAPL program or interactive generator block) are connected to the Ethernet network by means of virtual ports. The number of virtual ports that can be created is limited by the hardware (e. g. a VN5640 supports up to 32 virtual ports). In most cases, the virtual ports are automatically created by the application and no user interaction is necessary.

- Segment
  A segment acts as a coupling element between ports. At least one segment must be created and connected to a physical port. Each segment has a unique name and is assigned to exactly one network. Two types of segments are available: switch and link.

Note
Segments can be set up on the interface via a graphic interface (Ethernet Device Configuration).

- Switch Segment
  A switch segment provides the basic functions of a layer-2 switch. Any number of ports can be assigned to a switch segment.

- Link Segment
  A link segment always connects exactly two ports transparently to each other. The link segment is used to transparently forward Ethernet packets and the states of the physical layer.

Note
A link segment is used when the message traffic between two ports is to be considered.

  - TAP (Test Access Point)
    Connection of two physical ports with very low and constant latency (≤ 8 µs). A TAP is functionally similar to the MAC bypass offered in driver before version 11.2.
  - Direct Connection
    Connection of one physical port to one virtual port.

- Network
  A network groups one or more segments. As a minimum one network must be defined per device. One device can support multiple different networks. A network has a unique name. A network can span over multiple devices.

Note
The network name is also used to connect applications to the devices. For example, in CANoe the network name can be specified in the System View in the Simulation Setup.

- Uplink
  An uplink connects the device to a host. Filters can be configured to reduce the data transfer on an uplink. Depending on the Vector device, USB, Ethernet or PCIe uplink is possible.

7.2.3 General Information

Configuring networks

The XL Driver Library can only open networks that have been previously configured on a Vector device. Therefore, before an XL API application can perform measurement or simulation on a device, you must write an appropriate network configuration to the device using the Vector Hardware Manager. Usage instructions for the tool are available in the tool's integrated help.

Accessing Vector network (devices)

The usage of the XL Driver Library (in network based access mode) can be split into three major steps:

- **Step 1: Driver initialization**
  - Open a network of a certain type (for example Ethernet). A network can span over multiple devices.
- **Step 2: Network setup**
  Configuration of the opened network:
  - Open or add virtual ports.
  - Connect to physical ports with measurement points.
- **Step 3: On network/measurement tasks**
  - Definition of main tasks for Tx and Rx packets (messages).



Note
The network based access mode and the channel based access mode are different XL Driver Library APIs. Do not mix the `xlNet*()` function calls with `xlEth*()`.

7.2.3.1 Step 1: Driver Initialization

Open a network

Before you can setup and work with a network, you have to open the network with its configured name.

The network configuration (segmentation, naming, …) is typically done via the Vector Hardware Manager. The available network and port names can be retrieved with the function call xlCreateDriverConfig().

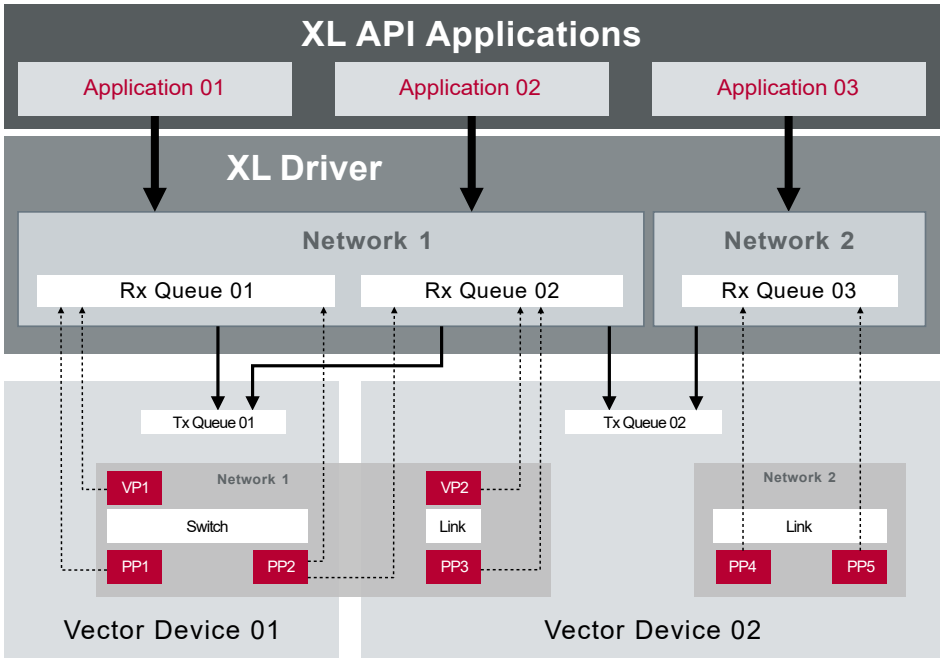


Figure 29: Principle structure of network applications

Multiple applications

In general, if a different application demands access on networks, the XL Driver Library returns another `XLnetworkHandle`. Applications can access physical ports at the same time, but not virtual ports. There is no initial access handling.

Note
An application can also open multiple networks at the same time.

Network handle

Once the network name is passed over to the xlNetEthOpenNetwork() function, the XL Driver Library returns a specific `XLnetworkHandle` that is used for all subsequent function calls on this network.

Note
The `XLnetworkHandle` is different from the `XLportHandle` used in channel based access mode and has a different meaning.

7.2.3.2 Step 2: Network Setup

Hardware initialization

After opening the network(s) and before an Ethernet packet can be transmitted or received, physical or virtual ports must be connected or opened.

To transmit an Ethernet packet directly from a physical port or to retrieve Ethernet packets from it, a measurement point with the name of the physical port must be connected with the function xlNetConnectMeasurementPoint().

To send an Ethernet packet into a switch or direct segment, a virtual port must be added with the function xlNetAddVirtualPort() or opened with xlNetOpenVirtualPort() if the port was already configured with the Vector Hardware Manager.

At the end of the setup, the network must be activated to transmit or receive Ethernet packets.



Note
On a switch segment, several virtual ports can be added. On a direct segment, one virtual port can be added.

All port names must be unique in a network.



Reference
For further information on the network setup, please refer to the flowchart on page 144.

Port handle

Once a measurement point has been connected or a virtual port has been added or opened, the XL Driver Library returns a specific port handle `XLethPortHandle` that is used for all subsequent function calls on this measurement point or virtual port.



Note
The `XLethPortHandle` is different from the `XLportHandle` used in channel based access mode and has a different meaning.

Receive handle

If a measurement point is connected or a virtual port is added or opened, the application must assign a receive handle `XLrxHandle` that identifies events originating from this measurement point or virtual port. The receive handles should be assigned in a way that allow the application to uniquely identify the source of an event.

7.2.3.3 Step 3: On Network/Measurement Tasks

Transmitting Ethernet packets

After the driver has been initialized and the networks set up, the actual functionality is performed in the main task. Each device is equipped with a common transmit queue. The transmit Ethernet packets are added to the matching queue as selected by the port handle `XLethPortHandle` returned by the functions xlNetConnectMeasurementPoint(), xlNetAddVirtualPort() and xlNetOpenVirtualPort() in the network setup step.

Receiving Ethernet packets

The received Ethernet packets are copied to the common receive queue of the according network handle. Ethernet packets stored in this queue can be read either by polling or via event driven notifications (WaitForSingleObject). The notification level can be set with the function xlNetSetNotification().

If an Ethernet packet is received by multiple measurement points or virtual ports on a switch segment, the XL Driver Library returns only one common event with a list of receive handles `XLrxHandle` instead of an event for each measurement point or virtual port.

This receive handle `XLrxHandle` is the same handle that was assigned by the application in xlNetConnectMeasurementPoint(), xlNetAddVirtualPort() and xlNetOpenVirtualPort() function calls in the network setup step.

7.2.4 Flowchart

Calling sequence

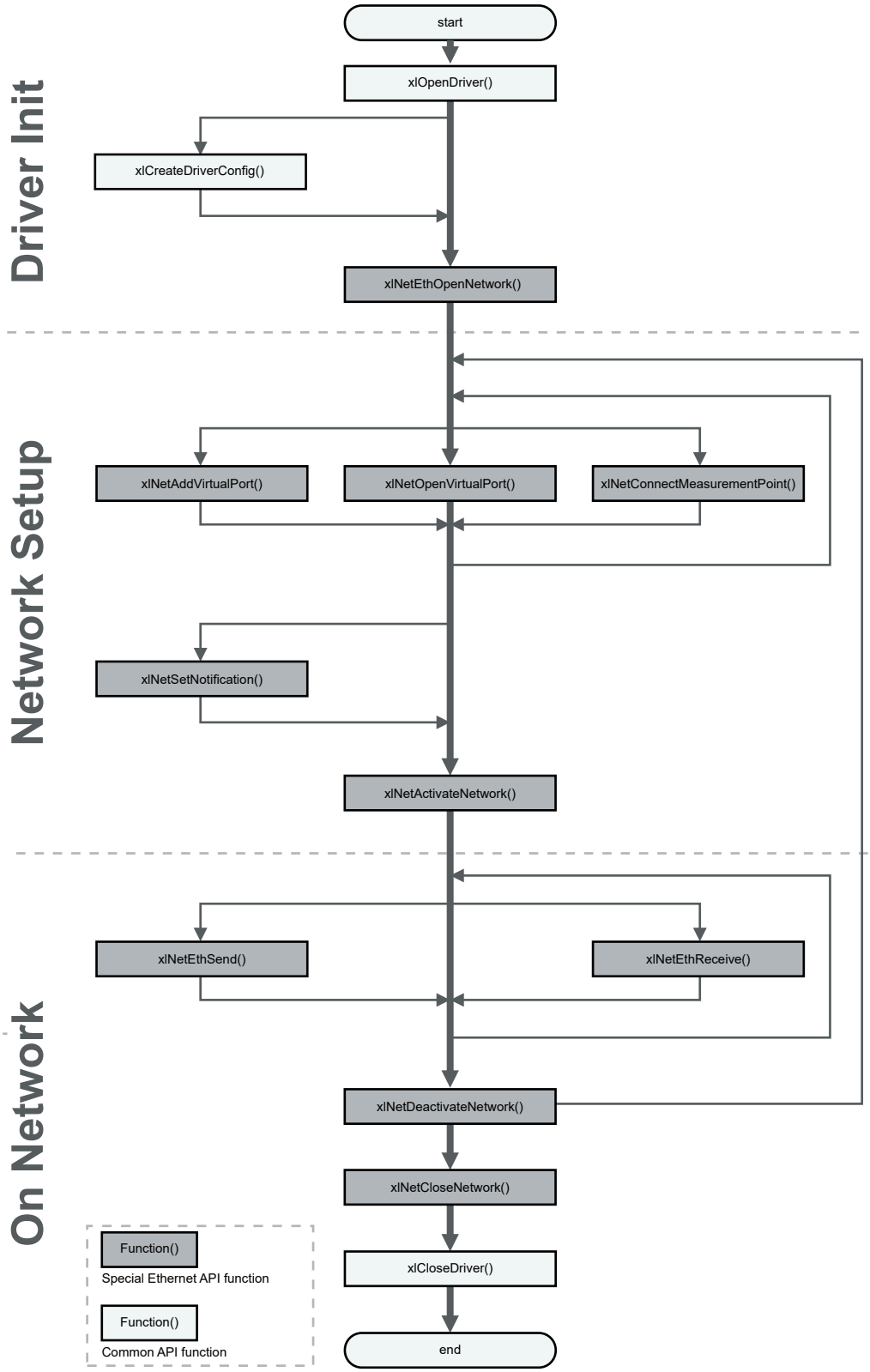


Figure 30: Function calls for Ethernet applications

7.2.5 Functions

7.2.5.1 xlNetActivateNetwork

Syntax

```
XLstatus xlNetActivateNetwork (
  XLnetworkHandle networkHandle
)
```

Description

Activates the network specified by the `networkHandle` and opens the receive network queue.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.2 xlNetAddVirtualPort

Syntax

```
XLstatus xlNetAddVirtualPort (
  XLnetworkHandle networkHandle,
  const char      *pSwitchName,
  const char *pVPortName,
  XLethPortHandle *pEthPortHandle,
  XLrxHandle rxHandle
)
```

Description

Adds a temporary virtual port to a switch segment with the given switch name on the network, specified by the `networkHandle`. The port name must be unique in the network. The number of virtual ports used is limited to one per Ethernet segment.

The port will not be persisted in the device configuration. The port will be added and immediately opened - no further xlNetOpenVirtualPort() function call is necessary. The port is closed automatically with the xlNetDeactivateNetwork() function call.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **pSwitchName**
  String for the switch segment name.
- **pVPortName**
  String for the virtual port name to add (must be unique).
- **pEthPortHandle**
  Return value to access the virtual port - for later function calls.
- **rxHandle**
  Application specific handle to identify the different events received.

Return value

Returns an error code (see section Error Codes on page 360).
Returns `XL_ERR_NOT_SUPPORTED` if more than one virtual port is used.

7.2.5.3 xlNetCloseNetwork

Syntax

```
XLstatus xlNetCloseNetwork (
  XLnetworkHandle networkHandle
```

```
)
```

Description

Close network specified by the `networkHandle` and delete the receive network queue.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.4 xlNetConnectMeasurementPoint

Syntax

```
XLstatus xlNetConnectMeasurementPoint (
  XLnetworkHandle networkHandle,
  const char      *pPortName,
  XLethPortHandle *pEthPortHandle,
  XLrxHandle      rxHandle
)
```

Description

Connect the application with a pre-defined measurement point on a network, specified by the `networkHandle`.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **pPortName**
  String for the measurement point name (must be unique).
- **pEthPortHandle**
  Return value to access the measurement point - for later function calls.
- **rxHandle**
  Application specific handle to identify the different events received.

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.5 xlNetDeactivateNetwork

Syntax

```
XLstatus xlNetDeactivateNetwork (
  XLnetworkHandle networkHandle
)
```

Description

Deactivates the network specified by the `networkHandle` and closes the receive network queue. Removes the temporary virtual ports.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.6 xlNetEthOpenNetwork

Syntax

```
XLstatus xlNetEthOpenNetwork (
  const char      *pNetworkName,
  XLnetworkHandle *pNetworkHandle,
  const char      *pAppName,
  unsigned int    accessType,
```

```
  unsigned int    queueSize
)
```

Description

Opens the network with the specified name and creates a receive network queue.

Input parameters

- **pNetworkName**:
  Name of the network to be opened. Available names can be retrieved by xlCreateDriverConfig().
- **pNetworkHandle**
  Return value to access the network - for later function calls.
- **pAppName**
  Optional application name that opens the network.
- **accessType**
  Defines if the network data will be transmitted reliable or (in future) unreliable for the corresponding uplink:
  - `XL_ACCESS_TYPE_RELIABLE`
    Always for USB uplink or TCP for Ethernet host uplink
  - `XL_ACCESS_TYPE_UNRELIABLE`
    Only for Ethernet uplink, means UDP transfers. (Not supported yet).
- **queueSize**
  Receive queue size in bytes. Value should be between:
  - `XL_ETH_RX_FIFO_QUEUE_SIZE_MIN`
    Minimum size of Ethernet receive queue: 64 Kbyte (64*1024).
  - `XL_ETH_RX_FIFO_QUEUE_SIZE_MAX`
    Maximum size of Ethernet receive queue: 64 Mbyte (64*1024*1024).

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.7 xlNetEthReceive

Syntax

```
XLstatus xlNetEthReceive (
  XLnetworkHandle networkHandle,
  XLethNetEvent   *pEventBuffer,
  unsigned int    *pRxHandleCount,
  XLrxHandle      *pRxHandle
)
```

Description

Retrieves one event from the receive event queue on a network, specified by the `networkHandle`.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **pEventBuffer**
  Buffer for a single Ethernet event (see section T_XL_NET_ETH_EVENT on page 157).
- **pRxHandleCount**
  - Input direction
    Maximum number of receive handles in the `pRxHandle` list.
  - Output direction:
    Actual number of receive handles within the `pRxHandle` list.
- **pRxHandle**
  List of the application specific receive handles set by xlNetAddVirtualPort(), xlNetOpenVirtualPort() or xlNetConnectMeasurementPoint().

Return value — Returns `XL_ERR_QUEUE_IS_EMPTY` if receive queue is empty.

7.2.5.8 xlNetEthRequestChannelStatus

Syntax

```
XLstatus xlNetEthRequestChannelStatus (
  XLnetworkHandle networkHandle
)
```

Description — Queries the current channel (real port) status on the whole network, specified by the `networkHandle`.

Sends an asynchronous request for the event that indicates the current status. Response event: XL_ETH_CHANNEL_STATUS.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().

Return value — Returns an error code (see section Error Codes on page 360).

7.2.5.9 xlNetEthSend

Syntax

```
XLstatus xlNetEthSend (
  XLnetworkHandle    networkHandle,
  XLethPortHandle    ethPortHandle,
  XLuserHandle       userHandle,
  const T_XL_ETH_DATAFRAME_TX* pEthTxFrame
)
```

Description — Transmits an Ethernet packet on the virtual port or measurement point which is indicated with the `ethPortHandle` on the network, specified by the `networkHandle`.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **ethPortHandle**
  Handle to access the virtual port or measurement point.
- **userHandle**
  The handle is created by the application and is used for the assignment of `T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ACK` and `T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ERROR` events.
- **pEthTxFrame**
  The Ethernet packet to be sent (see section T_XL_ETH_DATAFRAME_TX on page 183).

Return value — Returns `XL_ERR_QUEUE_IS_FULL` if transmit queue is full.

7.2.5.10 xlNetEthTc10WakeUpCmd

Syntax

```
xlNetEthTc10WakeUpCmd(
  XLnetworkHandle networkHandle,
  XLethPortHandle ethPortHandle
)
```

Description

A wake-up command is sent to the PHY (physical interface). Then, the PHY resolves the WUP/WUR TC10 symbols over the MDI. The link partner is also woken up so that a link can be established.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **ethPortHandle**
  Handle to access the measurement point.

Return value

Returns an error code (see section Error Codes on page 360).
Returns `XL_ERR_NOT_SUPPORTED` if the PHY does not support the TC10 feature (see the VN5000 manual).

7.2.5.11 xlNetFlushReceiveQueue

Syntax

```
XLstatus xlNetFlushReceiveQueue (
XLnetworkHandle networkHandle
)
```

Description

Flushes the application receive queue.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.12 xlNetOpenVirtualPort

Syntax

```
XLstatus xlNetOpenVirtualPort (
  XLnetworkHandle networkHandle,
  const char      *pVPortName,
  XLethPortHandle *pEthPortHandle,
  XLrxHandle      rxHandle
)
```

Description

Open a pre-defined virtual port on the network, specified by the `networkHandle`. The number of virtual ports used is limited to one per Ethernet segment.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **pVPortName**
  String for the virtual port name.
- **pEthPortHandle**
  Return value to access the virtual port - for later function calls.
- **rxHandle**
  Application specific handle to identify the different events received.

Return value

Returns an error code (see section Error Codes on page 360).
Returns `XL_ERR_NOT_SUPPORTED` if more than one virtual port is used.

7.2.5.13 xlNetReleaseMACAddress

Syntax

```
XLstatus xlNetReleaseMACAddress (
```

```
  XLnetworkHandle            networkHandle,
  const T_XL_ETH_MAC_ADDRESS *pMACAddress
)
```

Description

Release a former requested ETH MAC address from the pool of all application-reservable MAC addresses in the network, specified by the `networkHandle`.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **pMACAddress**
  Pointer to structure with ETH MAC address to release.

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.14 xlNetRequestMACAddress

Syntax

```
XLstatus xlNetRequestMACAddress (
  XLnetworkHandle       networkHandle,
  T_XL_ETH_MAC_ADDRESS  *pMACAddress
)
```

Description

Request and lock a globally-unique ETH MAC address from pool of application-reservable MAC addresses in the network, specified by the `networkHandle`.

The pool of the network is the union of all the MAC-pools of the devices that are part of the network. When the `networkHandle` is closed - or when the application terminates - the MAC address is implicitly released. While a MAC address is locked, no other application can request the address.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **pMACAddress**
  Pointer structure to retrieve ETH MAC address.

Return value

Returns an error code (see section Error Codes on page 360).

7.2.5.15 xlNetSetNotification

Syntax

```
XLstatus xlNetSetNotification (
  XLnetworkHandle networkHandle,
  XLhandle        *pHandle,
  int             queueLevel
)
```

Description

Sets up an event to notify the application if there are messages in the Ethernet network receive queue on the network, specified by the `networkHandle`.

Note that the event is triggered only once, when the `queueLevel` is reached.
An application should read all available messages by xlNetEthReceive() to be sure to re enable the event.

Input parameters

- **networkHandle**
  Handle to access the network retrieved by xlNetEthOpenNetwork().
- **pHandle**
  Generated handle.

- **queueLevel**
  Specifies the number of bytes that triggers the event.

Return value — Returns an error code (see section Error Codes on page 360).

7.2.6 Structs

7.2.6.1 T_XL_ETH_MAC_ADDRESS

Syntax

```
#define XL_ETH_MACADDR_OCTETS 6

typedef struct {
  unsigned char address[XL_ETH_MACADDR_OCTETS];
} T_XL_ETH_MAC_ADDRESS;
```

Description

This struct defines the composition of a MAC address.

Parameters

- **address**
  Array of bytes with the MAC address.

7.2.7 Events

7.2.7.1 T_XL_NET_ETH_CHANNEL_STATUS

Syntax

```
typedef T_XL_ETH_CHANNEL_STATUS
T_XL_NET_ETH_CHANNEL_STATUS;
```

Description

This event is generated each time the link information changes.

Parameters

For a description of the structure members refer to T_XL_ETH_CHANNEL_STATUS.

7.2.7.2 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX

Syntax

```
typedef T_XL_NET_ETH_DATAFRAME_RX
T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX;
```

Description

This event is indicated to the application each time an Ethernet packet has been successfully received from the network on a measurement point.

Tag

`XL_ETH_EVENT_TAG_FRAMERX_MEASUREMENT`

Parameters

For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX.

7.2.7.3 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX_ERROR

Syntax

```
typedef T_XL_NET_ETH_DATAFRAME_RX_ERROR
T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX_ERROR;
```

Description

This event is indicated to the application each time an erroneous Ethernet packet has been received from the network on a measurement point.

Tag

`XL_ETH_EVENT_TAG_FRAMERX_ERROR_MEASUREMENT`

Parameters

For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX_ERROR.

7.2.7.4 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX

Syntax

```
typedef T_XL_NET_ETH_DATAFRAME_RX
T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX;
```

Description

This event is indicated to the application each time an Ethernet packet has been successfully from a measurement point sent to the network. It is neither a delivery confirmation from the receiver, nor a guarantee that the intended recipient will receive that frame. It currently has an identical layout to the Rx packet; the different name is merely for a better understanding.

Tag

`XL_ETH_EVENT_TAG_FRAMETX_MEASUREMENT`

Parameters

For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX.

7.2.7.5 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX_ERROR

Syntax

```
typedef T_XL_NET_ETH_DATAFRAME_RX_ERROR
T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX_ERROR
```

Description

This event is indicated to the application each time an Ethernet packet has not been successfully sent to the network from a measurement point.

Tag

`XL_ETH_EVENT_TAG_FRAMETX_ERROR_MEASUREMENT`

Parameters

For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX_ERROR.

7.2.7.6 T_XL_NET_ETH_DATAFRAME_RX

Syntax

```
typedef struct s_xl_net_eth_dataframe_rx {
  unsigned int       frameDuration;
  unsigned short     dataLen;
  unsigned char      reserved1;
  unsigned char      reserved2;
  unsigned int       errorFlags;
  unsigned int       reserved3;
  unsigned int       fcs;
  unsigned char      destMAC[XL_ETH_MACADDR_OCTETS];
  unsigned char      sourceMAC[XL_ETH_MACADDR_OCTETS];
  T_XL_ETH_FRAMEDATA frameData;
} T_XL_NET_ETH_DATAFRAME_RX;
```

Description

Structure describing the network Ethernet packets that can be received (including Tx frames).

Tag

`XL_ETH_EVENT_TAG_FRAMERX_SIMULATION`

Parameters

- **frameDuration**
  Transmit duration of the frame, given in nanoseconds.
- **dataLen**
  Combined size of etherType and payload in bytes. This specifies the size actually used, not the maximum size of the struct.
- **reserved1**
  Not being used, ignore.
- **reserved2**
  Not being used, ignore.
- **errorFlags**

  In an Rx event the bits indicate following errors:

  - `XL_ETH_NETWORK_RX_ERROR_INVALID_LENGTH`
    Bit 0: Invalid length error. Set when the receive frame has an invalid length as defined by IEEE802.3
  - `XL_ETH_NETWORK_RX_ERROR_INVALID_CRC`
    Bit 1: CRC error. Set when frame is received with CRC-32 error but valid length
  - `XL_ETH_NETWORK_RX_ERROR_PHY_ERROR`
    Bit 2: Corrupted receive frame caused by a PHY error
  - `XL_ETH_NETWORK_RX_ERROR_MACADDR_ERROR`
    Bit 3: Invalid source or destination MAC address

In an Tx event the bits indicate following errors:

- `XL_ETH_NETWORK_TX_ERROR_NO_LINK`
  Bit 0: No Link
- `XL_ETH_NETWORK_TX_ERROR_PHY_NOT_CONFIGURED`
  Bit 1: PHY not yet configured
- `XL_ETH_NETWORK_TX_ERROR_PHY_BRIDGE_ENABLED`
  Bit 2: PHY bypass activated
- `XL_ETH_NETWORK_TX_ERROR_CONVERTER_RESET`
  Bit 3: RGMII converter in reset
- `XL_ETH_NETWORK_TX_ERROR_INVALID_LENGTH`
  Bit 4: Invalid length error. Set when the frame has an invalid length as defined by IEEE802.3
- `XL_ETH_NETWORK_TX_ERROR_INVALID_CRC`
  Bit 5: CRC error. Set when frame is transmitted with CRC-32 error but valid length
- `XL_ETH_NETWORK_TX_ERROR_MACADDR_ERROR`
  Bit 6: Invalid source or destination MAC address

- **reserved3**
  Not being used, ignore.
- **fcs**
  Frame Check Sequence as received from network.
- **destMAC**
  Destination MAC address.
- **sourceMAC**
  Source MAC address.
- **frameData**
  section T_XL_ETH_FRAME on page 175

7.2.7.7 T_XL_NET_ETH_DATAFRAME_RX_ERROR

Syntax

```
typedef struct s_xl_net_eth_dataframe_rx_error {
  unsigned int       frameDuration;
  unsigned int       errorFlags;
  unsigned short     dataLen;
  unsigned char      reserved1;
  unsigned char      reserved2;
  unsigned int       reserved3[2];
  unsigned int       fcs;
  unsigned char      destMAC[XL_ETH_MACADDR_OCTETS];
  unsigned char      sourceMAC[XL_ETH_MACADDR_OCTETS];
  T_XL_ETH_FRAMEDATA frameData;
} T_XL_NET_ETH_DATAFRAME_RX_ERROR;
```

Description

Structure describing the erroneous network Ethernet packets that can be received (including erroneous Tx frames).

Tag

`XL_ETH_EVENT_TAG_FRAMERX_ERROR_SIMULATION`

Parameters

- **frameDuration**
  Transmit duration of the Ethernet packet, in nanoseconds.

- **errorFlags**
  The meaning of the flags differ between an Rx event and a Tx event.

  In an Rx event, the bits indicate following errors:

  `XL_ETH_NETWORK_RX_ERROR_INVALID_LENGTH`
  Bit 0: Invalid length error. Set when the receive frame has an invalid length as defined by IEEE802.3.

  `XL_ETH_NETWORK_RX_ERROR_INVALID_CRC`
  Bit 1: CRC error. Set when frame is received with CRC-32 error but valid length.

  `XL_ETH_NETWORK_RX_ERROR_PHY_ERROR`
  Bit 2: Corrupted receive frame caused by a PHY error.

  `XL_ETH_NETWORK_RX_ERROR_MACADDR_ERROR`
  Bit 3: Invalid source or destination MAC address.

  In a Tx event, the bits indicate following errors:

  `XL_ETH_NETWORK_TX_ERROR_NO_LINK`
  Bit 0: No Link.

  `XL_ETH_NETWORK_TX_ERROR_PHY_NOT_CONFIGURED`
  Bit 1: PHY not yet configured.

  `XL_ETH_NETWORK_TX_ERROR_PHY_BRIDGE_ENABLED`
  Bit 2: PHY bypass activated.

  `XL_ETH_NETWORK_TX_ERROR_CONVERTER_RESET`
  Bit 3: RGMII converter in reset.

  `XL_ETH_NETWORK_TX_ERROR_INVALID_LENGTH`
  Bit 4: Invalid length error. Set when the frame has an invalid length as defined by IEEE802.3.

  `XL_ETH_NETWORK_TX_ERROR_INVALID_CRC`
  Bit 5: CRC error. Set when frame is transmitted with CRC-32 error but valid length.

  `XL_ETH_NETWORK_TX_ERROR_MACADDR_ERROR`
  Bit 6: Invalid source or destination MAC address.

- **dataLen**
  Combined size of the *etherType* and `payload` in bytes. This specifies the size actually used, not the maximum size of the struct.

- **reserved1**
  Not being used, ignore.

- **reserved2**
  Not being used, ignore.

- **reserved3[2]**
  Not being used, ignore

- **fcs**
  Frame Check Sequence, as received from network.

- **destMAC[XL_ETH_MACADDR_OCTETS]**
  Destination MAC address.

- **sourceMAC[XL_ETH_MACADDR_OCTETS]**
  Source MAC address.
- **frameData**
  See section T_XL_ETH_FRAME on page 175.

7.2.7.8 T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ACK

Syntax

```
typedef T_XL_NET_ETH_DATAFRAME_RX
T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ACK;
```

Description

This event is indicated to the application each time an Ethernet packet has been successfully sent to the network from a virtual port. It is neither a delivery confirmation from the receiver, nor a guarantee that the intended recipient will receive that frame. It currently has an identical layout to the Rx packet; the different name is merely for a better understanding.

Tag

`XL_ETH_EVENT_TAG_FRAMETX_ACK_SIMULATION`

Parameters

For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX.

7.2.7.9 T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ERROR

Syntax

```
typedef T_XL_NET_ETH_DATAFRAME_RX_ERROR
T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ERROR;
```

Description

This event is indicated to the application each time an Ethernet packet has not been successfully sent to the network from a virtual port.

Tag

`XL_ETH_EVENT_TAG_FRAMETX_ERROR_SIMULATION`

Parameters

For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX_ERROR.

7.2.7.10 T_XL_NET_ETH_EVENT

Syntax

```
typedef unsigned short XLethEventTag;

typedef struct s_xl_net_eth_event {
  unsigned int   size;
  XLethEventTag  tag;
  unsigned short channelIndex;
  unsigned int   userHandle;
  unsigned short flagsChip;
  unsigned short reserved;
  XLuint64       reserved1;
  XLuint64       timestampSync;

union s_xl_eth_net_tag_data {
    unsigned char rawData[XL_ETH_EVENT_SIZE_MAX];
    T_XL_NET_ETH_DATAFRAME_RX frameSimRx;
    T_XL_NET_ETH_DATAFRAME_RX_ERROR frameSimRxError;
    T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ACK frameSimTxAck;
    T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ERROR frameSimTxError;
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX frameMeasureRx;
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX_ERROR frameMeas-
ureRxError;
```

```
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX frameMeasureTx;
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX_ERROR frameMeas-
ureTxError;
    T_XL_NET_ETH_CHANNEL_STATUS channelStatus;
  } tagData;
} T_XL_NET_ETH_EVENT;
```

Description

Structure describing the network Ethernet events that can be received (including Tx events).

Parameters

- **size**
  Size of the complete Ethernet event, including header and payload data.
- **tag**
  Specifies the structure that is applied to `tagData`,
  e. g. XL_ETH_EVENT_TAG_FRAMERX_MEASUREMENT.
- **channelIndex**
  Logical channel number where this event originated or is target to.
- **userHandle**
  Application-specific handle that may be used to link associated events, e. g. a transmit confirmation to the original send request. Not used (set to 0) for indications not related to a request.
- **flagsChip**

  The lower 8 bit contain chip information:
  Bit 0: `XL_ETH_CONNECTOR_RJ45`
  Bit 1: `XL_ETH_CONNECTOR_DSUB`
  Bit 2: `XL_ETH_PHY_IEEE`
  Bit 3: `XL_ETH_PHY_BROADR`
  Bit 4: `XL_ETH_FRAME_BYPASSED`
  Bit 5..7: unused

  The upper 8 bit contain special flags:
  Bit 8: `XL_ETH_QUEUE_OVERFLOW`

  Not all events generated by the device could be indicated to the application.

  Bit 9..14: unused
  Bit 15: `XL_ETH_BYPASS_QUEUE_OVERFLOW`

  Indicates that one or more received packets could not be sent to the opposite bus in MAC bypass mode.
- **reserved**
  Not being used, ignore.
- **reserved1**
  Not being used, ignore.
- **timestampSync**
  Synchronized time stamp with 1 ns resolution (PC → device) and an accuracy of 8 µs. Time synchronization is applied if enabled in Vector Hardware Manager time synchronization settings.
- **tagData**
  See structures on page 154 ... page 153 for further details.

7.2.8 Application Examples

7.2.8.1 xlNetEthDemo

General Information

Description

The example xlNetEthDemo (further noted as app) demonstrates how to transmit/receive Ethernet packets using network-based mode. The app structure contains a small command line interface controlled by keyboard commands.

Starting the example

Before running the app, an Ethernet device must be connected that operates in network-based mode. Additionally, at least one network must be configured on the device with the Vector Hardware Manager.

Example Test Case

Startup

At execution, the app searches for all attached devices, whose drivers have been installed. Therefore, information shown in the Vector Hardware Manager will be available through this Demo too (see xlCreateDriverConfig()).

As soon as the app obtains hardware information from Vector Driver, you will see following printout:

```
-----------------------------------------------------------
- xlNetEthDemo - Test Application for XL Family Driver API -
-              (C) 2020 Vector Informatik GmbH              -
-----------------------------------------------------------
Found 4 devices
[0] Found VN5640:1(000002) device
[1] Found VN5640:2(005247) device
[2] Found Virtual CAN Bus:1(000000) device
[3] Found vTSService:1(000000) device
Choose Device for showing Hardware Configuration:
```

Figure 31: Found Vector hardware

Configurations of all devices with Ethernet ports can be printed out in console (see command <w>) - as is shown in the following picture. Hardware information (transceiver, hardware ports etc.) is related to the information about a device. However, this set of information about devices does not contain direct information about available networks on the selected device, but network information is a separate set of information.

```
Device : VN5640
-------------------------------------------------------------------------------
VN5640 :  1 (000002)              Hardware Configuration
-------------------------------------------------------------------------------
HwNr|HwCh|Transceiver              |MAC               |Link State
----+----+------------------------+------------------+----------------------
   0|   0|ETHmodule BCM89811      |00:16:81:00:d0:10|1000BASE-T1 100 MBit (S1), DSub
   0|   1|ETHmodule BCM89811      |00:16:81:00:d0:11|1000BASE-T1 100 MBit (S1), DSub
   0|   2|ETHmodule TJA1100       |00:16:81:00:d0:12|Link down
   0|   3|ETHmodule TJA1100       |00:16:81:00:d0:13|Link down
   0|   4|ETHmodule BCM89811      |00:16:81:00:d0:14|Link down
   0|   5|ETHmodule BCM89811      |00:16:81:00:d0:15|Link down
   0|   6|ETHmodule TJA1100       |00:16:81:00:d0:16|Link down
   0|   7|ETHmodule TJA1100       |00:16:81:00:d0:17|Link down
   0|   8|ETHmodule TJA1100       |00:16:81:00:d0:18|Link down
   0|   9|ETHmodule TJA1100       |00:16:81:00:d0:19|Link down
   0|  10|ETHmodule TJA1100       |00:16:81:00:d0:1a|Link down
   0|  11|ETHmodule TJA1100       |00:16:81:00:d0:1b|Link down
----+----+------------------------+------------------+----------------------
```

Figure 32: Example of device hardware configuration

As depicted below, the list of all networks, from all connected devices, contains information about each network: name, segment(s), segment name, segment type, measurement points (MPs) and virtual ports (VPs) connected to corresponding segment.

Adding a VP

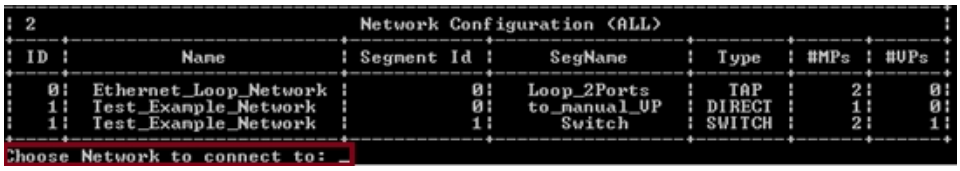


Figure 33: Network and segment configuration information

The app will at this point, for demonstration purposes, manually add a virtual port with the name `VP_manually_added`. In order to do so, the user firstly needs to choose a network from the list by typing the ID value shown in the first column (see figure above). In the example case, ID = 1 is chosen (`Test_Example_Network`). After choosing the network, only the configuration of the chosen network will be displayed – showing segments (ID, name, type) that are within the chosen network. At this point the user needs to choose the segment to which the VP will be manually added.

If the segment is of type DIRECT and it already contains a VP, or of type TAP, the app shows an error that the VP has not been added, as the chosen segment is already full. If the user selects a segment that is not full, manually adding a VP will succeed. Subsequently, the app displays the change in the configuration of the chosen network. In order to detect the added VP, the app calls xlCreateDriverConfig() again after adding the VP, so it gets new up-to-date snapshot of the network configuration.

Main menu

At this point, the initialization phase is over and the interaction menu is shown. The user is prompted by the main menu about further actions (see section Keyboard Commands on page 162). The <t> and <T> keys trigger the transmission of Ethernet packets via MP or VP with a dummy IP and UDP header. The user can set the destination MAC and IP with the <m> and <i> keys. By pressing <G>, the user can acquire a unique source MAC from the pool of MAC addresses available in the device. The source IP of the Ethernet packetis 192.168.0.x, where x is the port ID. MP port as a sender is chosen by typing the port ID or by <+> or <->. VP as a sender is chosen by typing <*> or </>. After sending the Ethernet packet, received events will be processed and a corresponding message is displayed (see example function handleEthernetEvent).

Events

The testbed setup for this example is depicted below. At this point, the user has chosen to send an Ethernet packet from `VP_manually_added`. The resulting Ethernet packet path is marked with arrows starting with number 1.

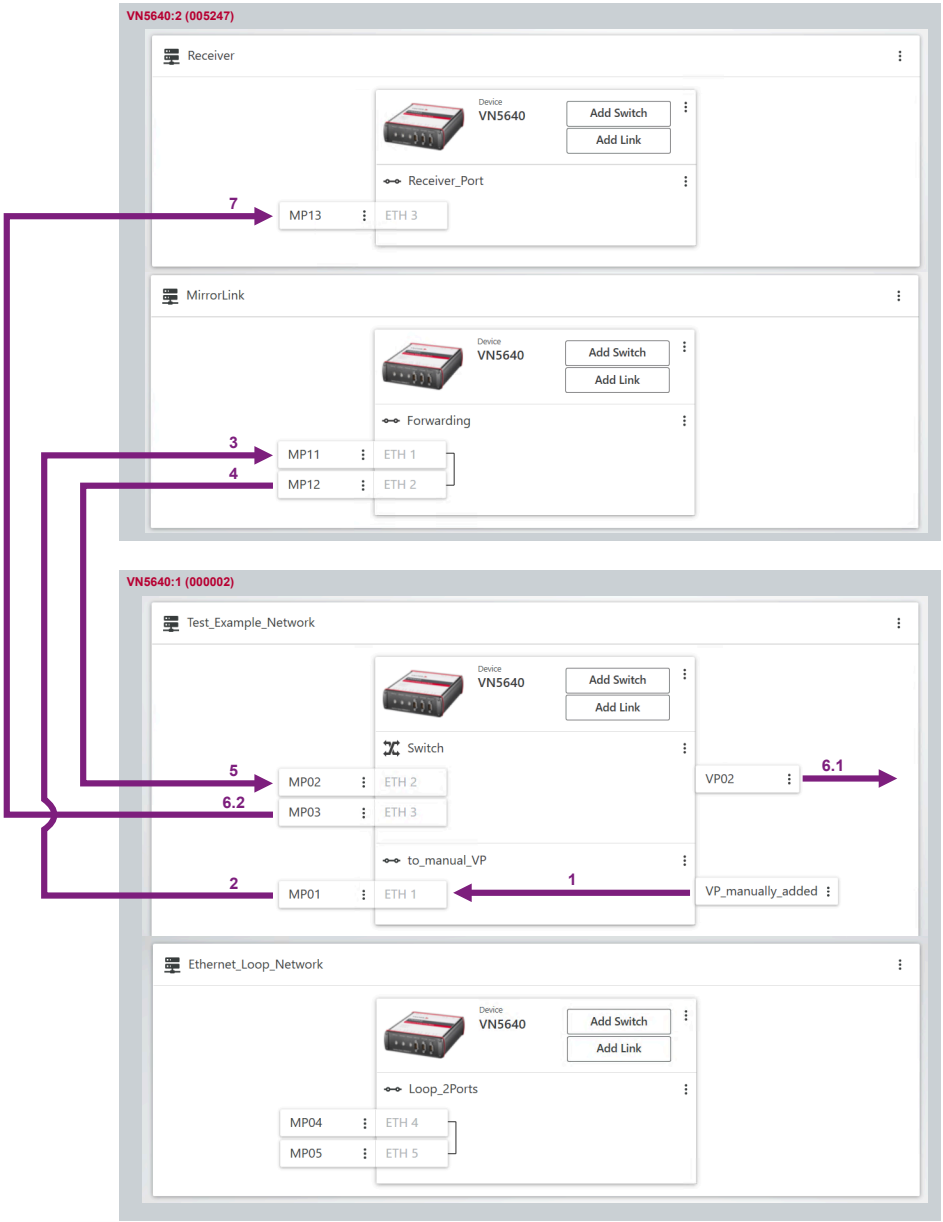


Figure 34: Testbed used for the example and Frame Path from VP_manually_added port

The app prints the following event output with the `rxHandle` that the app assigned in parenthesis.

- `Received TX Ack on VP_manuallyAdded (4)`
  Confirmation that Ethernet packet was sent (step 1 in figure above).
- `Received TX on MP01 (0)`
  MP01 forwards the Ethernet packet (step 2).
- `Received RX on MP02 (2)`
  MP01 and MP11 are connected by cable (violett connection) and thus MP11 receives the Ethernet packet (step 3) and MP12 forwards the Ethernet packet (step 4) to MP02 (step 5). We only receive an event form MP02 but not from MP11 and MP12, because the app only opened the Test_Example_Network and not the MirrorLink network.

- `Received RX on VP02 (1)`
  As MP02, MP03 and VP02 belong to the same segment, the switch forwards the Ethernet packet from MP02 to VP02 (step 6.1).
- `Received TX on MP03 (3)`
  The switch also forwards the Ethernet packet to MP03 (step 6.2), which transmits it to the other end of cable (step 7).

Keyboard Commands

The running app can be controlled by the following keyboard commands:

Key	Command
<1>…(max MPs)	Select specific MP in the chosen network by giving its ID (`1 < MP_ID < 9`).
<+>	Select next MP in the chosen network.
<->	Select previous MP in the chosen network.
<t>	Transmit Ethernet packet from currently chosen MP.
<*>	Select next VP in the chosen network.
</>	Select previous VP in the chosen network.
<T>	Transmit Ethernet packet from currently chosen VP.
<s>	Check current link status on all channels in the chosen network.
<p>	Print basic information from/about last received Ethernet packet in the chosen network Info: {SrcMAC, DstMAC, SrcIP, DstIP, Payload} .
<G>	Request to obtain MAC address from the device MAC address pool.
<R>	Release previously requested MAC address from the device MAC address pool.
<m>	Set destination MAC address.
<i>	Set destination IP address.
w>	Show hardware configuration of the chosen device (not related to chosen network and segment) .
<h> / <?>	Show/print main menu.
<q>	Quit xlNetEthDemo app.

7.3 Channel Based Access Mode

Note
The channel-based API will continue to be supported as a legacy API on devices that previously supported it.

For new projects, we recommend the network-based access mode API to access Ethernet networks (see section Network Based Access Mode on page 136).

Description

The **XL Driver Library** enables the development of Ethernet applications for supported Vector devices see section System Requirements on page 25 and also section Device Support on page 133.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel parameters can be changed/configured
- Ethernet packets can be received and transmitted
- bypasses can be set and cleared

Without init access

- Ethernet packets can be received and transmitted
- channel parameters can be read

The specific Ethernet functions of the **XL Driver Library** do not wait for completion on a requested operation (if not otherwise specified). Instead, an event is generated as soon as the operation has been completed if necessary.

Reference
See the flowchart on the next page for all available functions and the according calling sequence.

7.3.1 Flowchart

Calling sequence

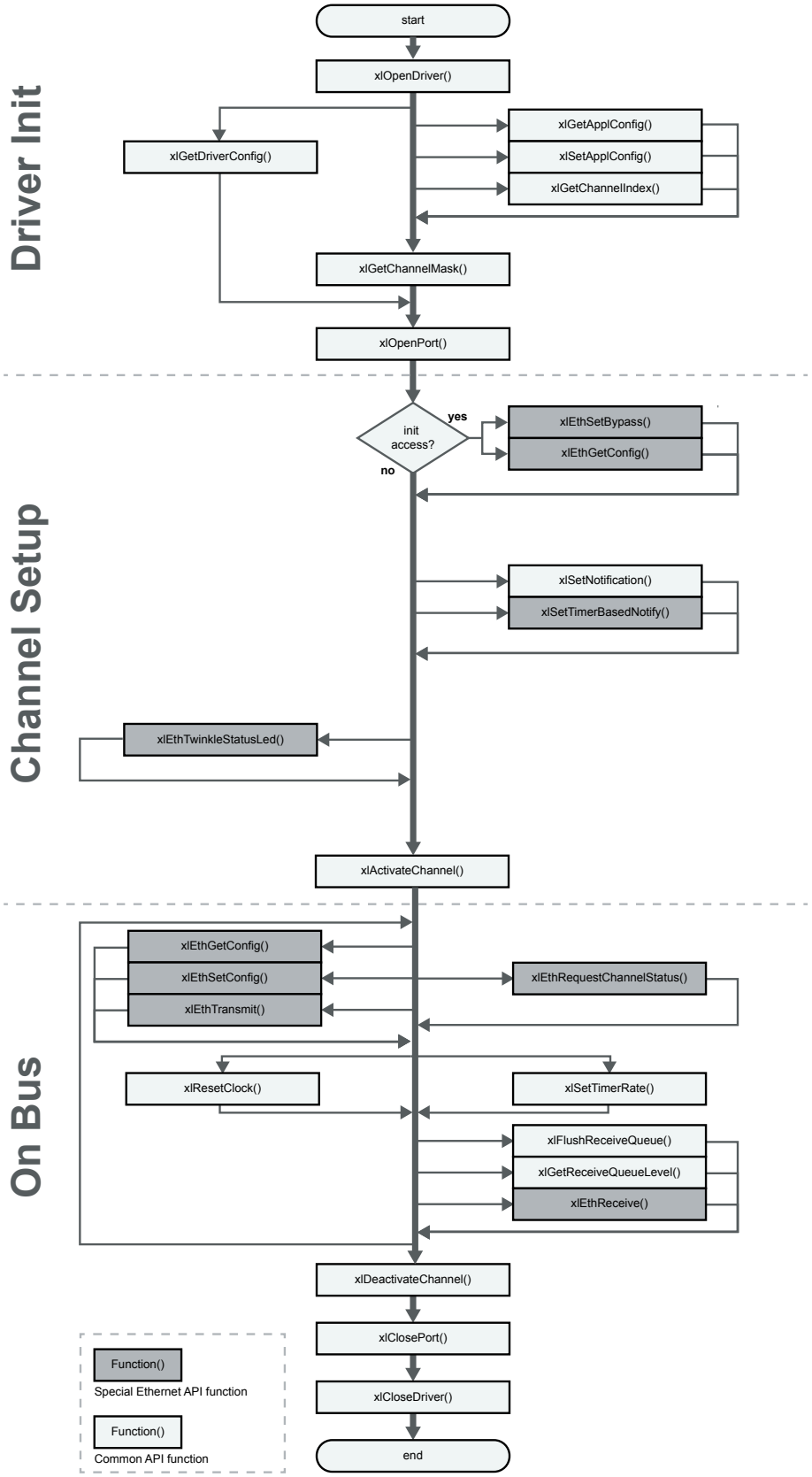


Figure 35: Function calls for Ethernet applications

7.3.2 Functions

7.3.2.1 xlEthSetConfig

Syntax

```
XLstatus xlEthSetConfig (
  XLportHandle  portHandle,
  XLaccess              accessMask,
  XLuserHandle          userHandle,
  const T_XL_ETH_CONFIG *config
)
```

Description

Configures basic Ethernet settings. The result of the operation is reported via a T_XL_ETH_CONFIG_RESULT event. This function needs **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **userHandle**
  The handle is created by the application and is used for the event assignment.
- **config**
  Ethernet configuration structure (see section T_XL_ETH_CONFIG on page 172).

Return value

Returns an error code (see section Error Codes on page 360).

7.3.2.2 xlEthGetConfig

Syntax

```
XLstatus xlEthGetConfig (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  XLuserHandle    userHandle,
  T_XL_ETH_CONFIG *config
)
```

Description

Reads the basic Ethernet settings from the device that was configured last. Note that the device does not keep those settings after a restart. This is a synchronous operation.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **userHandle**
  The handle is created by the application and is used for the event assignment.

- **config**
  Ethernet configuration structure (see section T_XL_ETH_CONFIG on page 172).

Return value — Returns an error code (see section Error Codes on page 360).

7.3.2.3 xlEthSetBypass

Syntax

```
XLstatus xlEthSetBypass (
  XLportHandle portHandle,
  XLaccess     accessMask,
  XLuserHandle userHandle,
  unsigned int mode
)
```

Description

The function sets the bypass mode for the channel specified in `accessMask`. For the given channel **init access** is required; for the bypass partner channel **init access** is required whenever the channel is currently used by any application.

When the PHY bypass mode is set, two Ethernet channels are internally hard-wired. This requires compatible settings (i. e. same speed, same duplex mode). Sending on either channel is not supported in this mode. The main purpose of this mode is to convert the physical layer from IEEE802.3 to BroadR-Reach and vice versa, with minimal impact on latency. The PHY bypass mode can also be used for monitoring with low latencies.

When in MAC bypass mode, the device connects two channels on Ethernet packet level using a store-and-forward mechanism. In this mode, channels of any mode can be connected, and sending is possible for applications as well. However, the latency imposed by the device is higher than in PHY bypass mode.

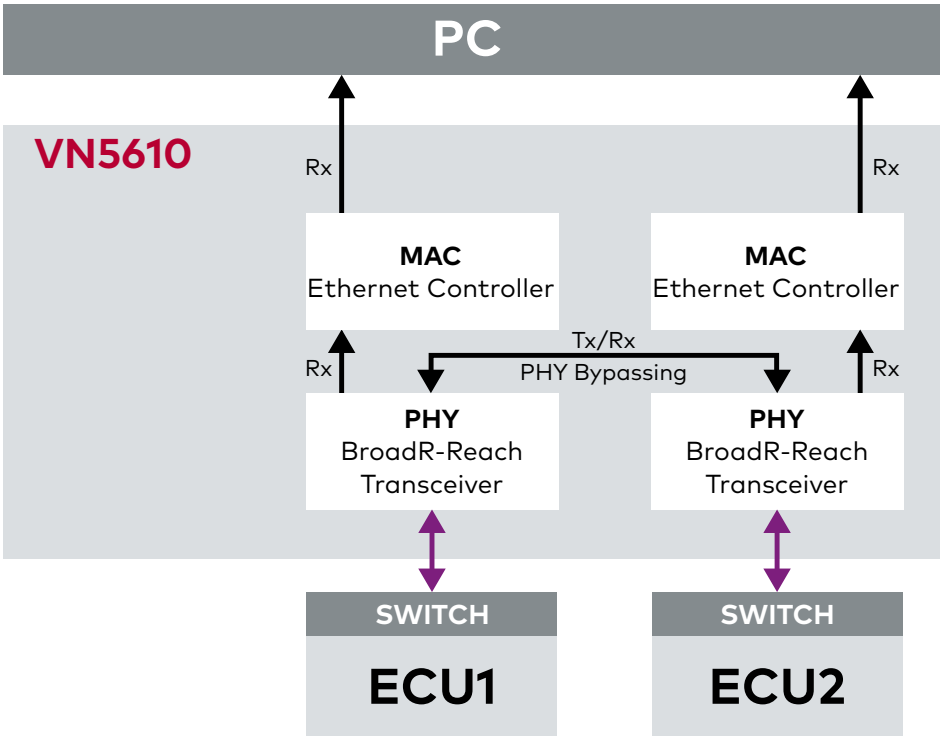


Figure 36: PHY bypassing in VN5610

In MAC bypass mode, the channels may be used as usual, including sending - the device will send that data as soon as there is a gap in the bypassed packet stream.

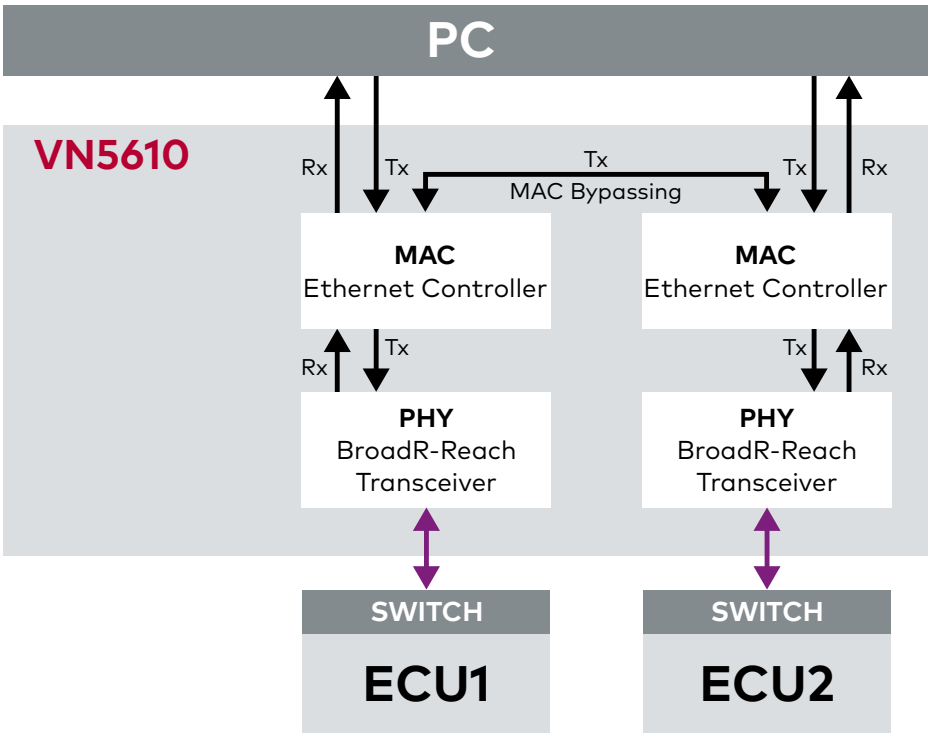


Figure 37: MAC bypassing in VN5610

Note
Since this mode does not require compatible settings of the Ethernet channels and additional data may also be sent by the application, an overflow of the internal switch queues could occur. In this case data may be lost!

Note
If the bypass is activated, we recommend calling this function before activating the channel, thus the hardware can immediately activate the channel bypass after activation and configuration of the hardware. Otherwise packets sent by a remote device immediately after link establishment may not be forwarded.

This is a synchronous operation, i. e. the requested bypass mode is active upon return from this function.

The current bypass state can be requested with xlGetDriverConfig(). The value is stored in the XLbusParams structure.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **userHandle**
  The handle is created by the application and is used for the event assignment.

- **mode**
  `XL_ETH_BYPASS_INACTIVE` (Default)
  `XL_ETH_BYPASS_PHY`
  `XL_ETH_BYPASS_MACCORE`

Return value — Returns an error code (see section Error Codes on page 360).

7.3.2.4 xlEthTransmit

Syntax

```
XLstatus xlEthTransmit(
  XLportHandle               portHandle,
  XLaccess                   accessMask,
  XLuserHandle               userHandle,
  const T_XL_ETH_DATAFRAME_TX *data
)
```

Description — Transmits an Ethernet packet on the channel which is indicated in `accessMask`.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **userHandle**
  The handle is created by the application and is used for the event assignment.
- **data**
  The Ethernet packet to be sent. Source MAC address can either be set by the application or be automatically inserted by the hardware. In order to use the Source MAC address as given in this request, the flags member has to contain the following values:
  `XL_ETH_DATAFRAME_FLAGS_USE_SOURCE_MAC`

  Note: No padding is executed. Data to be transmitted will not be extended to its minimal size.

Return value — Returns an error code (see section Error Codes on page 360).

7.3.2.5 xlEthReceive

Syntax

```
XLstatus xlEthReceive (
  XLportHandle   portHandle,
  T_XL_ETH_EVENT *eventBuffer
)
```

Description — Retrieves one event from the event queue. This operation is synchronous.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **eventBuffer**
  Buffer for a single Ethernet event (see section T_XL_ETH_EVENT on page 175).

Return value — Returns an error code (see section Error Codes on page 360).

7.3.2.6 xlEthTwinkleStatusLed

Syntax

```
XLstatus xlEthTwinkleStatusLed(
  XLportHandle portHandle,
  XLaccess     accessMask,
  XLuserHandle userHandle
)
```

Description — Twinkle the Status LED from the Vector device for a short period of time. For each device whose status LED should twinkle, at least one channel bit has to be set in the `accessMask`.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **userHandle**
  The handle is created by the application and is used for the event assignment.

Return value — Returns an error code (see section Error Codes on page 360).

7.3.3 Structs

7.3.3.1 XLdriverConfig

Syntax

```
typedef struct {
  unsigned int busType;
  union {
    struct {
    [...]
    } can;
    [...]
    struct {
      unsigned char macAddr[6];
      unsigned char connector;
      unsigned char phy;
      unsigned char link;
      unsigned char speed;
      unsigned char clockMode;
      unsigned char bypass;
    } ethernet;
    unsigned char raw[32];
  }data;
} XLbusParams;

typedef struct s_xl_channel_config {
  [...]
  XLbusParams busParams;
  [...]
} XLchannelConfig;

typedef struct s_xl_driver_config {
  [...]
  XLchannelConfig channel[XL_CONFIG_MAX_CHANNELS];
} XLdriverConfig;
```

Description

The global function xlGetDriverConfig()fills a driver configuration info structure. One of the channel-specific values returned is a parameter of type XLbusParams named `busParams` which contains bus-specific status information.

Parameters

- **macAddr**
  Device MAC address assigned to this channel.
- **connector**
  Device connector currently in use.

  `XL_ETH_STATUS_CONNECTOR_RJ45`
  RJ45 connector.

  `XL_ETH_STATUS_CONNECTOR_DSUB`
  D-SUB connector.

- **phy**
  Physical layer currently in use:

  `XL_ETH_STATUS_PHY_UNKNOWN`
  Currently unknown (e. g. if link is down).

  `XL_ETH_STATUS_PHY_IEEE_802_3`
  Ethernet according to IEEE 802.3u or 802.3ab.

  `XL_ETH_STATUS_PHY_BROADR_REACH`
  OPEN Alliance BroadR-Reach® physical layer.

- **link**
  Link status of this channel:

  `XL_ETH_STATUS_LINK_UNKNOWN`
  Link status could not be determined (e. g. if connection to device is lost).

  `XL_ETH_STATUS_LINK_DOWN`
  Link is down (e. g. no cable attached, not configured, remote station down).

  `XL_ETH_STATUS_LINK_UP`
  Link is up.

  `XL_ETH_STATUS_LINK_ERROR`
  Link is in error state (e. g. auto-negotiation failed).

- **speed**
  Network speed:

  `XL_ETH_STATUS_SPEED_UNKNOWN`
  Connection speed could not be determined
  (e. g. auto-negotiation not yet complete or link is down).

  `XL_ETH_STATUS_SPEED_100`
  Connection speed 100 Mbit/sec.

  `XL_ETH_STATUS_SPEED_1000`
  Connection speed 1000 Mbit/sec.

- **clockMode**
  Network speed:

  `XL_ETH_STATUS_CLOCK_DONT_CARE`
  Current connection does not have dedicated clocks.

  `XL_ETH_STATUS_CLOCK_MASTER`
  Device is clock master.

  `XL_ETH_STATUS_CLOCK_SLAVE`
  Device is clock slave.

- **bypass**
  Current bypass mode on this channel:

  `XL_ETH_BYPASS_INACTIVE`
  No bypass is active on this channel.

  `XL_ETH_BYPASS_PHY`
  Bypass is active in PHY mode.

  `XL_ETH_BYPASS_MACCORE`
  Bypass is active in MAC mode.

7.3.3.2 T_XL_ETH_CONFIG

Syntax

```
typedef struct {
  unsigned int speed;
  unsigned int duplex;
  unsigned int connector;
  unsigned int phy;
  unsigned int clockMode;
  unsigned int mdiMode;
  unsigned int brPairs;
} T_XL_ETH_CONFIG;
```

Parameters

- **speed**
  Specifies the desired channel bandwidth:
  - `XL_ETH_MODE_SPEED_AUTO_100`
    100BASE-TX only, enable auto-negotiation.
  - `XL_ETH_MODE_SPEED_AUTO_1000`
    1000BASE-T only, enable auto-negotiation.
  - `XL_ETH_MODE_SPEED_AUTO_100_1000`
    100BASE-TX or 1000BASE-T, enable auto-negotiation.
  - `XL_ETH_MODE_SPEED_FIXED_100`
    100BASE-TX or 100BASE-T1, no auto-negotiation.
  - `XL_ETH_MODE_SPEED_FIXED_1000`
    1000BASE-T1, no auto-negotiation.
- **duplex**
  Specifies the duplex mode for this channel:
  - `XL_ETH_MODE_DUPLEX_DONT_CARE`
    Used for BroadR-Reach, since only full duplex available for BR!
  - `XL_ETH_MODE_DUPLEX_AUTO`
    Requires auto-negotiation; only full duplex supported!
  - `XL_ETH_MODE_DUPLEX_FULL`
- **connector**
  Selects the connector to use for this channel:
  - `XL_ETH_MODE_CONNECTOR_DONT_CARE`
    Should be used on devices other than VN5610(A).
  - `XL_ETH_MODE_CONNECTOR_DSUB`
  - `XL_ETH_MODE_CONNECTOR_RJ45`

- **phy**
  Two different physical layers are supported on the VN5600 Interface Family: IEEE802.3 (“standard” Ethernet) and BroadR-Reach.
  - `XL_ETH_MODE_PHY_DONT_CARE`
  - `XL_ETH_MODE_PHY_IEEE_802_3`
    Only available on RJ-45 connector.
  - `XL_ETH_MODE_PHY_BROADR_REACH`
- **clockMode**
  Clock source to operation mode when using BroadR-Reach physical layer:
  - `XL_ETH_MODE_CLOCK_AUTO`
    Requires auto-negotiation, typically used for 1000 Base-T!
  - `XL_ETH_MODE_CLOCK_MASTER`
  - `XL_ETH_MODE_CLOCK_SLAVE`
  - `XL_ETH_MODE_CLOCK_DONT_CARE`
    Used for IEEE 802.3.
- **mdiMode**
  Medium-dependent interface mode (i.e. the assignment of transmit/receive wires on the connector):
  - `XL_ETH_MODE_MDI_AUTO`
    Auto-MDIX detection.
- **brPairs**
  Operation mode when using BroadR-Reach physical layer:
  - `XL_ETH_MODE_BR_PAIR_DONT_CARE`
    Used for IEEE 802.3.
  - `XL_ETH_MODE_BR_PAIR_1PAIR`
    Single-pair.

Valid configuration combinations

Due to hardware, protocol or driver restrictions, not all combinations of network speed, duplex mode, connector selection and clock mode are supported. See the fol-
lowing tables for the supported combinations.

Rj45

Configurations for the RJ-45 connector					
speed	**duplex**	**phy**	**clockMode**	**mdiMode**	**brPairs**
AUTO_100	AUTO	DON’T CARE	DON’T CARE	AUTO	DON’T CARE
AUTO_100_1000	AUTO	DON’T CARE	AUTO	AUTO	DON’T CARE
AUTO_1000	AUTO	DON’T CARE	AUTO	AUTO	DON’T CARE
FIXED_100	FULL	DON’T CARE	DON’T CARE	AUTO	DON’T CARE

Note
Connector should be set to `XL_ETH_MODE_CONNECTOR_DONT_CARE` on devices other than VN5610(A).

D-SUB9

Configurations for the RJ-45 connector					
speed	**duplex**	**phy**	**clockMode**	**mdiMode**	**brPairs**
FIXED_100	DON’T CARE	DON’T CARE	MASTER/SLAVE	AUTO	1PAIR



Note
On the D-SUB connector, only a single cable pair is provided per channel. Connector should be set to `XL_ETH_MODE_CONNECTOR_DONT_CARE` on devices other than VN5610(A).

ix Industrial

Configurations for the ix Industrial connector					
speed	**duplex**	**phy**	**clockMode**	**mdiMode**	**brPairs**
FIXED_100	DON'T CARE	DON'T CARE	MASTER/SLAVE	AUTO	1PAIR
FIXED_1000	DON'T CARE	DON'T CARE	MASTER/SLAVE	AUTO	1PAIR



Note
Connector should be set to `XL_ETH_MODE_CONNECTOR_DONT_CARE` on devices other than VN5610(A).

7.3.4 Events

7.3.4.1 T_XL_ETH_FRAME

Syntax

```
typedef union s_xl_eth_framedata {
  unsigned char  rawData[XL_ETH_RAW_FRAME_SIZE_MAX];
  T_XL_ETH_FRAME ethFrame;
} T_XL_ETH_FRAMEDATA;

typedef struct s_xl_eth_frame {
  unsigned short etherType;
  unsigned char  payload[XL_ETH_PAYLOAD_SIZE_MAX];
} T_XL_ETH_FRAME;
```

Description

Ethernet packet data definition used inside the Rx/Tx `tagData` structures.

Parameters

- **rawData**
  Raw values of the Ethernet packet.
- **etherType**
  Type of protocol encapsulated within the Ethernet packet.
- **payload**
  Packet payload.

7.3.4.2 T_XL_ETH_EVENT

Syntax

```
typedef unsigned short XLethEventTag;

typedef struct s_xl_eth_event {
  unsigned int   size;
  XLethEventTag  tag;
  unsigned short channelIndex;
  unsigned int   userHandle;
  unsigned short flagsChip;
  unsigned short reserved;
  XLuint64       reserved1;
  XLuint64       timestampSync;

  union s_xl_eth_tag_data {
    unsigned char rawData[XL_ETH_EVENT_SIZE_MAX];
    T_XL_ETH_DATAFRAME_RX               frameRxOk;
    T_XL_ETH_DATAFRAME_RX_ERROR         frameRxError;
    T_XL_ETH_DATAFRAME_TXACK            frameTxAck;
    T_XL_ETH_DATAFRAME_TXACK_OTHERAPP   frameTxAckOtherApp;
    T_XL_ETH_DATAFRAME_TXACK_SW         frameTxAckSw;
    T_XL_ETH_DATAFRAME_TX_ERROR         frameTxError;
    T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP frameTxErrorOtherApp;
    T_XL_ETH_DATAFRAME_TX_ERROR_SW      frameTxErrorSw;
    T_XL_ETH_CONFIG_RESULT              configResult;
    T_XL_ETH_LOSTEVENT                  lostEvent;
    T_XL_ETH_CHANNEL_STATUS             channelStatus;
    XL_SYNC_PULSE_EV syncPulse;
  } tagData;
} T_XL_ETH_EVENT;
```

Description

Structures describing the Ethernet events that can be received (including Tx events).

Parameters

- **size**
  Size of the complete Ethernet event, including header and payload data.

- **tag**
  Specifies the structure that is applied to `tagData`,
  e. g. XL_ETH_EVENT_TAG_FRAMERX.
- **channelIndex**
  Logical channel number where this event originated or is target to.
- **userHandle**
  Application-specific handle that may be used to link associated events, e. g. a transmit confirmation to the original send request. Not used (set to 0) for indications not related to a request.
- **flagsChip**

  The lower 8 bit contain chip information:
  Bit 0: `XL_ETH_CONNECTOR_RJ45`
  Bit 1: `XL_ETH_CONNECTOR_DSUB`
  Bit 2: `XL_ETH_PHY_IEEE`
  Bit 3: `XL_ETH_PHY_BROADR`
  Bit 4: `XL_ETH_FRAME_BYPASSED`
  Bit 5..7: unused

  The upper 8 bit contain special flags:
  Bit 8: XL_ETH_QUEUE_OVERFLOW

  Not all events generated by the device could be indicated to the application.

  Bit 9..14: unused
  Bit 15: XL_ETH_BYPASS_QUEUE_OVERFLOW

  Indicates that one or more received packets could not be sent to the opposite bus in MAC bypass mode.
- **reserved**
  Not being used, ignore.
- **reserved1**
  Not being used, ignore.
- **timestampSync**
  Synchronized time stamp with 1 ns resolution (PC → device) and an accuracy of 8 µs. Time synchronization is applied if enabled in Vector Hardware Control Panel. Offset correction is possible with xlResetClock.
- **tagData**
  See structures on page 176 ... page 182 for further details.

7.3.4.3 T_XL_ETH_DATAFRAME_RX

Syntax

```
typedef struct s_xl_eth_dataframe_rx {
  unsigned int       frameIdentifier;
  unsigned int       frameDuration;
  unsigned short     dataLen;
  unsigned short     reserved;
  unsigned int       reserved2[3];
  unsigned int       fcs;
  unsigned char      destMAC[XL_ETH_MACADDR_OCTETS];
  unsigned char      sourceMAC[XL_ETH_MACADDR_OCTETS];
  T_XL_ETH_FRAMEDATA frameData;
} T_XL_ETH_DATAFRAME_RX;
```

Description — This event carries a received Ethernet packet.

Tag — `XL_ETH_EVENT_TAG_FRAMERX`

Parameters

- **frameIdentifier**
  Unique identifier assigned during reception. Used to correlate a later Tx event (in case of MAC bypass) to the Rx event, so that the application may monitor incoming and outgoing frames.
- **frameDuration**
  Transmit duration of the frame, given in nanoseconds.
- **dataLen**
  Combined size of etherType and payload in bytes. This specifies the size actually used, not the maximum size of the struct.
- **reserved**
  Not being used, ignore.
- **reserved2**
  Not being used, ignore.
- **fcs**
  Frame Check Sequence as received from network.
- **destMAC**
  Destination MAC address.
- **sourceMAC**
  Source MAC address.
- **frameData**
  section T_XL_ETH_FRAME on page 175

7.3.4.4 T_XL_ETH_DATAFRAME_RX_ERROR

Syntax

```
typedef struct s_xl_eth_dataframe_rxerror {
  unsigned int frameIdentifier;
  unsigned int frameDuration;
  unsigned int errorFlags;
  unsigned short dataLen;
  unsigned short reserved;
  unsigned int reserved2[3];
  unsigned int fcs;
  unsigned char destMAC[6];
  unsigned char sourceMAC[6];
  T_XL_ETH_FRAMEDATA frameData;
} T_XL_ETH_DATAFRAME_RX_ERROR;
```

Description — This event carries a received Ethernet packet that was received with an error.

Tag — `XL_ETH_EVENT_TAG_FRAMERX_ERROR`

Parameters

- **frameIdentifier**
  Unique identifier assigned during receive. Used to correlate a later Tx event (in case of MAC bypass) to the Rx event.
- **frameDuration**
  Transmit duration of the Ethernet packet, given in nanoseconds.

- **errorFlags**
  Cause of receive error.
  `XL_ETH_RX_ERROR_INVALID_LENGTH`
  `XL_ETH_RX_ERROR_INVALID_CRC`
  `XL_ETH_RX_ERROR_PHY_ERROR`
- **dataLen**
  Combined size of `etherType` and `payload` in bytes. This specifies the size actually used, not the maximum size of the struct.
- **reserved**
  Not being used, ignore.
- **reserved2**
  Not being used, ignore.
- **fcs**
  Frame Check Sequence, as received from network.
- **destMAC**
  Destination MAC address.
- **sourceMAC**
  Source MAC address.
- **frameData**
  See section T_XL_ETH_FRAME on page 175.

7.3.4.5 T_XL_ETH_DATAFRAME_TX_EVENT

Syntax

```
typedef struct s_xl_eth_dataframe_tx_event {
  unsigned int       frameIdentifier;
  unsigned int       flags;
  unsigned short     dataLen;
  unsigned short     reserved;
  unsigned int       frameDuration;
  unsigned int       reserved2[2];
  unsigned int       fcs;
  unsigned char      destMAC[XL_ETH_MACADDR_OCTETS];
  unsigned char      sourceMAC[XL_ETH_MACADDR_OCTETS];
  T_XL_ETH_FRAMEDATA frameData;
} T_XL_ETH_DATAFRAME_TX_EVENT;
```

Description

The structure describes an Ethernet event that can be received after a Tx Ethernet packet has been sent by the application.

Note
The parameters `destMAC`, `sourceMAC`, `etherType` and payload are in ‘network byte order’.

Parameters

- **frameIdentifier**
  Unique identifier assigned by the device. For packets sent by the Bypass feature this matches the respective element of the original Rx event.
- **flags**
  Transmit flags requested by the application and processed by the device. See xlEthTransmit() for a description of allowed flags.
- **dataLen**
  Combined size of `etherType` and `payload` in bytes. This specifies the size actually used, not the maximum size of the struct.

- **reserved**
  Not being used, ignore.
- **frameDuration**
  Transmit duration of this Ethernet packet in nanoseconds.
- **reserved2**
  Not being used, ignore.
- **fcs**
  Frame Check Sequence generated for this Ethernet packet.
- **destMAC**
  Destination MAC address.
- **sourceMAC**
  Source MAC address.
- **frameData**
  See section T_XL_ETH_FRAME on page 175.

7.3.4.6 T_XL_ETH_DATAFRAME_TXACK

Syntax

```
typedef T_XL_ETH_DATAFRAME_TX_EVENT T_XL_ETH_DATAFRAME_TXACK;
```

Description

This event is indicated to the application each time an Ethernet packet has been successfully sent to the bus. It is neither a delivery confirmation from the receiver, nor a guarantee that the intended recipient will receive that frame. It currently has an identical layout to the Tx request packet; the different name is merely for a better understanding.

Tag

`XL_ETH_EVENT_TAG_FRAMETX_ACK`

Parameters

For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT.

7.3.4.7 T_XL_ETH_DATAFRAME_TXACK_OTHERAPP

Syntax

```
typedef T_XL_ETH_DATAFRAME_TX_EVENT
T_XL_ETH_DATAFRAME_TXACK_OTHERAPP;
```

Description

This event indicates the successful sending of an Ethernet packet by another application.

Tag

`XL_ETH_EVENT_TAG_FRAMETX_ACK_OTHER_APP`

Parameters

For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT.

7.3.4.8 T_XL_ETH_DATAFRAME_TXACK_SW

Syntax

```
typedef T_XL_ETH_DATAFRAME_TX_EVENT
T_XL_ETH_DATAFRAME_TXACK_SW;
```

Description

This event is indicated to the application each time a received Ethernet packet has been successfully forwarded to the connected bus when in MAC bypass mode. It is

neither a delivery confirmation from the receiver, nor a guarantee that the intended recipient will receive that frame. It currently has an identical layout to the Tx request packet; the different name is merely for a better understanding.

Tag `XL_ETH_EVENT_TAG_FRAMETX_ACK_SWITCH`

Parameters For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT.

7.3.4.9 T_XL_ETH_DATAFRAME_TX_ERROR

Syntax

```
typedef struct s_xl_eth_dataframe_txerror {
  unsigned int                 errorType;
  T_XL_ETH_DATAFRAME_TX_EVENT txFrame;
} T_XL_ETH_DATAFRAME_TX_ERROR;
```

Description This event is indicated to the application each time an Ethernet packet has not been sent to the bus.

Tag `XL_ETH_EVENT_TAG_FRAMETX_ERROR`

Parameters

- **errorType**
  Indicates the kind of transmission error and can be one of the following values:

  `XL_ETH_TX_ERROR_BYPASS_ENABLED`
  Bypass enabled.

  `XL_ETH_TX_ERROR_NO_LINK`
  No link established.

  `XL_ETH_TX_ERROR_PHY_NOT_CONFIGURED`
  PHY not yet configured.

- **txFrame**
  section T_XL_ETH_DATAFRAME_TX_EVENT on page 178

7.3.4.10 T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP

Syntax

```
typedef T_XL_ETH_DATAFRAME_TX_ERROR
T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP;
```

Description This event is indicated to the application each time an erroneous Ethernet packet has been sent to the bus by another application.

Tag `XL_ETH_EVENT_TAG_FRAMETX_ERROR_OTHER_APP`

Parameters For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_ERROR.

7.3.4.11 T_XL_ETH_DATAFRAME_TX_ERR_SW

Syntax

```
typedef T_XL_ETH_DATAFRAME_TX_ERROR
T_XL_ETH_DATAFRAME_TX_ERR_SW;
```

Description This event is indicated to the application each time a received Ethernet packet

could not be sent to the connected bus when in MAC bypass mode. This may occur if there is no active link on the connected bus, or in case of internal errors. Currently, the event has an identical layout to the Tx error event packet; the different name is merely for a better understanding.

Tag `XL_ETH_EVENT_TAG_FRAMETX_ERROR_SWITCH`

Parameters For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_ERROR.

7.3.4.12 T_XL_ETH_CONFIG_RESULT

Syntax

```
struct s_xl_eth_config_result {
  unsigned int result;
} T_XL_ETH_CONFIG_RESULT;
```

Description This event is generated when a configuration change via xlEthSetConfig() was triggered.

Tag `XL_ETH_EVENT_TAG_CONFIGRESULT`

Parameters

- **result**
  0: Valid parameter combination set via xlEthSetConfig().
  != 0: Invalid parameter combination set via xlEthSetConfig().

7.3.4.13 T_XL_ETH_LOSTEVENT

Syntax

```
typedef struct s_xl_eth_lostevent {
  XLethEventTag eventTypeLost;
  unsigned short reserved;
  unsigned int reason;

  union {
    struct {
      unsigned int frameIdentifier;
      unsigned int fcs;
      unsigned char sourceMAC[XL_ETH_MACADDR_OCTETS];
      unsigned char reserved[2];
    } txAck, txAckSw;

    struct {
      unsigned int errorType;
      unsigned int frameIdentifier;
      unsigned int fcs;
      unsigned char sourceMAC[XL_ETH_MACADDR_OCTETS];
      unsigned char reserved[2];
    } txError, txErrorSw;

    unsigned int reserved[20];
  } eventInfo;
} T_XL_ETH_LOSTEVENT;
```

Description This event is generated when the driver detects a regular event that could not be indicated to the application (e. g. not all data available).

Tag `XL_ETH_EVENT_TAG_LOSTEVENT`

Parameters See respective regular events.

7.3.4.14 T_XL_ETH_CHANNEL_STATUS

Syntax

```
struct s_xl_eth_channel_status {
  unsigned int link;
  unsigned int speed;
  unsigned int duplex;
  unsigned int mdiMode;
  unsigned int activeConnector;
  unsigned int activePhy;
  unsigned int clockMode;
  unsigned int brPairs;
} T_XL_ETH_CHANNEL_STATUS;
```

Description

This event is generated each time the link information changes.

Tag

`XL_ETH_EVENT_TAG_CHANNEL_STATUS`

Parameters

- **link**
  Link state:
  `XL_ETH_STATUS_LINK_UNKNOWN`
  `XL_ETH_STATUS_LINK_DOWN`
  `XL_ETH_STATUS_LINK_UP`
  `XL_ETH_STATUS_LINK_ERROR`

- **speed**
  Current Ethernet connection speed:

  `XL_ETH_STATUS_SPEED_UNKNOWN`

  `XL_ETH_STATUS_SPEED_100`
  100 Mbit/s operation.

  `XL_ETH_STATUS_SPEED_1000`
  1000 Mbit/s operation.

  `XL_ETH_STATUS_SPEED_2500`
  2500 Mbit/s operation.

  `XL_ETH_STATUS_SPEED_5000`
  5000 Mbit/s operation.

  `XL_ETH_STATUS_SPEED_10000`
  10000 Mbit/s operation.

- **Duplex**
  The duplex setting:
  `XL_ETH_STATUS_DUPLEX_UNKNOWN`
  `XL_ETH_STATUS_DUPLEX_FULL`

- **mdiMode**
  The active MDI state:

  `XL_ETH_STATUS_MDI_UNKNOWN`

  `XL_ETH_STATUS_MDI_STRAIGHT`
  MDI.

  `XL_ETH_STATUS_MDI_CROSSOVER`
  MDI-X.

- **activeConnector**
  The interface connector currently assigned to the MAC:
  `XL_ETH_STATUS_CONNECTOR_DEFAULT`
  `XL_ETH_STATUS_CONNECTOR_RJ45`
  `XL_ETH_STATUS_CONNECTOR_DSUB`

- **activePhy**
  The currently active transmitter (physical interface):
  `XL_ETH_STATUS_PHY_UNKNOWN`
  `XL_ETH_STATUS_PHY_802_3`
  `XL_ETH_STATUS_PHY_BROADR_REACH`
  `XL_ETH_STATUS_PHY_100BASE_T1`
  `XL_ETH_STATUS_PHY_1000BASE_T1`

- **clockMode**
  Clock mode setting of the connection:

  `XL_ETH_STATUS_CLOCK_DONT_CARE`
  Reported for IEEE 802.3.

  `XL_ETH_STATUS_CLOCK_MASTER`
  `XL_ETH_STATUS_CLOCK_SLAVE`

- **brPairs**
  The number of cable pairs used for the link:

  `XL_ETH_STATUS_BR_PAIR_DONT_CARE`
  Reported for IEEE 802.3.

  `XL_ETH_STATUS_BR_PAIR_1PAIR`

7.3.4.15 T_XL_ETH_DATAFRAME_TX

Syntax

```
typedef struct s_xl_eth_dataframe_tx {
  unsigned int       frameIdentifier;
  unsigned int       flags;
  unsigned short     dataLen;
  unsigned short     reserved;
  unsigned int       reserved2[4];
  unsigned char      destMAC[6];
  unsigned char      sourceMAC[6];
  T_XL_ETH_FRAMEDATA frameData;
} T_XL_ETH_DATAFRAME_TX;
```

Description

The following structure describes an Ethernet packet that can be sent to one or more network links using the xlEthTransmit() or xlNetEthSend() function.

Parameters

- **frameIdentifier**
  Unique identifier assigned by hardware. Set to 0.
- **flags**
  Bit field indicating whether to use the source MAC address given by application or whether the hardware should insert / calculate the respective values:

  `XL_ETH_DATAFRAME_FLAGS_USE_SOURCE_MAC`
  Use the given source MAC address (not inserted by hardware).
- **dataLen**
  Combined size of `etherType` and `payload` in bytes. This specifies the size actually used, not the maximum size of the struct.
- **reserved**
  Not used. Must be set to 0.
- **reserved2**
  Not used. Must be set to 0.
- **destMAC**
  Destination MAC address.
- **sourceMAC**
  Source MAC address.
- **frameData**
  section T_XL_ETH_FRAME on page 175

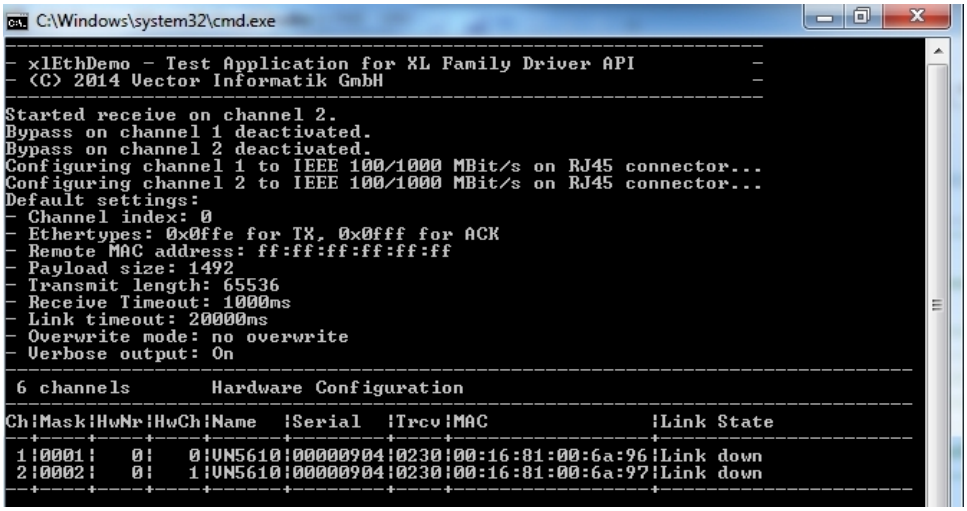
7.3.5 Application Examples

7.3.5.1 xlEthDemo

General Information

Description

This example demonstrates how to transmit/receive Ethernet packets. It contains a small command line interface which can be controlled by a few keyboard commands. After starting, the example searches for Ethernet channels on the connected devices, then it sets up a default Ethernet configuration and activates those channels. If the example finds more than one channel it is possible to send and receive Ethernet packets in a loop e. g. by pressing <t>. It is also possible to send frames in a burst mode by pressing <b>. To transmit a complete file via Ethernet use the command line options to start the example (/t).



Keyboard Commands

The running application can be controlled by the following keyboard commands:

Key	Command
<1>…(max eth channels)	Select an Ethernet channel.
<+>	Select next Ethernet channel.
<->	Select previous Ethernet channel.
<a>	Activate current channel.
<d>	Deactivate current channel.
<c>	Set channel configuration.
<t>	Transmit single packet.
<b>	Start burst transmission (needs active receiver).
<s>	Stop burst transmission.
<r>	Receive.
<e>	Set Ether type to use.
<p>	Set packet payload size.
<l>	Set burst data length.

Key	Command
<m>	Set receiver MAC address.
<k>	Twinkle status LED of device.
<w>	Show driver configuration.
<v>	Toggle verbose output.

Command Line Interface

The following command line options are available:

Key	Description
/h or /?	This help.
/dn	XLAPI Ethernet device channel n to use (n = 1,2,...).
/cX	Use channel configuration mode X.
/t	Transmit test pattern.
/t\"Name\"	Transmit content of file.
/r	Receive data.
/r\"Name\"	Receive data and write to file; the file must not exist.
/eX,Y	Use Ether type X for transmission, Y for acknowledge.
/pX	Maximum transmit packet payload in bytes (42...1500).
/lX	Maximum transmit length (0=no limit/file size).
/mX	Receiver MAC address X (format: aa:bb:cc:dd:ee:ff).
/oX	Transmit/receive timeout in milliseconds (0=Disable timeout).
/v	Verbose output.
/w	Twinkle status LED of device owning the given XLAPI channel and exit.
/q	Quit after transmit/receive.

7.3.5.2 xlEthBypassDemo

Description

The bypass demo is a small command-line tool that shows how to configure a Vector Ethernet device, activate a channel bypass and how to receive data indications.
The device can be started without arguments; in this case, a default operation is being used. For a list of the possible command-line arguments, run the tool with a “/?” argument.

8 LIN Commands

In this chapter you find the following information:

8.1 Introduction **189**
8.2 Flowchart **190**
8.3 LIN Basics **191**
8.4 Functions **192**
8.5 Structs **198**
8.6 Events **199**
8.7 Application Examples **202**

8.1 Introduction

Description

The **XL Driver Library** enables the development of LIN applications for supported Vector devices (see section System Requirements on page 25). A LIN application always requires **init access**(see section xlOpenPort on page 38)multiple LIN applications cannot use a common physical LIN channel at the same time.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel parameters can be changed/configured
- LIN messages can be transmitted on the channel
- LIN messages can be received on the channel

Without init access

- Not supported. If the application gets no **init access** on a specific channel, no further function call is possible on the according channel.



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

8.2 Flowchart

Calling sequence

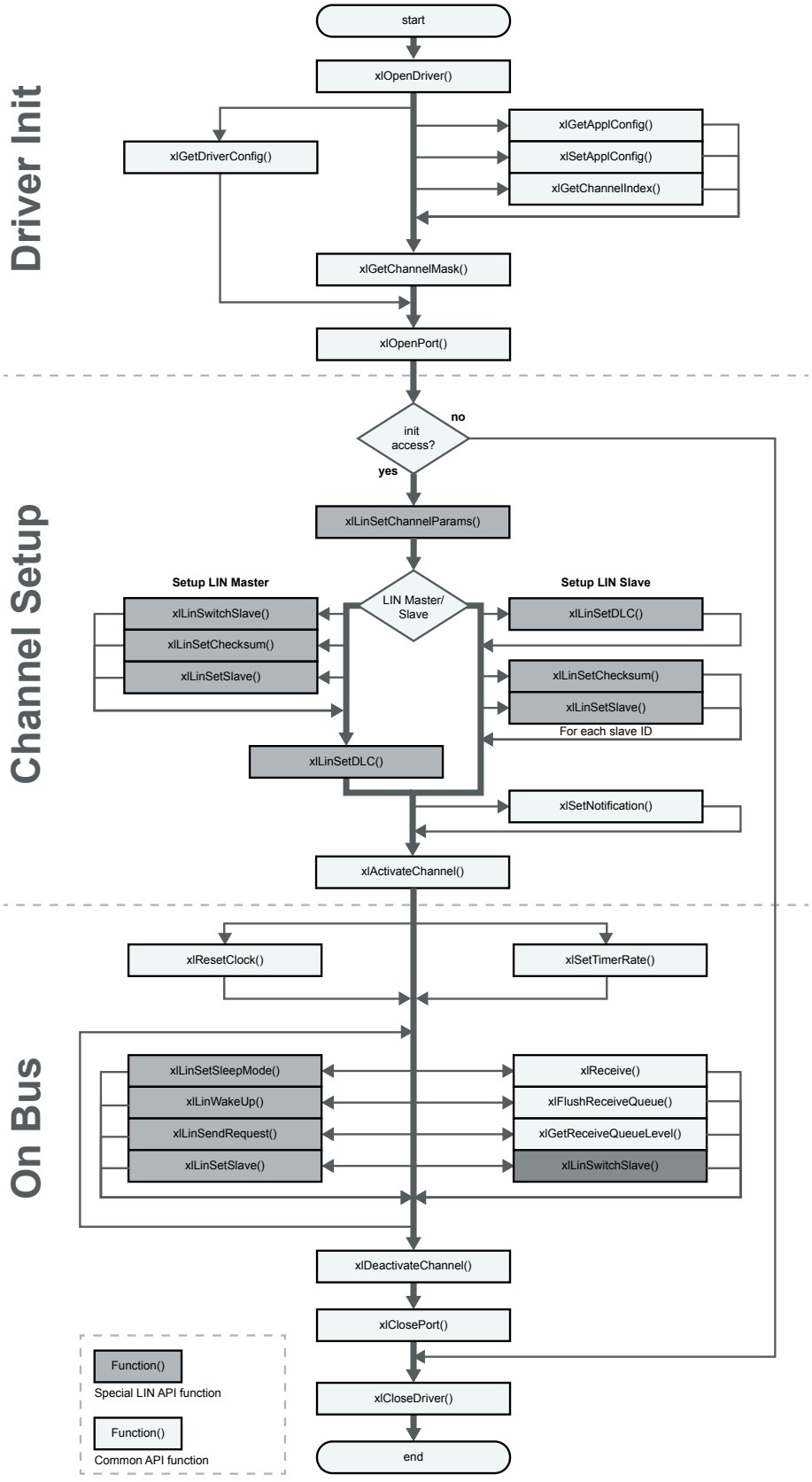


Figure 38: Function calls for LIN applications

8.3 LIN Basics

Advantages of LIN

LIN (Local Interconnect Network) is a cheap way to connect many sensors and actuators to an ECU via one common communication medium (bus). This diminishes complexity as well as costs, weight and space problems and in addition it offers the possibility of diagnostics. Furthermore, LIN offers a high flexibility to extend a system.

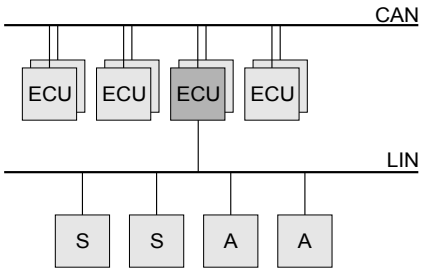
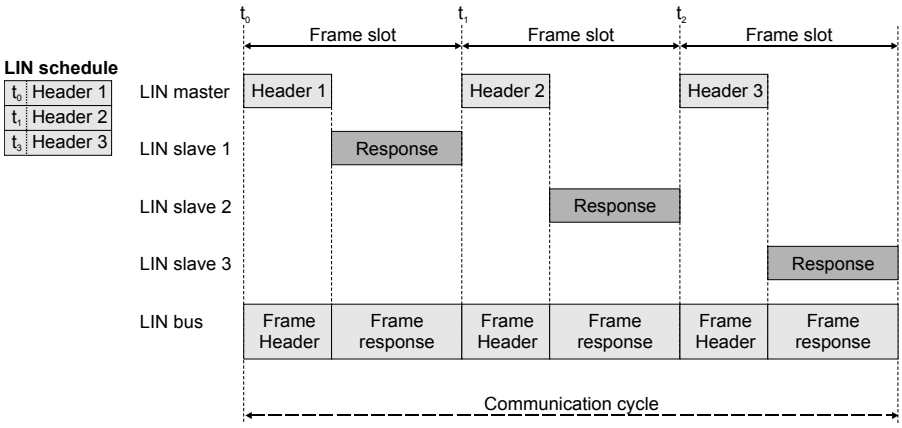


Figure 39: CAN and LIN

Functional principle

The LIN network is based on a master-slave architecture where the LIN master is one privileged node of the LIN network. The master consists of a master task as well as a slave task, while the slaves only comprise a slave task.

The LIN master task controls slave tasks by sending special patterns called headers on the bus at times defined within a so called schedule table. Such a header contains a message address and can be viewed as a request to be responded to by one LIN slave task. The total of header plus slave task response is called a LIN message. All other slaves can either receive the LIN message or ignore it.



LIN messages

Generally, there are 62 identifiers i.e. LIN messages possible within a LIN2.x network, two of which (60 and 61) are dedicated to diagnostics on LIN (see xlLinSetDLC()). A response can contain up to eight data bytes (defined for each slave, see xlLinSetSlave()).

XL API

The XL API comprises functions for the LIN master as well as the LIN slaves, allowing sending and receiving messages on the LIN bus with any Vector XL Interface. If using the XL API for the master, be sure to have it defined via xlLinSetChannelParams() with Master flag. Furthermore, the XL API can be simultaneously used for LIN slaves, which must be configured separately via `xlLinSetChannelParams()` (Slave flag), `xlLinSetDLC`, xlLinSetChecksum() and `xlLinSetSlave`. See the LIN flowchart and the provided LIN examples for further details.

8.4 Functions

8.4.1 xlLinSetChannelParams

Syntax

```
XLstatus xlLinSetChannelParams (
  XLportHandle portHandle,
  XLaccess     accessMask,
  XLlinStatPar statPar)
```

Description

Sets the channel parameters like baud rate, master, slave.

Note
The function opens all acceptance filters for LIN. In other words, the application receives `XL_LIN_MSG` events for all LIN IDs. Resets all DLC's (xlLinSetDLC())!

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **statPar**
  Defines the mode of the LIN channel and the baud rate (see section XLlinStatPar on page 198).

Return value

Returns an error code (see section Error Codes on page 360).

8.4.2 xlLinSetDLC

Syntax

```
XLstatus xlLinSetDLC(
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned char DLC[64]
)
```

Description

Defines the data length for all requested messages. This is needed for the LIN master (and recommended for LIN slave) and must be called before activating a channel.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **DLC**
  Specifies the length of all LIN messages (0…63).
  The value can be `0…8` for a valid DLC.

Return value

Returns an error code (see section Error Codes on page 360).



Example

Setting DLC for LIN message with ID 0x04 to 8 and for all other IDs to undefined.

```
unsigned char DLC[64];
for (int i=0;i<64;i++) DLC[i] = XL_LIN_UNDEFINED_DLC;
DLC[4] = 8;
xlStatus = xlLinSetDLC(m_XLportHandle,
                       m_xlChannelMask[MASTER],
                       DLC);
```

8.4.3 xlLinSetChecksum

Syntax

```
XLstatus xlLinSetChecksum (
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned char checksum[60])
```

Description

This function is only for a LIN 2.0 node and must be called before activating a channel. The checksum calculation can be changed here from the classic to enhanced model for the LIN IDs 0..59. The LIN ID 60..63 range is fixed to the classic model and cannot be changed. The classic model is always set for all IDs by default. There are no changes when it is called for a LIN 1.3 node.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **checksum**
  `XL_LIN_CHECKSUM_CLASSIC`
  Sets to classic calculation (use only data bytes).

  `XL_LIN_CHECKSUM_ENHANCED`
  Sets to enhanced calculation (use data bytes including the id field).

  `XL_LIN_CHECKSUM_UNDEFINED`
  Sets to undefined calculation.

Return value

Returns an error code (see section Error Codes on page 360).

Example

Setting the checksum for a LIN message with the ID 0x04 to “enhanced” and for all other IDs to “undefined”

```
unsigned char checksum[60];
for (int i = 0; i < 60; i++)
  checksum[i] = XL_LIN_CHECKSUM_UNDEFINED;

checksum[4] = XL_LIN_CHECKSUM_ENHANCED;
xlStatus = xlLinSetChecksum(m_XLportHandle,
                            m_xlChannelMask[MASTER],
                            checksum);
```

8.4.4 xlLinSetSlave

Syntax

```
XLstatus xlLinSetSlave (
  XLportHandle   portHandle,
  XLaccess       accessMask,
  unsigned char linId,
  unsigned char data[8],
  unsigned char dlc,
  unsigned short checksum)
```

Description

Sets up a LIN slave. This function must be called **before** activating a channel and for **each** slave ID separately. After activating the channel it is only possible to change the data, dlc and checksum but not the `linID`.

This function is also used to setup a slave task within a master node. If the function is not called but activated the channel is only listening.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **linID**
  LIN ID on which the slave transmits a response.
- **data**
  Contains the data bytes.
- **dlc**
  Defines the dlc for the LIN message.
- **checksum**
  Defines the checksum (it is also possible to set a faulty checksum). If the API should calculate the checksum use the following defines:

  `XL_LIN_CALC_CHECKSUM`
  Use the classic checksum calculation (only databytes)

  `XL_LIN_CALC_CHECKSUM_ENHANCED`
  Use the enhanced checksum calculation (databytes and id field)

Return value

Returns an error code (see section Error Codes on page 360).



Example

Setting up a LIN slave for ID=0x04

```
unsigned char data[8];
unsigned char id  = 0x04;
unsigned char dlc = 8;

data[0] = databyte;
data[1] = 0x00;
data[2] = 0x00;
data[3] = 0x00;
data[4] = 0x00;
data[5] = 0x00;
data[6] = 0x00;
data[7] = 0x00;
xlStatus = xlLinSetSlave(m_XLportHandle,
                         m_xlChannelMask[SLAVE],
                         id,
                         data,
                         dlc,
                         XL_LIN_CALC_CHECKSUM);
```

8.4.5 xlLinSwitchSlave

Syntax

```
XLstatus xlLinSwitchSlave (
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned char linId,
  unsigned int  mode)
```

Description

The function can switch on/off a LIN slave during measurement.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **linID**
  Contains the master request LIN ID.
- **mode**
  `XL_LIN_SLAVE_ON`
  Switch on the LIN slave.

  `XL_LIN_SLAVE_OFF`
  Switch off the LIN slave.

Return value

Returns an error code (see section Error Codes on page 360).

8.4.6 xlLinSendRequest

Syntax

```
XLstatus xlLinSendRequest (
  XLportHandle  portHandle,
  XLaccess      accessMask,
```

```
  unsigned char linId,
  unsigned int  flags)
```

Description

Sends a master LIN request to the slave(s).
After successfully transmission, the port (which sends the message) gets a `XL_LIN_MSG` event with a set `XL_LIN_MSGFLAG_TX` flag.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **linID**
  Contains the master request LIN ID.
- **flags**
  For future use. Set to 0.

Return value

Returns `XL_ERR_INVALID_ACCESS` if it is done on a LIN slave (see section Error Codes on page 360)

8.4.7 xlLinWakeUp

Syntax

```
XLstatus xlLinWakeUp (
  XLportHandle portHandle,
  XLaccess accessMask)
```

Description

Transmits a wake-up request. The call generates a wake-up event.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

Return value

Returns an error code (see section Error Codes on page 360).

8.4.8 xlLinSetSleepMode

Syntax

```
XLstatus xlLinSetSleepMode (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int flags,
  unsigned char linId)
```

Description

Sets a LIN channel into sleep mode. With the parameter `flag` its possible to setup a `linID` which will be send at wake-up. The call generates a sleep mode event. If the LIN bus is inactive the node automatically enter the sleep mode.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **flags**
  `XL_LIN_FLAG_NO_SLEEP_MODE_EVENT`
  No sleep mode event is generated.

  `XL_LIN_FLAG_USE_ID_AS_WAKEUPID`
  Use the given ID as wake-up ID.

  With `flags`, it is possible to control the sleep mode event generation and to define `linID` which will be send at wake-up. Per default a sleep mode event is generated. The flags can be used in disjunction (`XL_LIN_FLAG_NO_SLEEP_MODE_EVENT | XL_LIN_FLAG_USE_ID_AS_WAKEUPID`) to inhibit the sleep mode event generation and use the given ID as wake-up ID.
- **linID**
  Defines the LIN ID that is transmitted at wake-up.

Return value

Returns an error code (see section Error Codes on page 360).

8.5 Structs

8.5.1 XLlinStatPar

Syntax

```
typedef struct {
  unsigned int LINMode;
  int          baudrate;
  unsigned int LINVersion;
  unsigned int reserved;
} XLlinStatPar;
```

Parameters

- **LINMode**
  Sets the channel mode.

  `XL_LIN_MASTER`
  Set channel to a LIN master.

  `XL_LIN_SLAVE`
  Set channel to LIN slave.

- **baudrate**
  Sets the baud rate, e. g. 9600, 19200, ...

  The baud rate range is 200 … 30.000 Bd. Please note that the functionality of the XL API is guaranteed for 200 … 20.000 Bd according to the LIN specification. Higher values should be used with care. With LINpiggy 7269mag and the onboard LIN of VN1611 and VN1531, up to 300.000 Bd ("Flash Mode") can be configured, depending on the bus physic.

- **LINVersion**

  `XL_LIN_VERSION_1_3`
  Use LIN 1.3 protocol

  `XL_LIN_VERSION_2_0`
  Use LIN 2.0 protocol

  `XL_LIN_VERSION_2_1`
  Use LIN 2.1 protocol

- **reserved**
  Reserved for future use. Set to 0.



Example

Setting up channel as a SLAVE to 9k6 and LIN 1.3

```
XLlinStatPar xlStatPar;
xlStatPar.LINMode = XL_LIN_SLAVE;
xlStatPar.baud rate = 9600;

// use LIN 1.3
xlStatPar.LINVersion = XL_LIN_VERSION_1_3;
xlStatus = xlLinSetChannelParams(m_XLportHandle,
                                 m_xlChannelMask[SLAVE],
                                 xlStatPar);
```

8.6 Events

8.6.1 XL LIN Message API

Syntax

```
union s_xl_lin_msg_api {
  struct s_xl_lin_msg      linMsg;
  struct s_xl_lin_no_ans   linNoAns;
  struct s_xl_lin_wake_up  linWakeUp;
  struct s_xl_lin_sleep    linSleep;
  struct s_xl_lin_crc_info linCRCinfo;
};
```

Parameters

- **linMsg**
  Structure for the LIN messages (see section XL LIN Message on page 199).
- **linNoAns**
  Structure for the LIN message that gets no answer (see section LIN No Answer on page 200).
- **linWakeUp**
  Structure for the wake up events (see section LIN Wake Up on page 200).
- **linSleep**
  Structure for the sleep events (see section LIN Sleep on page 201).
- **linCRCino**
  Structure for the CRC info events (see section LIN CRC Info on page 201).

8.6.2 XL LIN Message

Syntax

```
struct s_xl_lin_msg {
  unsigned char  id;
  unsigned char  dlc;
  unsigned short flags;
  unsigned char  data[8];
  unsigned char  crc;
};
```

Tag

`XL_LIN_MSG` (see section XLevent on page 70).

Parameters

- **id**
  Received LIN message ID.
- **dlc**
  The DLC of the received LIN message.
- **flags**
  `XL_LIN_MSGFLAG_TX`
  The LIN message was sent by the same LIN channel.

  `XL_LIN_MSGFLAG_CRCERROR`
  LIN CRC error.
- **data**
  Content of the message.
- **crc**
  Checksum.

8.6.3 XL LIN Error Message

Tag `XL_LIN_ERRMSG` (see section XLevent on page 70).

8.6.4 XL LIN Sync Error

Description Notifies an error in analyzing the sync field.

Tag `XL_LIN_SYNC_ERR` (see section XLevent on page 70).

8.6.5 LIN No Answer

Syntax

```
struct s_lin_NoAns {
  unsigned char id;
}
```

Description If a LIN master request gets no slave response a `linNoAns` event is received.

Tag `XL_LIN_NOANS` (see section XLevent on page 70).

Parameters

- **id**
  The LIN ID on which was the master request.

8.6.6 LIN Wake Up

Syntax

```
struct s_xl_lin_wake_up {
  unsigned char flag;
  unsigned char unused[3];
  unsigned int startOffs;
  unsigned int width;
};
```

Description This event indicates that the channel received a wake up signal.

Tag `XL_LIN_WAKEUP` (see section XLevent on page 70).

Parameters

- **flag**
  If the wake-up signal comes from the internal hardware, the flag is set to `XL_LIN_WAKUP_INTERNAL` otherwise it is not set (external wake-up).
- **unused**
  Reserved for future use.
- **startOffs**
  Timestamp correction offset.
- **width**
  Timestamp correction width.

Note
The real time stamp can be calculated as follows:

```
time stamp = (pxlEvent→timeStamp - wakeUp.StartOffs)
+ wakeUp.Width
```

8.6.7 LIN Sleep

Syntax

```
struct s_lin_Sleep {
  unsigned char flag;
}
```

Description

This event indicates changes in the sleep state of the channel.

Tag

`XL_LIN_SLEEP` (see section XLevent on page 70).

Parameters

- **flag**
  - `XL_LIN_SET_SLEEPMODE`
    Channel entered sleep mode.
  - `XL_LIN_COMESFROM_SLEEPMODE`
    Channel exited sleep mode.
  - `XL_LIN_STAYALIVE`
    Channel should enter sleep mode but cannot (for example because of bus activity).

8.6.8 LIN CRC Info

Syntax

```
struct s_xl_lin_crc_info {
  unsigned char id;
  unsigned char flags;
};
```

Description

This event is only used if the LIN protocol is >= 2.0.

If a LIN >= 2.0 node is initialized and the function xlLinSetChecksum() is not called (and no checksum model is defined) the hardware detects the according checksum model by itself. The event occurs only one time for the according LIN ID.

Tag

`XL_LIN_CRCINFO` (see section XLevent on page 70).

Parameters

- **id**
  Contains the id for the according checksum model.
- **flag**
  `XL_LIN_CHECKSUM_CLASSIC`
  Classic checksum model detected.

  `XL_LIN_CHECKSUM_ENHANCED`
  Enhanced checksum model detected.

8.7 Application Examples

8.7.1 xlLINExample

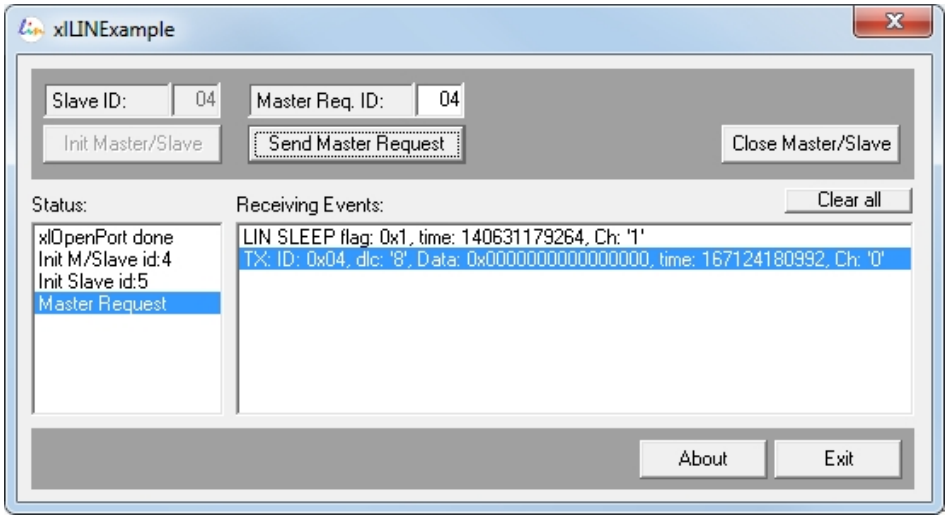
8.7.1.1 General Information

Description

This example demonstrates the basic use of the LIN API. It sets a LIN master including a LIN slave at one channel and if available a LIN slave to the second channel.

The channel assignment can be done with the Vector Hardware Manager. If the application starts the first time, it sets CH01 to a LIN master including a slave, and if possible CH02 to a LIN slave.

After the successfully LIN initialization the LIN master can transmit some requests.



8.7.1.2 Classes

Description

The example has the following class structure:

- **CaboutDlg**
  About box. → `AboutDlg.cpp`
- **CLINExampleApp**
  Main MFC class → `xlLINExample.cpp`
- **CLINExampleDlg**
  The ‘main’ dialog box → `xlLINExampleDlg.cpp`
- **CLINFunctions**
  Contains all functions for the LIN access → `xlLINFunctions.cpp`

8.7.1.3 Functions

Description

- **LINGetDevice**
  In order to get the channel mask, use xlGetChannelMask() to read all hardware parameters. xlGetApplConfig() checks whether the application has already been assigned. If not, a new entry with xlSetApplConfig() is created.
- **LINInit**
  `LINInit` opens one port for one channel, or if available two channels (CH1 and CH2). The first channel will be initialized as LIN master including a LIN slave (id=4), the other channel as LIN slave (id=5). After a successfully xlOpenPort() call, a Rx thread is created. Use xlLinSetChannelParams() in order to initialize the channels (like master/slave and the baud rate). It is also recommended to set up the LIN dlc with xlLinSetDLC().
- **linInitMaster**
  In order to use the LIN bus, it is necessary to define the specific DLC for each LIN ID. → xlLinSetDLC(). This must be done only for a LIN master and before you go ‘onBus’.
- **linInitSlave**
  Use xlLinSetSlave() to set up slave. Before you go ‘onBus’ it is needed to define the LIN slave ID that cannot be changed after xlActivateChannel(). All other parameters like the data values or the DLC can be varied.
- **LINSendMasterReq**
  After the LIN network is specified and the master/slaves are ‘onBus’, the master can transmit master requests with xlLinSendRequest().
- **LINClose**
  When all is done, the port is closed with xlClosePort().

9 K-Line Commands

In this chapter you find the following information:

9.1 Introduction **205**
9.2 Flowchart **206**
9.3 Functions **207**
9.4 Structs **212**
9.5 Events **216**

9.1 Introduction

Description

The **XL Driver Library** enables the development of K-Line applications for supported Vector devices (see section System Requirements on page 25). A K-Line application always requires **init access**(see section xlOpenPort on page 38)multiple K-Line applications cannot use a common physical K-Line channel at the same time.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel parameters can be changed/configured
- K-Line messages can be transmitted on the channel
- K-Line messages can be received on the channel

Without init access

- Not supported. If the application gets no **init access** on a specific channel, no further function call is possible on the according channel.



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

9.2 Flowchart

Calling sequence

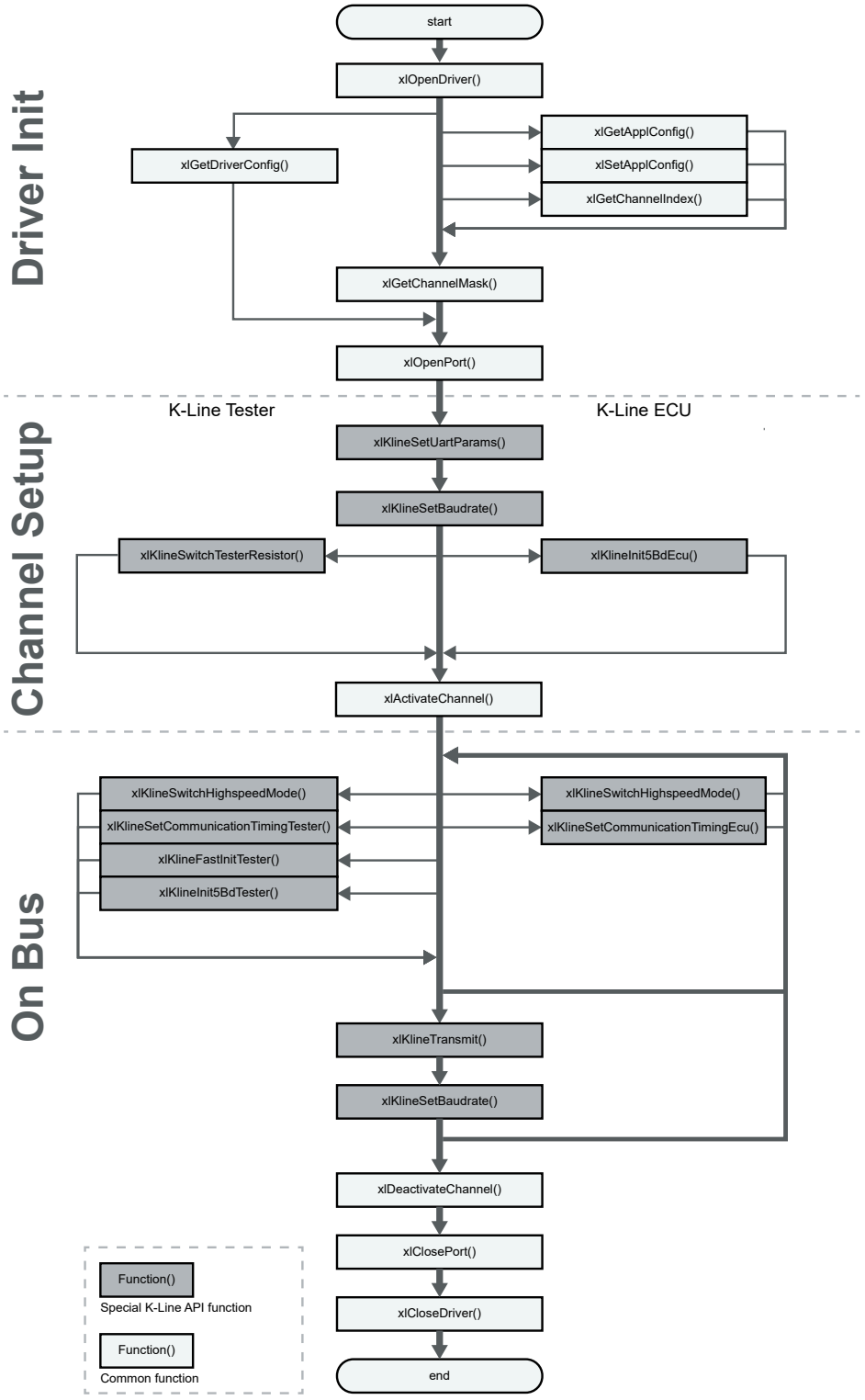


Figure 40: Function calls for K-Line applications

9.3 Functions

9.3.1 xlKlineFastInitTester

Syntax

```
XLstatus xlKlineFastInitTester (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int length,
  unsigned char *data,
  XLklineInitTester *pxlKlineInitTester
)
```

Description

Execute fast init sequence followed by a data frame (tester). Called after activating the channel.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **length**
  See xlKlineTransmit on page 211.
- **data**
  See xlKlineTransmit on page 211.
- **pxlKlineInitTester**
  Pointer to the Tester structure (see XLklineInitTester on page 213).

Return value

Returns an error code (see section Error Codes on page 360).

9.3.2 xlKlineInit5BdEcu

Syntax

```
XLstatus xlKlineInit5BdEcu (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLkline5BdEcu *pxlKline5BdEcu
)
```

Description

Configure the 5Bd-init response frame per ECU. Can be called only once per ECU before activating the channel.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlKline5BdEcu**
  See XLkline5BdEcu on page 212.

Return value Returns an error code (see section Error Codes on page 360).

9.3.3 xlKlineInit5BdTester

Syntax

```
XLstatus xlKlineInit5BdTester (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLkline5BdTester *pxlKline5BdTester
)
```

Description Execute 5Baud init pattern (tester). Called after activating the channel.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlKline5BdTester**
  See XLkline5BdTester on page 213.

Return value Returns an error code (see section Error Codes on page 360).

9.3.4 xlKlineSetBaudrate

Syntax

```
XLstatus xlKlineSetBaudrate (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int baudrate
)
```

Description Sets the baudrate of the tester or the ECU.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **baudrate**
  In baud. 200 Bd …115.2 kBd.

Return value Returns an error code (see section Error Codes on page 360).

9.3.5 xlKlineSetCommunicationTimingEcu

Syntax

```
XLstatus xlKlineSetCommunicationTimingEcu (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLklineSetComEcu *pxlKlineSetComEcu
```

```
)
```

Description

Setup the ECU communication parameters. It returns a `KLINE_CONFIRMATION` event (see K-Line Confirmation Event on page 216).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlKlineSetComEcu**
  Pointer to the Tester structure (see XLklineSetComEcu on page 214).

Return value

Returns an error code (see section Error Codes on page 360).

9.3.6 xlKlineSetCommunicationTimingTester

Syntax

```
XLstatus xlKlineSetCommunicationTimingTester (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLklineSetComTester *pxlKlineSetComTester
)
```

Description

Setup the tester communication parameters. It returns a `KLINE_CONFIRMATION` event (see K-Line Confirmation Event on page 216).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlKlineSetComTester**
  Pointer to the structure (see XLklineSetComTester on page 214).

Return value

Returns an error code (see section Error Codes on page 360).

9.3.7 xlKlineSetUartParams

Syntax

```
XLstatus xlKlineSetUartParams (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLklineUartParameter *pxlKlineUartParams
)
```

Description

Sets up the UART parameters.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlKlineUartParams**
  See XLklineUartParameter on page 215

Return value — Returns an error code (see section Error Codes on page 360).

9.3.8 xlKlineSwitchHighspeedMode

Syntax

```
XLstatus xlKlineSwitchHighspeedMode (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int trxMode
)
```

Description — Switches the high-speed mode for LINPiggy7269 on or off.Required for baudrates >30kBd. It is an asynchronous command and returns a K-Line confirmation event.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **trxMode**
  Select the speed mode:
  - XL_KLINE_TRXMODE_NORMAL
  - XL_KLINE_TRXMODE_HIGHSPEED

Return value — Returns an error code (see section Error Codes on page 360).

9.3.9 xlKlineSwitchTesterResistor

Syntax

```
XLstatus xlKlineSwitchTesterResistor (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int testerR
)
```

Description — Switches the tester resistor for K-Line on or off.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

- **testerR**
  Select the appropriate flag:
  - XL_KLINE_TESTERRESISTOR_ON
  - XL_KLINE_TESTERRESISTOR_OFF

Return value — Returns an error code (see section Error Codes on page 360).

9.3.10 xlKlineTransmit

Syntax

```
XLstatus xlKlineTransmit (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int length,
  unsigned char *data
)
```

Description — Transmits a complete K-Line frame. Can also be used to send a single byte.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **length**
  Data buffer length in bytes. Set to ‘1’ to send a single byte of a K-Line frame. The maximum length is limited to 300 bytes.
- **data**
  K-Line data buffer.Contains a complete K-Line TX frame, e.g.:

```
unsigned char fmtByte;
unsigned char srcAddr;
unsigned char trgAddr;
unsigned char length;
unsigned char data[frame_length]
unsigned char checksum;
```

Return value — Returns an error code (see section Error Codes on page 360).

9.4 Structs

9.4.1 XLkline5BdEcu

Syntax

```
struct s_xl_kline_init_5BdEcu {
  unsigned int configure;
  unsigned ing addr;
  unsigned int rate5bd;
  unsigned int syncPattern;
  unsigned int W1;
  unsigned int W2;
  unsigned int W3;
  unsigned int W4;
  unsigned int W4min;
  unsigned int W4max;
  unsigned int kb1;
  unsigned int kb2;
  unsigned int addrNot;
  unsigned int reserved;
} XLkline5BdEcu;
```

Parameters

- **configure**
  Flag to configure / unconfigure the ECU:
  - `XL_KLINE_UNCONFIGURE_ECU`
  - `XL_KLINE_CONFIGURE_ECU`
- **addr**
  ECU address.
- **rate5bd**
  Baudrate of initial communication: 4.6 ... 5.4 Bd in 1/10 Baud (46 to 54).
- **W1**
  In 1 µs (resolution 10 µs).
- **W2**
  In 1 µs (resolution 10 µs).
- **W3**
  In 1 µs (resolution 10 µs).
- **W4**
  In 1 µs (resolution 10 µs).
- **W4min**
  In 1 µs (resolution 10 µs).
- **W4max**
  In 1 µs (resolution 10 µs).
- **kb1**
  Key byte 1.
- **kb2**
  Key byte 2.
- **addrNot**
  Inverted ECU address with flag | `XL_KLINE_FLAG_TAKE_ADDRNOT`.
- **reserved**
  For future use.

9.4.2 XLkline5BdTester

Syntax

```
struct s_xl_kline_init_5BdTester {
  unsigned int addr;
  unsigned int rate5bd;
  unsigned int W1min;
  unsigned int W1max;
  unsigned int W2min;
  unsigned int W2max;
  unsigned int W3min
  unsigned int W3max;
  unsigned int W4;
  unsigned int W4min;
  unsigned int W4max;
  unsigned int kb2Not;
  unsigned int reserved;
} XLkline5BdTester;
```

Parameters

- **addr**
  ECU address.
- **rate5bd**
  Baudrate of initial communication: 4.6 Bd ... 5.4 Bd in 1/10 Baud (46 to 54).
- **W1min**
  In 1 µs (resolution 10 µs).
- **W1max**
  In 1 µs (resolution 10 µs).
- **W2min**
  In 1 µs (resolution 10 µs).
- **W2max**
  In 1 µs (resolution 10 µs).
- **W3min**
  In 1 µs (resolution 10 µs).
- **W3max**
  In 1 µs (resolution 10 µs).
- **W4**
  In 1 µs (resolution 10 µs).
- **W4min**
  In 1 µs (resolution 10 µs).
- **W4max**
  In 1 µs (resolution 10 µs).
- **kb2Not**
  Inverted key byte 2 with flag `| XL_KLINE_FLAG_TAKE_KB2NOT`.
- **reserved**
  For future use.

9.4.3 XLklineInitTester

Syntax

```
struct s_xl_kline_init_tester {
  unsigned int TiniL;
  unsigned int Twup;
  unsigned int reserved;
```

```
} XLklineInitTester;
```

Parameters

- **TiniL**
  Low phase of the wake-up pattern in 1 µs (resolution 10 µs).
- **Twup**
  Length of the wake-up pattern in 1 µs (resolution 10 µs)
- **reserved**
  For future use.

9.4.4 XLklineSetComEcu

Syntax

```
struct s_xl_kline_set_com_ecu {
  unsigned int P1;
  unsigned int P4min;
  unsigned int TinilMin;
  unsigned int TinilMax;
  unsigned int TwupMin
  unsigned int TwupMax;
  unsigned int reserved;
} XLklineSetComEcu;
```

Parameters

- **P1**
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.
- **P4min**
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.
- **TinilMin**
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.
- **TinilMax**
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.
- **TwupMin**
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.
- **TwupMax**
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.
- **reserved**
  For future use.

9.4.5 XLklineSetComTester

Syntax

```
struct s_xl_kline_set_com_tester {
  unsigned int P1min;
  unsigned int P4;
  unsigned int reserved;
} XLklineSetComTester;
```

Parameters

- **P1min**
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.
- **P4**
  In 1 µs (resolution 10 µs).
- **reserved**
  For future use.

9.4.6 XLklineUartParameter

Syntax

```
struct s_xl_kline_uart_params {
  unsigned int databits;
  unsigned int stopbits;
  unsigned int parity;
} XLklineUartParameter;
```

Parameters

- **databits**
  Number of data bits (7, 8 or 9).
- **stopbits**
  Number of stop bits (1 or 2).
- **parity**
  - XL_XLINE_UART_PARITY_NONE
  - XL_KLINE_UART_PARITY_EVEN
  - XL_KLINE_UART_PARITY_ODD

9.5 Events

9.5.1 K-Line Data

Syntax

```
struct s_xl_kline_data {
  unsigned int klineEvtTag;
  unsigned int reserved;

  union {
    XL_KLINE_RX_DATA klineRx;
    XL_KLINE_TX_DATA klineTx;
    XL_KLINE_TESTER_5BD klineTester5Bd;
    XL_KLINE_ECU_5BD klineEcu5Bd;
    XL_KLINE_TESTER_FI_WU_PATTERN klineTesterFiWu;
    XL_KLINE_ECU_FI_WU_PATTERN klineEcuFiWu;
    XL_KLINE_CONFIRMATION klineConfirmation;
    XL_KLINE_ERROR klineError;
  } data;
}; XL_KLINE_DATA;
```

Description

Frame for all K-Line events.

Tag

XL_KLINE_MSG

Parameters

► **klineEvtTag**

- `XL_KLINE_EVT_RX_DATA`
  Tag indicating a received data frame.
- `XL_KLINE_EVT_TX_DATA`
  Tag indicating a transmitted data frame.
- `XL_KLINE_EVT_TESTER_5BD`
  5Bd init related event (tester).
- `XL_KLINE_EVT_ECU_5BD`
  5Bd init related event (ECU).
- `XL_KLINE_EVT_TESTER_FI_WU_PATTERN`
  Tag indicating a fast init wake-up pattern (tester).
- `XL_KLINE_EVT_ECU_FI_WU_PATTERN`
  Tag indicating a fast init wake-up pattern (ECU).
- `XL_KLINE_EVT_ERROR`
  Tag indicating a K-Line communication error.
- `XL_KLINE_EVT_CONFIRMATION`
  Confirmation event for asynchronous commands.

9.5.2 K-Line Confirmation Event

Syntax

```
typedef struct s_xl_kline_confirmation {
  unsigned int channel;
  unsigned int confTag;
  unsigned int result;
} XL_KLINE_CONFIRMATION;
```

Description

Confirmation event for asynchronous commands.

Tag

XL_KLINE_EVT_CONFIRMATION

Parameters

- **channel**
  K-Line channel.
- **confTag**
  The confirmation tag:
  - `XL_KLINE_EVT_TAG_SET_COMM_PARAM_TESTER`
    Indicates (asynchronously) the result of setting the communication timing parameters (tester).
  - `XL_KLINE_EVT_TAG_COMM_PARAM_ECU`
    Indicates (asynchronously) the result of setting the communication timing parameters (ECU).
  - `XL_KLINE_EVT_TAG_SWITCH_HIGHSPEED`
    Indicates (asynchronously) the result of setting the high speed mode.
- **result**
  Success or error (0,1).

9.5.3 K-Line Error Event

Syntax

```
struct s_xl_kline_error {
  unsigned int klineErrorTag;
  unsigned int reserved;

  union {
    XL_KLINE_ERROR_RXTX rxtxErr;
    XL_KLINE_ERROR_TESTER_5BD tester5BdErr;
    XL_KLINE_ERROR_ECU_5BD ecu5BdErr;
    XL_KLINE_ERROR_IBS ibsErr;
    unsigned int reserved[4];
  } data;
}; XL_KLINE_ERROR;
```

Description

Event indicating a K-Line communication error.

Tag

XL_KLINE_EVT_ERROR

Parameters

- **klineErrorTag**
  - `XL_KLINE_ERROR_TYPE_RXTX_ERROR`
    Indicates byte level errors.
  - `XL_KLINE_ERROR_TYPE_5BD_TESTER`
    Indicates timing errors during 5Bd init detected by the tester.
  - `XL_KLINE_ERROR_TYPE_5BD_ECU`
    Indicates timing errors during 5Bd init detected by the ECU.
  - `XL_KLINE_ERROR_TYPE_IBS`
    Indicates an inter-byte space timing error during regular communication.
  - `XL_KLINE_ERROR_TYPE_FI`
    Indicates timing errors during fast init.

9.5.4 K-Line ECU 5Bd Error

Syntax

```
typedef struct s_xl_kline_error_5bd_ecu {
  unsigned int ecu5BdErr;
} XL_KLINE_ERROR_ECU_5BD;
```

Description

Sub structure of `XL_KLINE_ERROR` with details on timing errors during 5Bd init

detected by the ECU.

Tag XL_KLINE_ERROR_TYPE_5BD_ECU

Parameters

► **ecu5BdErr**

- XL_KLINE_ERR_ECU_W4MIN
  Timing violation of W4min during 5Bd init.
- XL_KLINE_ERR_ECU_W4MAX
  Timing violation of W4max during 5Bd init.

9.5.5 K-Line Tester 5Bd Error

Syntax

```
typedef struct s_xl_kline_error_5bd_tester {
 unsigned int tester5BdErr;
} XL_KLINE_ERROR_TESTER_5BD;
```

Description Sub structure of XL_KLINE_ERROR with details on timing errors during 5Bd init detected by the tester.

Tag XL_KLINE_ERROR_TYPE_5BD_TESTER

Parameters

► **tester5BdErr**

- XL_KLINE_ERR_TESTER_W1MIN
  Timing violation of W1min during 5Bd init.
- XL_KLINE_ERR_TESTER_W1MAX
  Timing violation of W1max during 5Bd init.
- XL_KLINE_ERR_TESTER_W2MIN
  Timing violation of W2min during 5Bd init.
- XL_KLINE_ERR_TESTER_W2MAX
  Timing violation of W2max during 5Bd init.
- XL_KLINE_ERR_TESTER_W3MIN
  Timing violation of W3min during 5Bd init.
- XL_KLINE_ERR_TESTER_W3MAX
  Timing violation of W3max during 5Bd init.
- XL_KLINE_ERR_TESTER_W4MIN
  Timing violation of W4min during 5Bd init.
- XL_KLINE_ERR_TESTER_W4MAX
  Timing violation of W4max during 5Bd init.

9.5.6 K-Line ibsErr Error

Syntax

```
typedef struct s_xl_kline_error_ibs {
  unsigned int ibsErr;
  unsigned int rxtxErrData;
} XL_KLINE_ERROR_IBS;
```

Description Sub structure of XL_KLINE_ERROR with details on inter-byte space timing errors during regular communication.

Tag XL_KLINE_ERROR_TYPE_IBS

Parameters

► **ibsErr**

- XL_KLINE_ERR_IBS_P1
  Timing violation of P1 during regular communication.
- XL_KLINE_ERR_IBS_P4
  Timing violation of P4 during regular communication.

▶ **rxtxErrData**

- XL_KLINE_ERR_RXTX_UA
  Unexpected activity.

9.5.7 K-Line RXTX Error

Syntax

```
typedef struct s_xl_kline_error_rxtx {
  unsigned int rxtxErrData;
} XL_KLINE_ERROR_RXTX;
```

Description

Sub structure of XL_KLINE_ERRORwith details on byte level errors.

Tag

XL_KLINE_ERROR_TYPE_RXTX_ERROR

Parameters

▶ **rxtxErrData**

- XL_KLINE_ERR_RXTX_UA
  Unexpected activity.
- XL_KLINE_ERR_RXTX_MA
  Missing activity.
- XL_KLINE_ERR_RXTX_ISB
  Invalid sync byte during 5Bd init.

9.5.8 K-Line RX Data

Syntax

```
struct s_xl_kline_rx_data {
  unsigned int timeDiff;
  unsigned int data;
  unsigned int error;
} XL_KLINE_RX_DATA;
```

Description

Event for a data frame.

Tag

XL_KLINE_EVT_RX_DATA

Parameters

▶ **timeDiff**
Difference between EOF and SOF timestamps in nano seconds.

▶ **data**
Data field containing up to 9 bits.

▶ **error**
Error mask:

- XL_KLINE_BYTE_PARITY_ERROR_MASK
  Parity check of the data field failed,
  e.g. transmission error, parity settings mismatch.
- XL_KLINE_BYTE_FRAMING_ERROR_MASK
  Framing error detected,
  e.g. wrong number of data or stop bits.

9.5.9 K-Line Tester 5Bd

Syntax

```
struct s_xl_kline_tester_5bd {
  unsigned int tag5bd;
  unsigned int timeDiff;
  unsigned int data;
} XL_KLINE_TESTER_5BD;
```

Description

5Bd init related event (tester).

Tag

XL_KLINE_EVT_TESTER_5BD

Parameters

- **tag5Bd**
  - `XL_KLINE_EVT_TAG_5BD_ADDR`
    Event tag for ECU address during 5Bd init.
  - `XL_KLINE_EVT_TAG_5BD_BAUDRATE`
    Event tag indicating regular communication baudrate after 5Bd init.
  - `XL_KLINE_EVT_TAG_5BD_KB1`
    Event tag indicating key byte 1 during 5Bd init.
  - `XL_KLINE_EVT_TAG_5BD_KB2`
    Event tag indicating key byte 2 during 5Bd init.
  - `XL_KLINE_EVT_TAG_5BD_KB2NOT`
    Event tag indicating inverted key byte 2 during 5Bd init.
  - `XL_KLINE_EVT_TAG_5BD_ADDRNOT`
    Event tag indicating inverted ECU address during 5Bd init.
- **timeDiff**
  Difference between EOF and SOF timestamps in nano seconds.
- **data**
  Data byte or address depending on `tag5bd`.

9.5.10 K-Line ECU Fastinit WU Pattern

Syntax

```
struct s_xl_kline_ecu_fastinit_wu_pattern {
  unsigned int timeDiff;
  unsigned int fastInitEdgeTimeDiff;
} XL_KLINE_ECU_FI_WU_PATTERN;
```

Description

Event indicating a fast init wake-up pattern (ECU).

Tag

XL_KLINE_EVT_ECU_FI_WU_PATTERN

Parameters

- **timeDiff**
  Difference between EOF and SOF timestamps in nano seconds.
- **fastInitEdgeTimeDiff**
  In nano seconds (Twup - TiniL).

9.5.11 K-Line Tester Fastinit WU Pattern

Syntax

```
struct s_xl_kline_tester_fastinit_wu_pattern {
  unsigned int timeDiff;
  unsigned int fastInitEdgeTimeDiff;
} XL_KLINE_TESTER_FI_WU_PATTERN;
```

Description Event indicating a fast init wake-up pattern (tester).

Tag XL_KLINE_EVT_TESTER_FI_WU_PATTERN

Parameters
- **timeDiff**
  Difference between EOF and SOF timestamps in nano seconds.
- **fastInitEdgeTimeDiff**
  In nano seconds (Twup - TiniL).

9.5.12 K-Line TX Data

Syntax

```
struct s_xl_kline_tx_data {
  unsigned int timeDiff;
  unsigned int data;
  unsigned int error;
} XL_KLINE_TX_DATA;
```

Description Event for a transmitted data frame.

Tag XL_KLINE_EVT_TX_DATA

Parameters
- **timeDiff**
  Difference between EOF and SOF timestamps in nano seconds.
- **data**
  Data field containing up to 9 bits.
- **error**
  Error mask:
  - `XL_KLINE_BYTE_PARITY_ERROR_MASK`
    Parity check of the data field failed,
    e.g. transmission error, parity settings mismatch.
  - `XL_KLINE_BYTE_FRAMING_ERROR_MASK`
    Framing error detected,
    e.g. wrong number of data or stop bits.

10 D/A IO Commands (IOcab)

In this chapter you find the following information:

10.1 Introduction **223**
10.2 Flowchart **224**
10.3 Functions **225**
10.4 Events **231**
10.5 Application Examples **233**

10.1 Introduction

Description

The XL Driver Library enables the development of DAIO applications for the Vector IOcab 8444opto.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel parameters can be changed/configured
- DAIO lines can be set
- DAIO lines can be read

Without init access

- DAIO lines can be read



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

10.2 Flowchart

Calling sequence

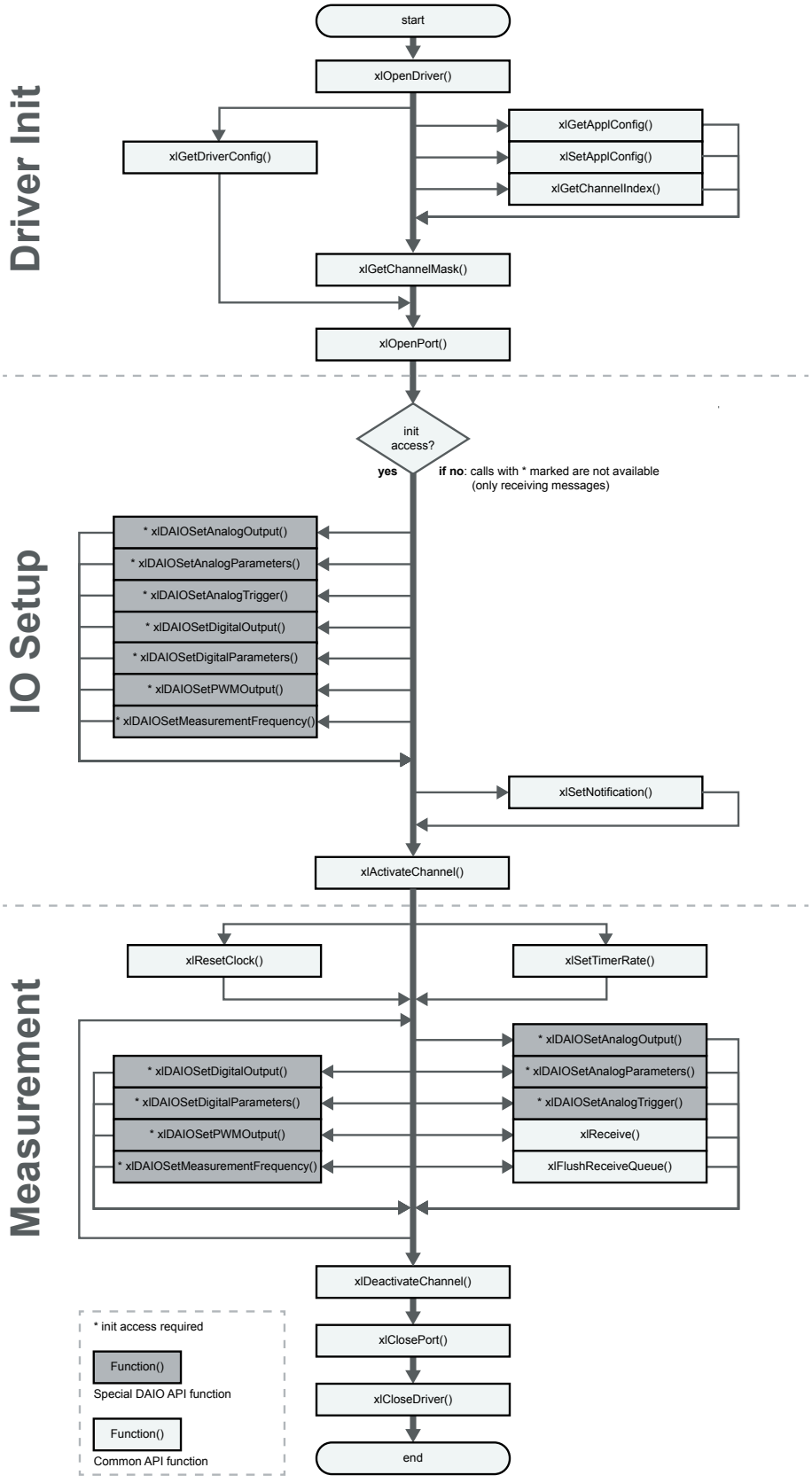


Figure 41: Function calls for DAIO applications

10.3 Functions

10.3.1 xlDAIOSetAnalogParameters

Syntax

```
XLstatus xlDAIOSetAnalogParameters (
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int inputMask,
  unsigned int outputMask,
  unsigned int highRangeMask)
```

Description

Configures the analog lines. All lines are set to input by default. The bit sequence to access the physical pins on the D-SUB15 connector is as follows:

- AIO0 = 0001 (0x01)
- AIO1 = 0010 (0x02)
- AIO2 = 0100 (0x04)
- AIO3 = 1000 (0x08)

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **inputMask**
  Mask for lines to be configured as input. Generally the inverted value of the output mask can be used.
- **outputMask**
  Mask for lines to be configured as output. Generally the inverted value of the input mask can be used.
- **highRangeMask**
  Mask for lines that should use high range mask for input resolution.
  - Low range 0 … 8.192 V (3.1 kHz)
  - High range 0 … 32.768 V (6.4 kHz)
  Line AIO0 and AIO1 supports both ranges, AIO2 and AIO3 high range only.

Return value

Returns an error code (see section Error Codes on page 360).

Example

Setting up the IOcab8444 with four analog lines and two different ranges

- `inputMask = 0x01(0b0001)`
  analogLine1 → input
  analogLine2 → not input
  analogLine3 → not input
  analogLine4 → not input
- `outputMask = 0x0E(0b1110)`
  analogLine1 → not output
  analogLine2 → output
  analogLine3 → output
  analogLine4 → output
- `highRangeMask = 0x01(0b0001)`
  analogLine1 → high range
  analogLine2 → low range
  analogLine3 → high range (always)
  analogLine4 → high range (always)

10.3.2 xlDAIOSetAnalogOutput

Syntax

```
XLstatus xlDAIOSetAnalogOutput (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int analogLine1,
  unsigned int analogLine2,
  unsigned int analogLine3,
  unsigned int analogLine4)
```

Description

Sets analog output line to voltage level as requested (specified in millivolts). Optionally, the flag `XL_DAIO_IGNORE_CHANNEL` can be used not to change line's current level.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **analogLine1**
  Voltage level for AIO0.
- **analogLine2**
  Voltage level for AIO1.
- **analogLine3**
  Voltage level for AIO2.
- **analogLine4**
  Voltage level for AIO3.

Return value

Returns an error code (see section Error Codes on page 360).

10.3.3 xlDAIOSetAnalogTrigger

Syntax

```
XLstatus xlDAIOSetAnalogTrigger (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int triggerMask,
  unsigned int triggerLevel,
  unsigned int triggerEventMode)
```

Description

Configures analog trigger functionality.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **triggerMask**
  Line to be used as trigger input. Currently the analog trigger is only supported by line AIO3 of the IOcab 8444opto (mask = 0b1000).
- **triggerLevel**
  Voltage level (in millivolts) for the trigger.
- **triggerEventMode**
  One of following options can be set:

  `XL_DAIO_TRIGGER_MODE_ANALOG_ASCENDING`
  Triggers when descending voltage level falls under `triggerLevel`

  `XL_DAIO_TRIGGER_MODE_ANALOG_DESCENDING`
  Triggers when descending voltage level goes over `triggerLevel`

  `XL_DAIO_TRIGGER_MODE_ANALOG`
  Triggers when the voltage level falls under or goes over `triggerLevel`

Return value

Returns an error code (see section Error Codes on page 360).

10.3.4 xlDAIOSetDigitalParameters

Syntax

```
XLstatus xlDAIOSetDigitalParameters (
XLportHandle portHandle,
XLaccess accessMask,
unsigned int inputMask,
unsigned int outputMask)
```

Description

Configures the digital lines. All lines are set to input by default. The bit sequence to access the physical pins on the D-SUB15 connector is as follows:

- DAIO0: `0b00000001`
- DAIO1: `0b00000010`
- DAIO2: `0b00000100`
- DAIO3: `0b00001000`

- DAIO4: 0b000**1**0000
- DAIO5: 0b00**1**00000
- DAIO6: 0b0**1**000000
- DAIO7: 0b**1**0000000

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **inputMask**
  Mask for lines to be configured as input. Generally the inverted value of the output mask will be used.
- **outputMask**
  Mask for lines to be configured as output. A set output line affects always a defined second digital line.



WARNING!
The digital outputs consist internally of electronic switches (photo MOS relays) and need always two digital lines of the IOcab 8444opto: a general output line and a line for external supply. In other words: When the switch is closed (by software), the applied voltage can be measured at the second output line, otherwise not. The line pairs are defined as follows: DIO0/DIO1, DIO2/DIO3, DIO4/DIO5 and DIO6/DIO7.

Return value

Returns an error code (see section Error Codes on page 360).

10.3.5 xlDAIOSetDigitalOutput

Syntax

```
XLstatus xlDAIOSetDigitalOutput (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int outputMask,
  unsigned int valuePattern)
```

Description

Sets digital output line to desired logical level.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

- **outputMask**
  Switches to be changed:
  - DAIO0/DAIO1: 0b000**1**
  - DAIO2/DAIO3: 0b00**1**0
  - DAIO4/DAIO5: 0b0**1**00
  - DAIO6/DAIO7: 0b**1**000

- **valuePattern**
  Mask specifying the switch state for digital output.
  - DAIO0/DAIO1: 0b000**x**
  - DAIO2/DAIO3: 0b00**x**0
  - DAIO4/DAIO5: 0b0**x**00
  - DAIO6/DAIO7: 0b**x**000
  x = 0 (switch opened) or 1 (switch closed)

Return value — Returns an error code (see section Error Codes on page 360).



Example

Setting up IOcab8444

Update digital output DIO0/DIO1 and DIO4/DIO5
```
outputMask = 0x05(0b0101)
```

Close relay DIO0/DIO1, open relay DIO4/DIO5
```
valuePattern = 0x01(0b0001)
```

10.3.6 xlDAIOSetPWMOutput

Syntax

```
XLstatus xlDAIOSetPWMOutput (
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int frequency,
  unsigned int value)
```

Description — Changes PWM output to defined frequency and value.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

- **frequency**
  Set PWM frequency to specified value in Hertz.
  Allowed values: 40…500 Hertz and 2.4 kHz…100 kHz.

- **Value**
  Ratio for pulse high pulse low times with resolution of 0.01 percent.
  Allowed values: 0 (100% pulse low)…10000 (100% pulse high).

Return value — Returns an error code (see section Error Codes on page 360).



Example

Setting up the IOcab8444

Set PWM frequency to 2500 Hz
`frequency = 2500`

Set PWM ratio to 25% (75% pulse low, 25% pulse high)
`value = 2500`

10.3.7 xlDAIOSetMeasurementFrequency

Syntax

```
XLstatus xlDAIOSetMeasurementFrequency (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int measurementInterval)
```

Description

Sets the measurement frequency. `xlEvents` will be automatically triggered, which can be received by `xlReceive`. For manual trigger, see section xlDAIORequestMeasurement on page 230.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **measurementInterval**
  Measurement frequency in ms.

Return value

Returns an error code (see section Error Codes on page 360).

10.3.8 xlDAIORequestMeasurement

Syntax

```
XLstatus xlDAIORequestMeasurement (
  XLportHandle portHandle,
  XLaccess     accessMask)
```

Description

Forces manual measurement of DAIO values.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

Return value

Returns an error code (see section Error Codes on page 360).

10.4 Events

10.4.1 XL DAIO Data

Syntax

```
struct s_xl_daio_data {
  unsigned short flags;
  unsigned int   timestamp_correction;
  unsigned char  mask_digital;
  unsigned char  value_digital;
  unsigned char  mask_analog;
  unsigned char  reserved0;
  unsigned short value_analog[4];
  unsigned int   pwm_frequency;
  unsigned short pwm_value;
  unsigned int   reserved1;
  unsigned int   reserved2;
};
```

Tag

`XL_DAIO_DATA` (see section XLevent on page 70).

Parameters

- **flags**
  Flags describing valid fields in the event structure:
  `XL_DAIO_DATA_GET`
  Structure contains valid received data.

  `XL_DAIO_DATA_VALUE_DIGITAL`
  Digital values are valid.

  `XL_DAIO_DATA_VALUE_ANALOG`
  Analog values are valid.

  `XL_DAIO_DATA_PWM`
  PWM values are valid.
- **timestamp_correction**
  Value to correct time stamp in this event (in order to get real time of measurement). In order to get real time of measurement substract this value from event's time stamp. Value is in nanoseconds.
- **mask_digital**
  Mask of digital lines that contains valid value in this event.
- **value_digital**
  Value of digital lines specified by `mask_digital` parameter.
- **mask_analog**
  Mask of analog lines that contains valid value in this event.
- **reserved**
  Reserved for future use. Set to 0.
- **value_analog**
  Array of measured analog values for analog lines specified by `mask_analog` parameter. Value is in millivolts.
- **pwm_frequency**
  Measured capture frequency in Hz.

- **pwm_value**
  Measured capture value in percent.
- **reserved1**
  Reserved for future use. Set to 0.
- **reserved2**
  Reserved for future use. Set to 0.

10.5 Application Examples

10.5.1 xlDAIOexample

10.5.1.1 General Information

Description

This example demonstrates the setup of a single IOcab 8444opto for a test, and the way of accessing the inputs and outputs for cyclically measurement.

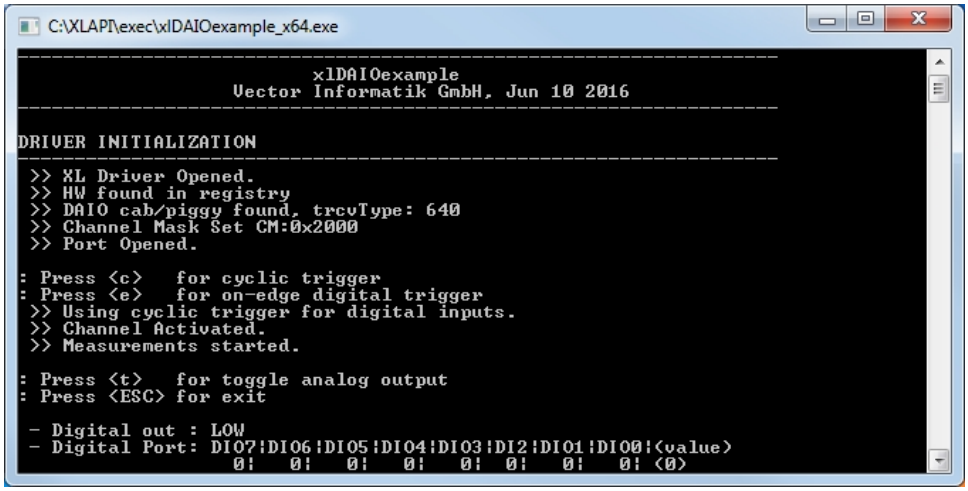


Figure 42: Running xlDAIOexample

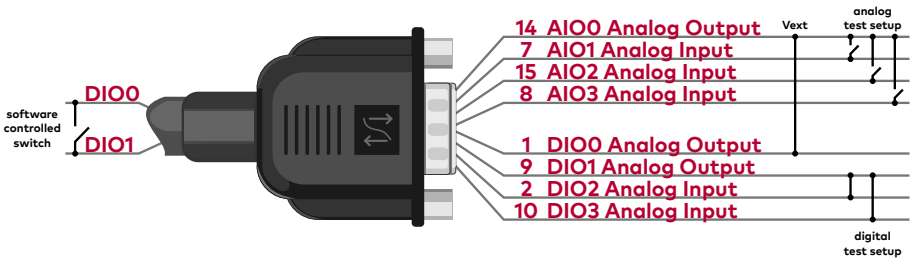
10.5.1.2 Setup

Pin definition

The following pins of the IOcab 8444opto are used in this example:

Signal	Pin	Type
AIO0	14	Analog output
AIO1	7	Analog input
AIO2	15	Analog input
AIO3	8	Analog input
DIO0	1	Digital output (shared electronic switch with DIO1).
DIO1	9	Digital output (supplied by DIO0, when switch is closed).
DIO2	2	Digital input.
DIO3	10	Digital input.

Setup



Note
The internal switch between DIO0 (supplied by AIO0) and DIO1 is closed/opened with xlDAIOSetDigitalOutput(). If the switch is closed, the applied voltage at DIO0 can be measured at DIO1.

10.5.1.3 Keyboard commands

The running application can be controlled via the following keyboard commands:

Key	Command
<ENTER>	Toggle digital output.
<x>	Closes application.

10.5.1.4 Output Examples

Example

```
AIO0:             4032mV
AIO1:             0mV
AIO2:             0mV
AIO3:             0mV
Switch selected: DIO0/DI01
Switch states: OPEN
Digital Port:    DIO7 DIO6 DIO5 DIO4 DIO3 DIO2 DIO1 DIO0 val
                    0    0    0    0    0    0    0    1 (1)
```

Explanation

- “AIO0” displays 4032mV, since it is set to output with maximum output level.
- "AIO1” displays 0mV, since there is no applied voltage at this input.
- “AIO2” displays 0mV, since there is no applied voltage at this input.
- “AIO3” displays 0mV, since there is no applied voltage at this input.
- ”Switch selected” displays DIO0/DIO1 (first switch)
- ”Switch states” displays the state of switch between DIO0/DIO1
- “Digital Port” shows the single states of DIO7…DIO0:
  - DIO0: displays ‘1’ (always ‘1’, due the voltage supply)
  - DIO1: displays ‘0’ (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays ‘0’ (output of DIO1)
  - DIO3: displays ‘0’ (output of DIO1)
  - DIO4: displays ‘0’ (n.c.)
  - DIO5: displays ‘0’ (n.c.)
  - DIO6: displays ‘0’ (n.c.)
  - DIO7: displays ‘0’ (n.c.)



Example

```
AIO0:             4032mV
AIO1:             0mV
AIO2:             4032mV
AIO3:             0mV
Switch selected: DIO0/DI01
Switch states: CLOSED
Digital Port:    DIO7 DIO6 DIO5 DIO4 DIO3 DIO2 DIO1 DIO0 val
                    0    0    0    0    1    1    1    1 (1)
```

Explanation

- "AIO0" displays 4032mV, since it is set to output with maximum output level.
- "AIO1" displays 0mV, since there is no applied voltage at this input.
- "AIO0" displays 4032mV, since it is connected to AIO0.
- "AIO3" displays 0mV, since there is no applied voltage at this input.
- "Switch selected" displays DIO0/DIO1 (first switch)
- "Switch state" displays the state of switch between DIO0/DIO1
- "Digital Port" shows the single states of DIO7…DIO0:
  - DIO0: displays '1' (always '1', due the voltage supply)
  - DIO1: displays '1' (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays '1' (output of DIO1)
  - DIO3: displays '1' (output of DIO1)
  - DIO4: displays '0' (n.c.)
  - DIO5: displays '0' (n.c.)
  - DIO6: displays '0' (n.c.)
  - DIO7: displays '0' (n.c.)

10.5.1.5 Functions

Description

- **InitIOcab**
  This function opens the driver and reads the current hardware configuration. A valid `channelMask` is calculated and one port is opened afterwards.
- **ToggleSwitch**
  This function toggles all switches and passes through the applied voltage at DIO0 to DIO1.
- **CloseExample**
  Closes the driver and the application.

10.5.2 xlDAIOdemo

10.5.2.1 General Information

Description

This example demonstrates the basic digital/analog IO handling with the **XL Driver Library**. To run the application, one connected IOcab 8444opto is needed.

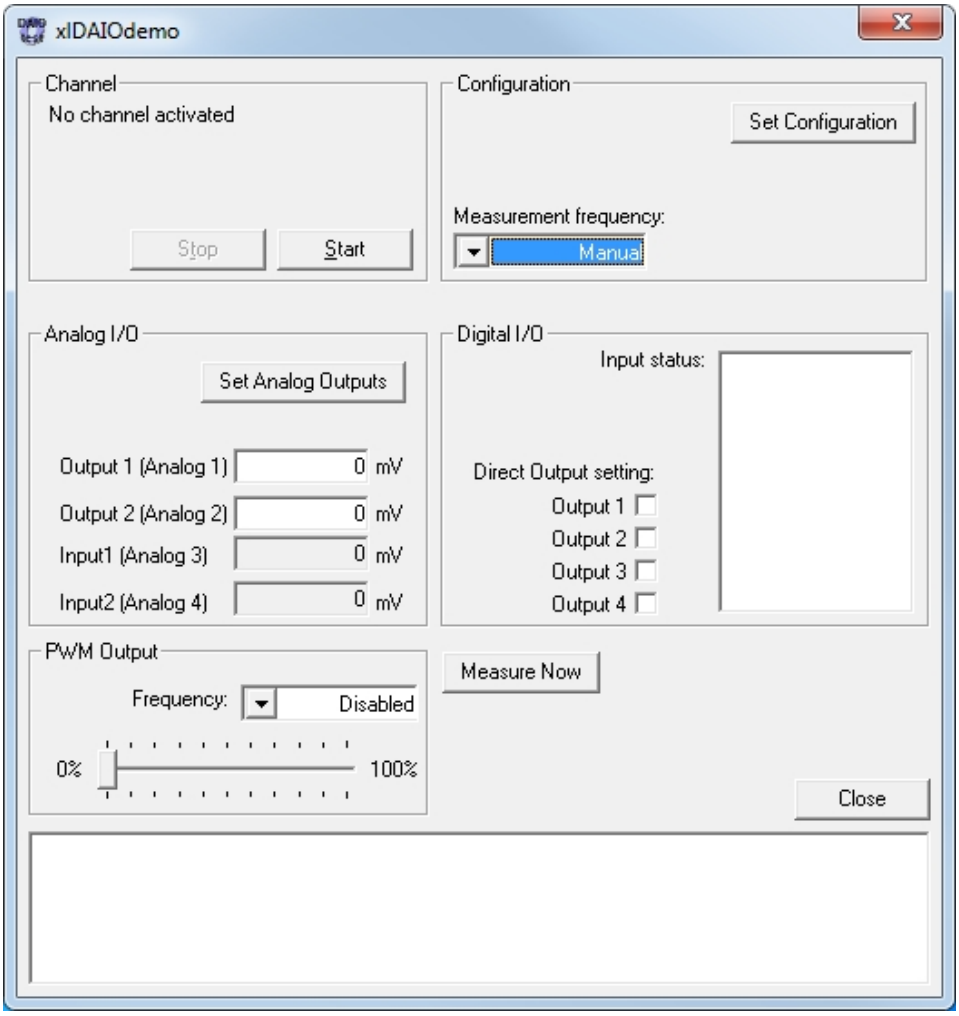


Figure 43: xlDAIOdemo

10.5.2.2 Classes

Description

The example has the following class structure:

- **CXIDAIOdemoApp**
  Main MFC class → `xlDAIOdemo.cpp`
- **CXIDAIOdemoDlg**
  Handles the window dialog messages and control the IOcab → `xlDAIO-demoDlg.cpp`
- **ReceiveThread**
  Thread to handle the DAIO events.

11 D/A IO Commands (Piggybacks)

In this chapter you find the following information:

11.1 Introduction **238**
11.2 Flowchart **239**
11.3 Functions **240**
11.4 Structs **244**
11.5 Events **249**

11.1 Introduction

Description

The XL Driver Library enables the development of DAIO applications for the Vector IOpiggy 8642 and the VNpiggy30 IO 8644.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel parameters can be changed/configured
- DAIO lines can be set



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

11.2 Flowchart

Calling sequence

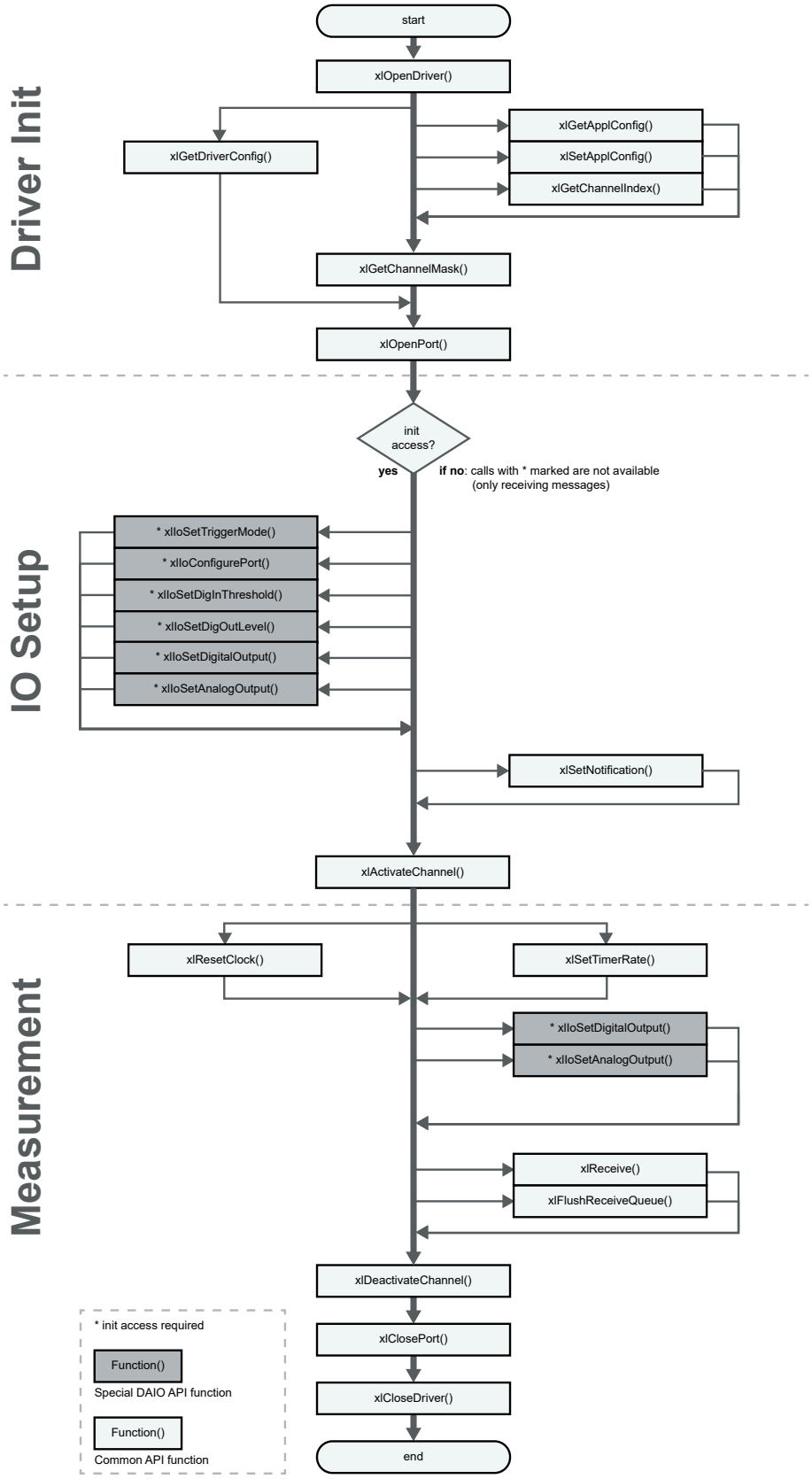


Figure 44: Function calls for DAIO (IOpiggy) applications

11.3 Functions

11.3.1 xlIoSetTriggerMode (IOpiggy)

Syntax

```
XLstatus xlIoSetTriggerMode (
  XLportHandle portHandle,
  XLaccess           accessMask,
  XLdaioTriggerMode* pxlDaioTriggerMode)
```

Description

Sets the DAIO trigger mode for the analog and digital ports.

Note
This command can be called only once per port type (analog and digital) and only when the channel is deactivated (see flowchart in sectionIntroduction on page 252).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlDaioTriggerMode**
  Use this structure to define the trigger type (see section XLdaioTriggerMode on page 244).

Return value

Returns an error code (see section Error Codes on page 360).

11.3.2 xlIoConfigurePorts

Syntax

```
XLstatus xlIoConfigurePorts (
  XLportHandle portHandle,
  XLaccess      accessMask,
  XLdaioSetPort *pxlDaioSetPort)
```

Description

Configures the DAIO ports.

Note
This command can be called only once.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlDaioSetPort**
  Port configuration (see section XLdaioSetPort on page 245).

Return value Returns an error code (see section Error Codes on page 360).

11.3.3 xlIoSetDigInThreshold

Syntax

```
XLstatus xlIoSetDigInThreshold (
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int level)
```

Description Defines the voltage level for logical high and logical low (digital input).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **level**
  Value that defines the voltage level (mV) for the input threshold.

Return value Returns an error code (see section Error Codes on page 360).

11.3.4 xlIoSetDigOutLevel

Syntax

```
XLstatus xlIoSetDigOutLevel(
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int level)
```

Description Defines the voltage level for logical high (digital output).

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **level**
  `XL_DAIO_DO_LEVEL_5V`
  `XL_DAIO_DO_LEVEL_12V`

Return value Returns an error code (see section Error Codes on page 360).

11.3.5 xlIoSetDigitalOutput

Syntax

```
XLstatus xlIoSetDigitalOutput (
  XLportHandle         portHandle,
  XLaccess             accessMask,
  XLdaioDigitalParams* pxlDaioDigitalParams)
```

Description — Configures the digital output.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlDaioDigitalParams**
  Use this structure to set the value of the digital out pin (see section XLdaioDigitalParams (IOpiggy) on page 247).

Return value — Returns an error code (see section Error Codes on page 360).

11.3.6 xlIoSetAnalogOutput

Syntax

```
XLstatus xlIoSetAnalogOutput (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLdaioAnalogParams*  pxlDaioAnalogParams)
```

Description — Configures the analog output.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlDaioAnalogParams**
  Use this structure to set the value of the analog out pin (see section XLdaioAnalogParams on page 247).

Return value — Returns an error code (see section Error Codes on page 360).

11.3.7 xlIoStartSampling

Syntax

```
XLstatus xlIoStartSampling (
  XLportHandle portHandle,
  XLaccess accessMask,
  unsigned int portTypeMask)
```

Description — This command requests DAIO measurement data and is independent of the defined trigger mode.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **portTypeMask**
  `XL_DAIO_PORT_TYPE_MASK_ANALOG`
  `XL_DAIO_PORT_TYPE_MASK_DIGITAL`

Return value — Returns an error code (see section Error Codes on page 360).

11.4 Structs

11.4.1 XLdaioTriggerMode

Syntax

```
typedef struct s_xl_daio_trigger_mode {
  unsigned int portTypeMask;
  unsigned int triggerType;

  union triggerTypeParams {
    unsigned int cycleTime;
      struct {
        unsigned int portMask;
        unsigned int type;
      } digital;
    } param;
} XLdaioTriggerMode;
```

Parameters

- **portTypeMask**
  Defines the port type:
  `XL_DAIO_PORT_TYPE_MASK_ANALOG`
  `XL_DAIO_PORT_TYPE_MASK_DIGITAL`
- **triggerType**
  Defines the trigger type:
  `XL_DAIO_TRIGGER_TYPE_CYCLIC` (for analog and digital port type)
  `XL_DAIO_TRIGGER_TYPE_PORT` (for digital port type)
- **cycleTime**
  For VNpiggy30 IO 8644 a minimum of 100 µs is possible.

  During a cycle additional events may also be fired, e. g. if the digital IO pin toggles.
- **portMask**
  For use with `XL_DAIO_TRIGGER_TYPE_PORT`.
  Specifies the digital port (D0…D07):
  `XL_DAIO_PORT_MASK_DIGITAL_D0`
  `XL_DAIO_PORT_MASK_DIGITAL_D1`
  `XL_DAIO_PORT_MASK_DIGITAL_D2`
  `XL_DAIO_PORT_MASK_DIGITAL_D3`
  `XL_DAIO_PORT_MASK_DIGITAL_D4`
  `XL_DAIO_PORT_MASK_DIGITAL_D5`
  `XL_DAIO_PORT_MASK_DIGITAL_D6`
  `XL_DAIO_PORT_MASK_DIGITAL_D7`
  `XL_DAIO_PORT_MASK_ANALOG_A2`
  `XL_DAIO_PORT_MASK_ANALOG_A3`
- **type**
  For use with `XL_DAIO_TRIGGER_TYPE_PORT`.
  `XL_DAIO_TRIGGER_TYPE_RISING`
  `XL_DAIO_TRIGGER_TYPE_FALLING`
  `XL_DAIO_TRIGGER_TYPE_BOTH`



Example

```
XLstatus xlStatus;
XLportHandle portHandle = ...;
XLaccess mask = ...;
XLdaioTriggerMode xlDaioTmAna;

memset(&xlDaioTmAna, 0x00, sizeof(xlDaioTmAna));
xlDaioTmAna.triggerType = XL_DAIO_TRIGGER_TYPE_CYCLIC;
xlDaioTmAna.portTypeMask = XL_DAIO_PORT_TYPE_MASK_ANALOG;
xlDaioTmAna.param.cycleTime = 50000; // in us
xlStatus = xlIoSetTriggerMode(portHandle, mask, &xlDaioTmAna);
```



Example

```
XLstatus        xlStatus;
XLportHandle portHandle = ...;
XLaccess mask = ...;

XLdaioTriggerMode xlDaioTmDig;
memset(&xlDaioTmDig, 0x00, sizeof(xlDaioTmDig));
xlDaioTmDig.triggerType = XL_DAIO_TRIGGER_TYPE_PORT;

xlDaioTmDig.portTypeMask = XL_DAIO_PORT_TYPE_MASK_DIGITAL;

xlDaioTmDig.param.digital.portMask = XL_DAIO_PORT_MASK_DIGITAL_D4
|
                                     XL_DAIO_PORT_MASK_DIGITAL_
D5;

xlDaioTmDig.param.digital.type = XL_DAIO_TRIGGER_TYPE_BOTH;
xlStatus = xlIoSetTriggerMode(portHandle, mask, &xlDaioTmDig);
```

11.4.2 XLdaioSetPort

Syntax

```
struct xl_daio_set_port{
  unsigned int portType;
  unsigned int portMask;
  unsigned int portFunction[8];
  unsigned int reserved[8];
} XLdaioSetPort;
```

Parameters

- **portType**
  XL_DAIO_PORT_TYPE_MASK_ANALOG
  XL_DAIO_PORT_TYPE_MASK_DIGITAL

- **portMask**
  Specifies the digital port (D0…D07):
  `XL_DAIO_PORT_MASK_DIGITAL_D0`
  `XL_DAIO_PORT_MASK_DIGITAL_D1`
  `XL_DAIO_PORT_MASK_DIGITAL_D2`
  `XL_DAIO_PORT_MASK_DIGITAL_D3`
  `XL_DAIO_PORT_MASK_DIGITAL_D4`
  `XL_DAIO_PORT_MASK_DIGITAL_D5`
  `XL_DAIO_PORT_MASK_DIGITAL_D6`
  `XL_DAIO_PORT_MASK_DIGITAL_D7`

  Specifies the analog port (A0…A3):
  `XL_DAIO_PORT_MASK_ANALOG_A0`
  `XL_DAIO_PORT_MASK_ANALOG_A1`
  `XL_DAIO_PORT_MASK_ANALOG_A2`
  `XL_DAIO_PORT_MASK_ANALOG_A3`

- **portFunction**
  For digital ports:
  `XL_DAIO_PORT_DIGITAL_OPENDRAIN`
  `XL_DAIO_PORT_DIGITAL_PUSHPULL`
  `XL_DAIO_PORT_DIGITAL_IN`
  `XL_DAIO_PORT_DIGITAL_SWITCH`

  For analog ports:
  `XL_DAIO_PORT_ANALOG_IN`
  `XL_DAIO_PORT_ANALOG_OUT`
  `XL_DAIO_PORT_ANALOG_DIFF`
  `XL_DAIO_PORT_ANALOG_OFF`

  `XL_DAIO_PORT_ANALOG_IN` and `XL_DAIO_PORT_ANALOG_OUT` can be defined at the same time.

- **reserved**
  Set to 0.

Note

- The DoIP Activation Lines of VN5640, VN5650 and VN7640 are assigned to `XL_DAIO_PORT_MASK_DIGITAL_D3` and `XL_DAIO_PORT_MASK_DIGITAL_D4`.
- The DoIP Activation Line of VN5610A, VN5620 and VN5620A is assigned to `XL_DAIO_PORT_MASK_DIGITAL_D3`.



Example

```
XLstatus      xlStatus;
XLportHandle portHandle = ...;
XLaccess mask = ...;
XLdaioSetPort confDaioPortsDig;

memset(&confDaioPortsDig, 0x00, sizeof(confDaioPortsDig));
confDaioPortsDig.portType = XL_DAIO_PORT_TYPE_MASK_DIGITAL;
confDaioPortsDig.portMask = (XL_DAIO_PORT_MASK_DIGITAL_D0 |
                             XL_DAIO_PORT_MASK_DIGITAL_D1 |
                             XL_DAIO_PORT_MASK_DIGITAL_D2 |
                             XL_DAIO_PORT_MASK_DIGITAL_D3 |
                             XL_DAIO_PORT_MASK_DIGITAL_D4 |
                             XL_DAIO_PORT_MASK_DIGITAL_D5 |
                             XL_DAIO_PORT_MASK_DIGITAL_D6 |
                             XL_DAIO_PORT_MASK_DIGITAL_D7);

confDaioPortsDig.portFunction[0] = XL_DAIO_PORT_DIGITAL_PUSHPULL;
confDaioPortsDig.portFunction[1] = XL_DAIO_PORT_DIGITAL_PUSHPULL;
confDaioPortsDig.portFunction[2] = XL_DAIO_PORT_DIGITAL_
OPENDRAIN;
confDaioPortsDig.portFunction[3] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[4] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[5] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[6] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[7] = XL_DAIO_PORT_DIGITAL_IN;

xlStatus = xlIoConfigurePorts(portHandle, mask, &con-
fDaioPortsDig);
```

11.4.3 XLdaioDigitalParams (IOpiggy)

Syntax

```
typedef struct xl_daio_digital_params{
  unsigned int portMask;
  unsigned int valueMask;
} XLdaioDigitalParams;
```

Parameters

- **portMask**
  Specifies the digital port (D0…D07):
  XL_DAIO_PORT_MASK_DIGITAL_D0
  XL_DAIO_PORT_MASK_DIGITAL_D1
  XL_DAIO_PORT_MASK_DIGITAL_D2
  XL_DAIO_PORT_MASK_DIGITAL_D3
  XL_DAIO_PORT_MASK_DIGITAL_D4
  XL_DAIO_PORT_MASK_DIGITAL_D5
  XL_DAIO_PORT_MASK_DIGITAL_D6
  XL_DAIO_PORT_MASK_DIGITAL_D7
- **valueMask**
  Specifies the port value:
  ON/HIGH: 1
  OFF/LOW: 0

11.4.4 XLdaioAnalogParams

Syntax

```
struct xl_daio_analog_params{
  unsigned int portMask;
  unsigned int value[8];
```

```
} XLdaioAnalogParams;
```

Parameters

- **portMask**
  Specifies the analog port (A0…A1):
  `XL_DAIO_PORT_MASK_ANALOG_A0`
  `XL_DAIO_PORT_MASK_ANALOG_A1`
- **valueMask**
  Specifies the port value (12 bit).

11.5 Events

11.5.1 XL DAIO Piggy Data

Syntax

```
struct s_xl_daio_piggy_data {
  unsigned int daioEvtTag;
  unsigned int triggerType;

  union {
    XL_IO_DIGITAL_DATA digital;
    XL_IO_ANALOG_DATA analog;
  } data;
};
```

Description

The event is fired as configured via xlIoSetTriggerMode().

- **For VN1630A/VN1640A**
  See section xlIoSetTriggerMode (VN1600) on page 254.
- **IOpiggy**
  xlIoSetTriggerMode (IOpiggy) on page 240.

An additional event will be fired if the value changes at the digital input (if trigger is configured via `XL_DAIO_TRIGGER_TYPE_PORT`).

Parameters

- **daioEvtTag**
  For analog measurements use `XL_DAIO_EVT_ID_ANALOG`.
  Note: only `measuredAnalogData0` is supported.

  For digital measurements use `XL_DAIO_EVT_ID_DIGITAL`.
  Note: the value is stored in `digitalInputData`, both inputs are mapped to bit 0 and bit 1.

  The input ports can be accessed with the following defines:
  `XL_DAIO_PORT_MASK_DIGITAL_D0`
  `XL_DAIO_PORT_MASK_DIGITAL_D1`
  (see example below).
- **triggerType**
  Not used.
- **data**
  See section XL IO Digital Data on page 250and section XL IO Analog Data on page 249.



Example

Checking digital port D0

```
if (ev.daioData.digital.digitalInputData &
    XL_DAIO_PORT_MASK_DIGITAL_D0) {…}
```

11.5.2 XL IO Analog Data

Syntax

```
typedef struct s_xl_io_analog_data {
  unsigned int measuredAnalogData0;
  unsigned int measuredAnalogData1;
  unsigned int measuredAnalogData2;
```

```
  unsigned int measuredAnalogData3;
} XL_IO_ANALOG_DATA;
```

Parameters

- **measuredAnalogData0**
  First analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData1**
  Second analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData2**
  Third analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData3**
  Fourth analog port that is defined as an input.
  This value is 0 for differential input.



Note

The `measuredAnalogData` returned by the IOpiggy is in millivolt, while VN1630(A) and VN1640(A) return the value in samples of their ADC.

This ADC has an input range of 18 V and a resolution of 10 bit, therefore the application must multiply the `measuredAnalogData` returned by VN1630(A) or VN1640(A) by 17.58 mV.

For more information on the ADC including the use of series resistors, refer to the VN1600 Interface Family manual.

11.5.3 XL IO Digital Data

Syntax

```
typedef struct s_xl_io_digital_data {
  unsigned int digitalInputData;
} XL_IO_DIGITAL_DATA;
```

Parameters

- **digitalInputData**
  Contains the data of port 0 ..7. It is independent of the port function.

12 D/A IO Commands (Other)

In this chapter you find the following information:

12.1 Introduction **252**
12.2 Flowchart **253**
12.3 Functions **254**
12.4 Structs **255**
12.5 Events **256**

12.1 Introduction

Description

The XL Driver Library enables the development of DAIO applications for the VN1600 interface family.

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel parameters can be changed/configured
- DAIO lines can be set
- DAIO lines can be read

Without init access

- DAIO lines can be read



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

12.2 Flowchart

Calling sequence

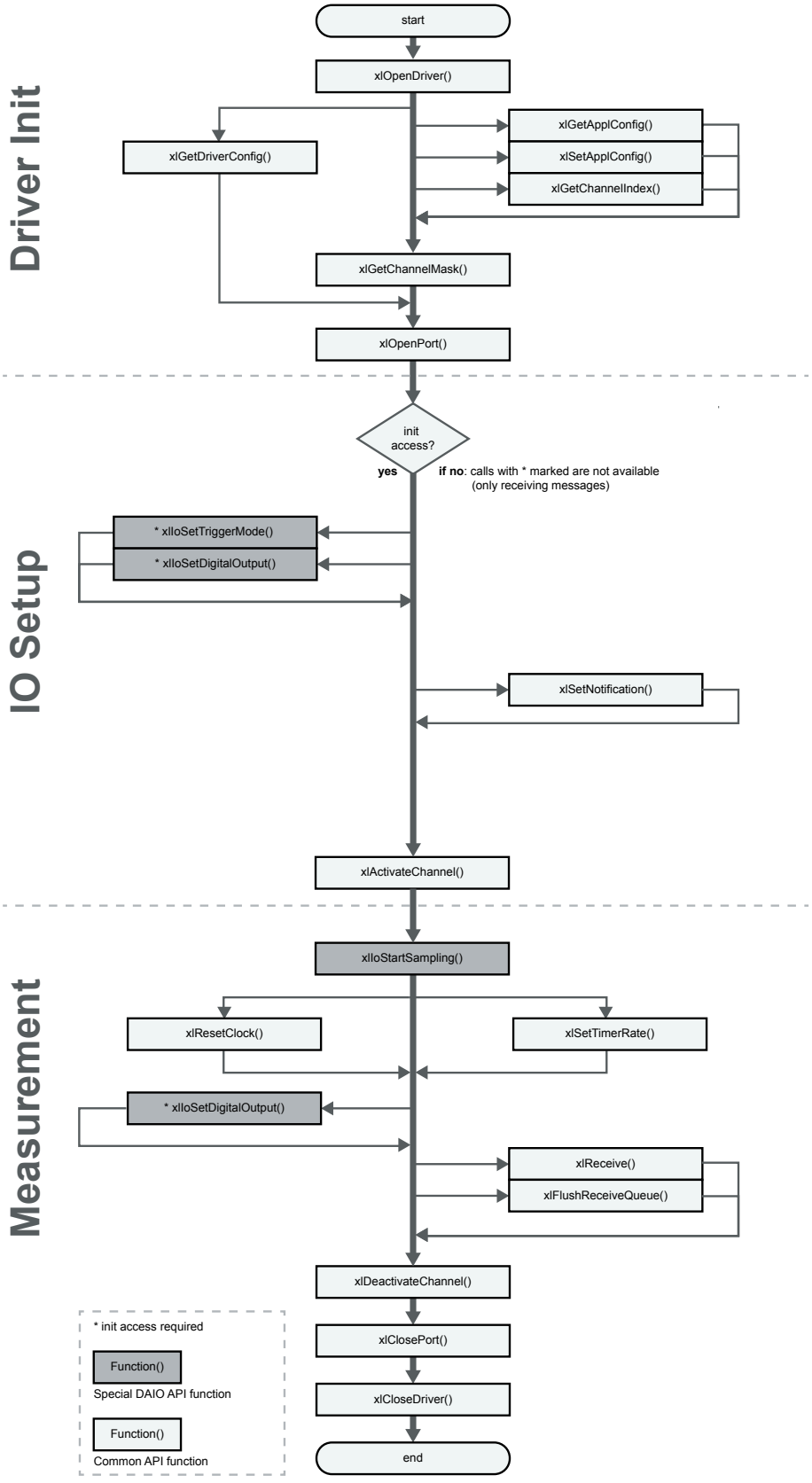


Figure 45: Function calls for DAIO (VN1600) applications

12.3 Functions

12.3.1 xlIoSetTriggerMode (VN1600)

Syntax

```
XLstatus xlIoSetTriggerMode (
  XLportHandle       portHandle,
  XLaccess           accessMask,
  XLdaioTriggerMode* pxlDaioTriggerMode)
```

Description

Sets the DAIO trigger mode for the analog and digital ports. A port group must not have more than one trigger source.



Note
This command can be called only once before xlActivateChannel().

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlDaioTriggerMode**
  Use this structure to define the trigger type (see section XLdaioTriggerMode on page 244).
  Note: Currently only `XL_DAIO_TRIGGER_TYPE_CYCLIC` is supported.

Return value

Returns an error code (see section Error Codes on page 360).

12.3.2 xlIoSetDigitalOutput

Syntax

```
XLstatus xlIoSetDigitalOutput (
  XLportHandle         portHandle,
  XLaccess             accessMask,
  XLdaioDigitalParams* pxlDaioDigitalParams)
```

Description

Configures the digital output.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlDaioDigitalParams**
  Use this structure to set the value of the digital out pin (see section XLdaioDigitalParams on page 255).

Return value

Returns an error code (see section Error Codes on page 360).

12.4 Structs

12.4.1 XLdaioDigitalParams

Syntax

```
typedef struct xl_daio_digital_params{
  unsigned int portMask;
  unsigned int valueMask;
} XLdaioDigitalParams;
```

Parameters

- **portMask**
  Use `XL_DAIO_PORT_MASK_DIGITAL_D0` etc. Please refer to the interface manual for the number of digital ports available on your device.
- **valueMask**
  Specifies the port value:
  ON/HIGH: 1
  OFF/LOW: 0

12.5 Events

12.5.1 XL DAIO Piggy Data

Syntax

```
struct s_xl_daio_piggy_data {
  unsigned int daioEvtTag;
  unsigned int triggerType;

  union {
    XL_IO_DIGITAL_DATA digital;
    XL_IO_ANALOG_DATA analog;
  } data;
};
```

Description

The event is fired as configured via xlIoSetTriggerMode().

- **For VN1630A/VN1640A**
  See section xlIoSetTriggerMode (VN1600) on page 254.
- **IOpiggy**
  xlIoSetTriggerMode (IOpiggy) on page 240.

An additional event will be fired if the value changes at the digital input.

Parameters

- **daioEvtTag**
  For analog measurements use `XL_DAIO_EVT_ID_ANALOG`.
  Note: only `measuredAnalogData0` is supported.

  For digital measurements use `XL_DAIO_EVT_ID_DIGITAL`.
  Note: the value is stored in `digitalInputData`, both inputs are mapped to bit 0 and bit 1.

  The input ports can be accessed with the following defines:
  `XL_DAIO_PORT_MASK_DIGITAL_D0`
  `XL_DAIO_PORT_MASK_DIGITAL_D1`
  (see example below).
- **triggerType**
  Not used.
- **data**
  section XL IO Digital Data on page 257 and section XL IO Analog Data on page 256.



Example

Checking digital port D0

```
if (ev.daioData.digital.digitalInputData &
    XL_DAIO_PORT_MASK_DIGITAL_D0) {…}
```

12.5.2 XL IO Analog Data

Syntax

```
typedef struct s_xl_io_analog_data {
  unsigned int measuredAnalogData0;
  unsigned int measuredAnalogData1;
  unsigned int measuredAnalogData2;
  unsigned int measuredAnalogData3;
```

```
} XL_IO_ANALOG_DATA;
```

Parameters

- **measuredAnalogData0**
  First analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData1**
  Second analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData2**
  Third analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData3**
  Fourth analog port that is defined as an input.
  This value is 0 for differential input.



Note
The `measuredAnalogData` returned by the IOpiggy is in millivolt, while VN1630(A) and VN1640(A) return the value in samples of their ADC.

This ADC has an input range of 18 V and a resolution of 10 bit, therefore the application must multiply the `measuredAnalogData` returned by VN1630(A) or VN1640(A) by 17.58 mV.

For more information on the ADC including the use of series resistors, refer to the VN1600 Interface Family manual.

12.5.3 XL IO Digital Data

Syntax

```
typedef struct s_xl_io_digital_data {
  unsigned int digitalInputData;
} XL_IO_DIGITAL_DATA;
```

Parameters

- **digitalInputData**
  Contains the data of port 0 ..7. It is independent of the port function.

13 FlexRay Commands

In this chapter you find the following information:

13.1 Introduction **259**
13.2 Flowchart **260**
13.3 Free Library and Advanced Library **261**
13.4 FlexRay Basics **262**
13.5 Functions **268**
13.6 Structs **275**
13.7 Events **283**
13.8 Application Examples **297**

13.1 Introduction

Description

The **XL Driver Library** enables the development of FlexRay applications for supported Vector devices (see section System Requirements on page 25).

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- channel configuration can be initialized/modified
- channel can be deactivated/shut down
- FlexRay frames can be transmitted on the channel
- FlexRay frames can be received on the channel

Without init access

- FlexRay frames can be received on the channel
- notification events (initiated by the application with **init access**) can be received (XL_APPLICATION_NOTIFICATION_EV), e. g. activating-/deactivating the channel or closing the port.

Spy mode

In general, if the FlexRay channel is configured for asynchronous mode (spy mode), no FlexRay frame transmission is possible.



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

13.2 Flowchart

Calling sequence

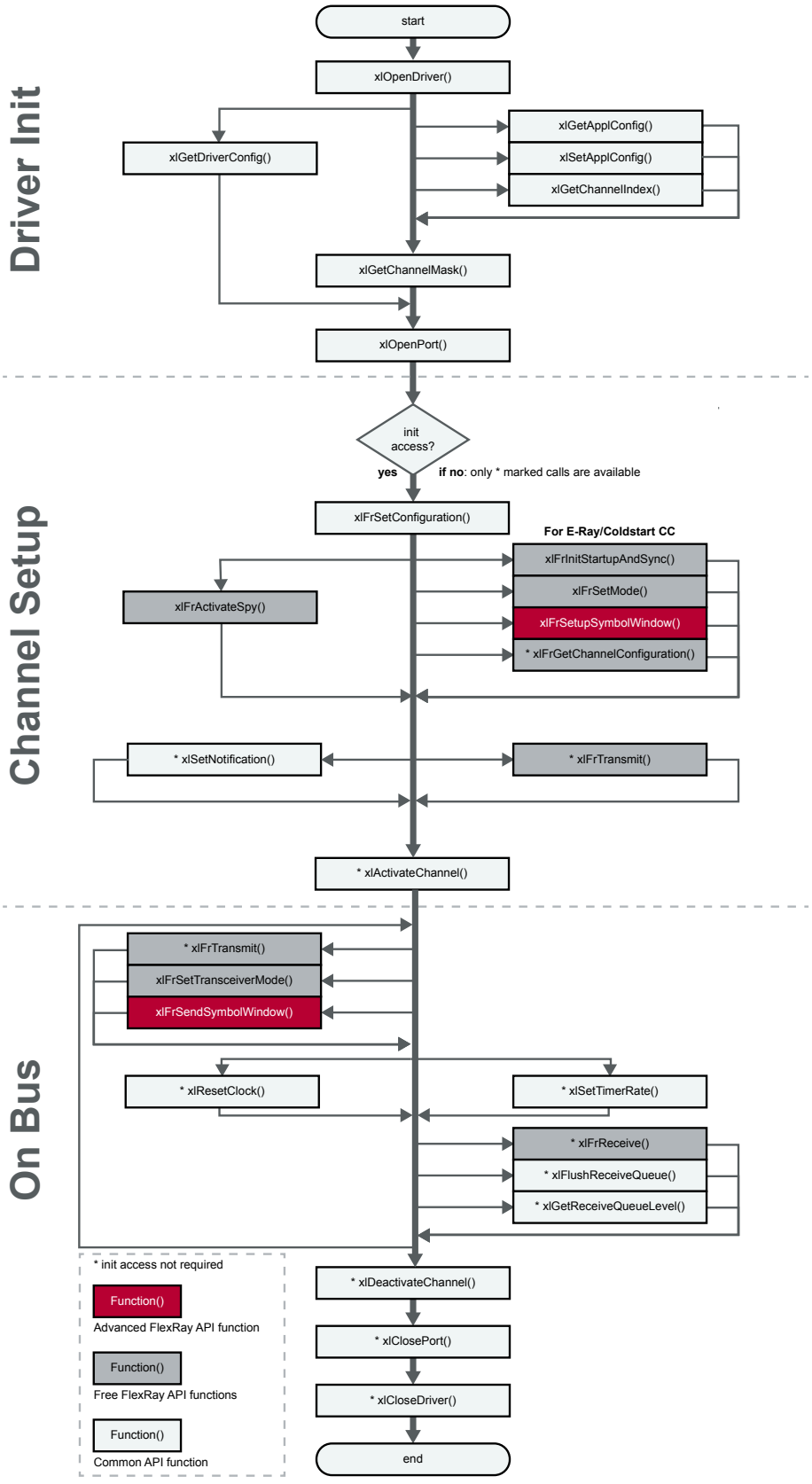


Figure 46: Function calls for FlexRay applications

13.3 Free Library and Advanced Library

Differences

The **XL Driver Library** for FlexRay is split into a free and an advanced version. The differences are as follows:

Init commands

Command	Free	Advanced
xlFrSetConfiguration	X	X
xlFrSetMode	Limited (only E-Ray can be used)	X
xlFrInitStartupAndSync	Limited (only E-Ray can be used)	X
xlFrSetupSymbolWindow	X	X
xlSetTimerBasedNotify	X	X
xlFrActivateSpy	X	X

Messages

Command	Free	Advanced
xlFrReceive	X	X
xlFrTransmit	Limited (only 128 different txFrames can be used) no PayloadIncrement	X
xlFrSendSymbolWindow	-	X

General commands

Command	Free	Advanced
xlFrSetTranseiverMode	X	X
xlFrAcceptanceFilter	-	X



Note

The advanced version is unlocked by the Advanced FlexRay Library License or a CANoe/CANalyzer FlexRay license (see section License Management on page 27).

13.4 FlexRay Basics

13.4.1 Introduction

Deterministic and quick data transmission

Implementations of ever more challenging safety and driver-assistance functions go hand in hand with the increasingly more intensive integration of electronic ECUs in the automobile. These implementations require very high data rates to transmit the increasing number of control and status signals. They are signals that not only need to be transmitted extremely quickly; their transmission also needs to be absolutely deterministic.

Fault-tolerant structures required

That is the reason for the growing importance of communication systems that guarantee fast and deterministic data transmission in the automobile. Potential use of by-wire systems further requires the design of fault-tolerant structures and mechanisms. Although by-wire systems may offer wide-ranging capabilities and the benefits of increased design freedom, simplified assembly, personalization of the vehicle, etc., data transmission requirements in the automobile are elevated considerably, because these systems belong to the class of fail-operational systems. They must continue to operate acceptably even when an error occurs.

CAN cannot satisfy these requires due to its event-driven and priority-driven bus ac-cess, its limited bandwidth of 500 KBit/sec based on physical constraints in the auto-mobile, and lack of fault-tolerant structures and mechanisms.

13.4.2 Data Transmission Requirements

Other bus technologies

The certainty that CAN could hardly be expected to satisfy growing data transmission requirements in the automobile over the mid-term, led to the development of a num-ber of deterministic and fault-tolerant serial bus systems with far greater data rates than CAN. Examples include: TTP (Time Triggered Protocol), Byteflight and TTCAN (Time Triggered CAN).

FlexRay communication standard

Based on Byteflight bus technology, the FlexRay Consortium created the cross-OEM, deterministic and fault-tolerant FlexRay communication standard with a data rate of 10 Mbit/sec for extremely safety- and time-critical applications in the automobile.

FlexRay specification

Making a significant contribution to the success of FlexRay was the detailed documentation of the FlexRay specification. The two most important specifications, the communication protocol and the physical layer, are currently in Version 2.1. These and other FlexRay bus technology specifications can be downloaded from the homepage of the FlexRay Consortium.

13.4.3 FlexRay Communication Architecture

FlexRay unlike CAN

Just as in the case of data communication in a CAN cluster, data communication in a FlexRay cluster is also based on a multi-master communication structure. However, the FlexRay nodes are not allowed uncontrolled bus access in response to applica-tion-related events, as is the case in CAN. Rather they must conform to a precisely defined communication cycle that allocates a specific time slot to each FlexRay mes-sage (Time Division Multiple Access - TDMA) and thereby prescribes the send times of all FlexRay messages.

FlexRay communication

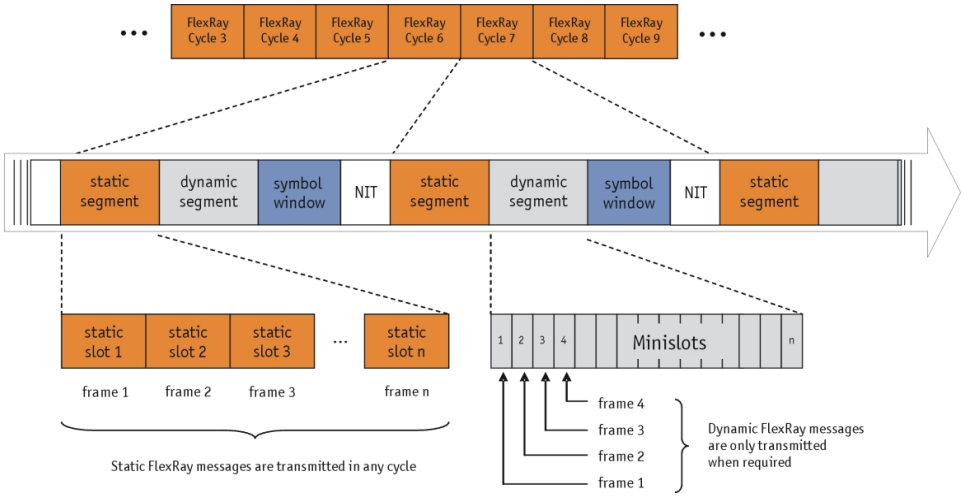


Figure 47: Principle of FlexRay communication

Deterministic data communication

Time-triggered communication not only ensures deterministic data communication; it also ensures that all nodes of a FlexRay cluster can be developed and tested independent of one another. In addition, removal or addition of FlexRay nodes in an existing cluster must not impact the communication process; this is consistent with the goal of re-use that is often pursued in automotive development.

Synchronism of FlexRay nodes

Following the paradigms of time-triggered communication architectures, the under-lying logic of FlexRay communication consists of triggering all system activities when specific points are reached in the time cycle. The network-wide synchronism of FlexRay nodes that is necessary here is assured by a distributed, fault-tolerant clock synchronization mechanism: All FlexRay nodes not only continuously correct for the beginning times (offset correction) of regularly transmitted synchronization messages; they also correct for the duration (slope correction) of the com-munication cycles. This increases both the bandwidth efficiency and robustness of the synchronization.

Star topology

FlexRay communication is not bound by a specific topology. A simple, passive bus structure is just as feasible as an active star topology or a combination of the two. The primary advantages of the active star topology lie in possibility of disconnecting faulty communication branches or FlexRay nodes and - in designing larger clusters - the ability to terminate with ideal bus terminations when physical signal trans-mission is electrical.

FlexRay topology

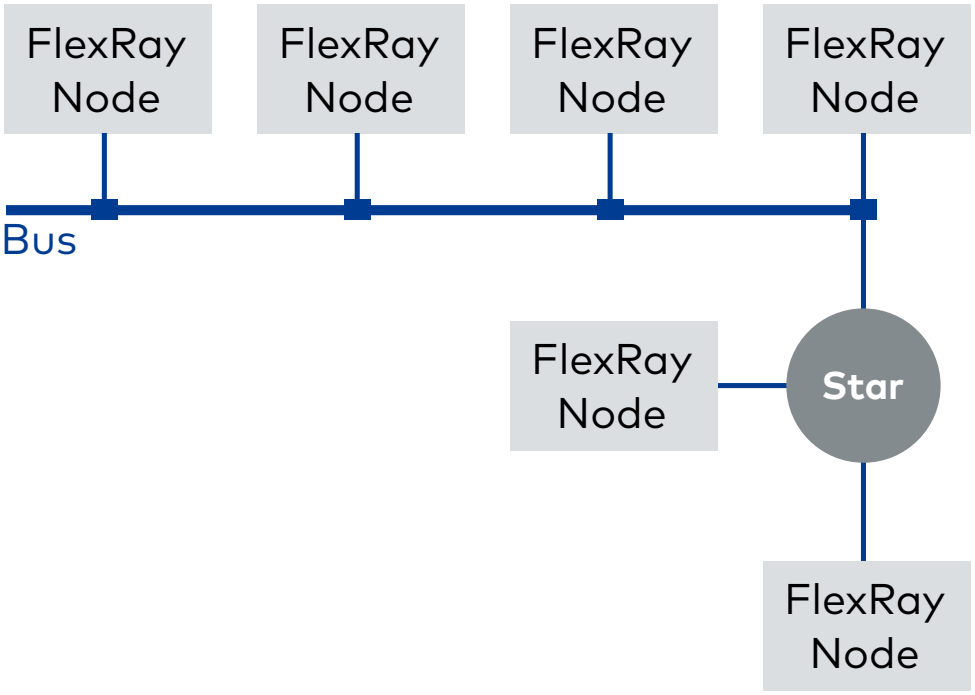


Figure 48: Combined topology of passive bus and active star

Redundant communication

To minimize failure risk, FlexRay offers redundant layout of the communication chan-nel. This redundant communication channel could, on the other hand, be used to increase the data rate to 20 Mbit/sec. The choice between fault tolerance and additional bandwidth can be made individually for each FlexRay message.

FlexRay nodes

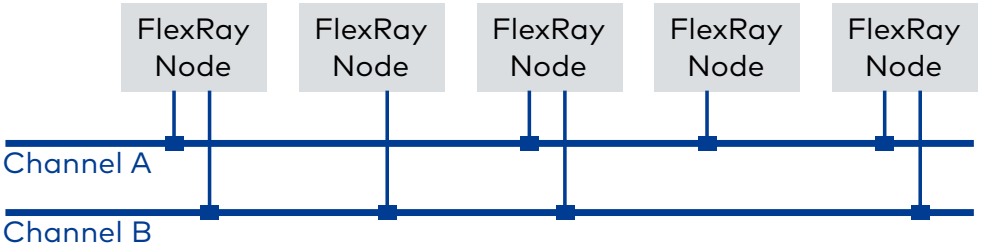


Figure 49: Passive bus structure with two communication channels minimizes failure risk

13.4.4 Deterministic and Dynamic

Each cycle with equal length

Each communication cycle is equal in length and is essentially organized into a static time segment and a dynamic time segment. Of central importance here is the static segment that begins each communication cycle. It is subdivided into a user-definable number (maximum 1023) of equally long static slots.

Static segment

Each static slot is assigned to a FlexRay message to be sent by a FlexRay node. Assignments of static slots, FlexRay messages and FlexRay nodes are made by slot number, message identifier (ID), and the value of the slot counter implemented on each FlexRay node. To ensure that all FlexRay messages are transmitted at the right time and in the correct sequence in each cycle, the slot counters on all FlexRay nodes are incremented synchronously at the beginning of each static slot. Because of its guaranteed equidistant and therefore deterministic data transmission, the static segment is predestined for the transmission of real-time relevant messages.

Dynamic segment

Following the static segment is an optional dynamic segment that has the same length in every communication cycle. This segment is also organized into slots, but not static slots, rather so-called minislots. Communication in the dynamic segment (mini-slotting) is also based on allocations and synchronous incrementing of the slot counters on the FlexRay nodes.

However, it is not mandatory to transmit the FlexRay messages associated to the minislots with each communication cycle, rather they are only sent as needed. If messages are not needed, the slot counter of a minislot is incremented after the defined time period. While a (dynamic) FlexRay message is being transmitted, incrementing of the slot counter is delayed by the message transmission time.

Bus structure with two channels

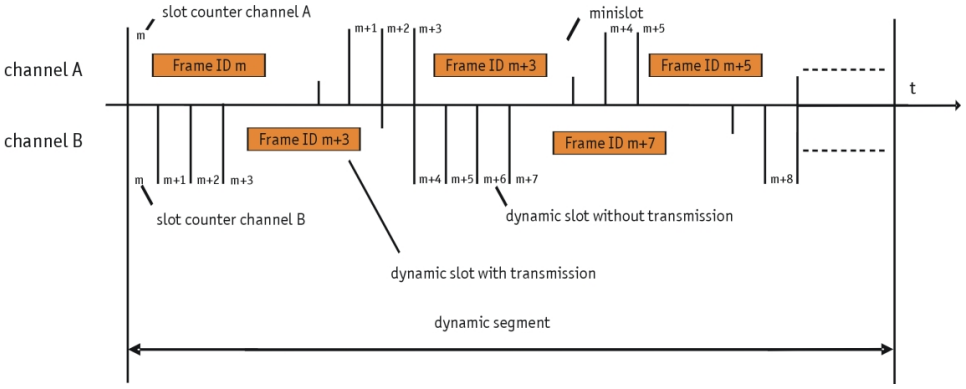


Figure 50: Passive bus structure with two communication channels minimizes failure risk

Priority of dynamic messages

The allocation of a dynamic FlexRay message to a minislot implicitly defines the priority of the FlexRay message: The lower the number of the minislot, the higher the priority of the dynamic FlexRay message, the earlier it will be transmitted, and the higher the probability of transmission given a limited dynamic time segment length. The dynamic FlexRay message assigned to the first minislot is always transmitted as necessary, provided that there is a sufficiently long dynamic time segment.



Note
In the communication design it must be ensured that the lowest priority dynamic FlexRay message can be transmitted too – at least provided that there are no other, higher priority needs. The designer of a FlexRay cluster must also ensure that trans-mission of the longest dynamic FlexRay message is even possible. Otherwise, the communication design would not make any sense.

Communication cycle

The communication cycle is completed by two additional time segments. The “Symbol Window” segment serves to check the functionality of the Bus Guardian, and the “Network Idle Time – NIT” time segment closes the communication cycle. During the NIT the FlexRay nodes calculate the correction factors needed to synchronize their local clocks. At the end of the NIT, an offset correction is made if necessary (the slope correction is always distributed over the entire communication cycle). There is no data transmission during the NIT.

13.4.5 CRC-Protected Data Transmission

Signals

The signals in a FlexRay cluster are transmitted by the well-defined FlexRay message, wherein there is essentially no difference in the formats of the FlexRay messages transmitted in the static segment and those transmitted in the dynamic segment. They are each composed of a header, payload and trailer.

Structure of FlexRay messages

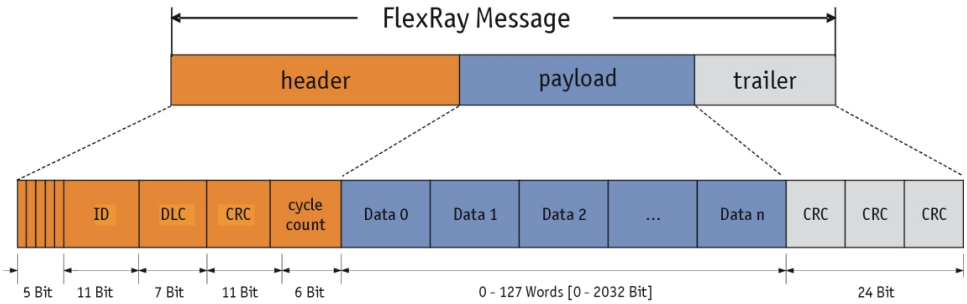


Figure 51: Structure of the FlexRay message with header, payload and trailer

Contents of header

The header comprises the five-bit wide status field, ID, payload length and cycle counter. The header-CRC (11 bits) protects parts of the status field, ID and payload length with a Hamming distance of 6. The ID identifies the FlexRay message and represents a slot in the static or dynamic segment. In the dynamic segment the ID corresponds to the priority of the FlexRay message. The individual bits of the status field specify the FlexRay message more precisely. For example, the “sync frame indicator bit” indicates whether the FlexRay message may be used for clock synchronization.

Payload

After the header the so-called payload follows. A total of up to 254 useful bytes may be transported by one FlexRay message. The trailer encompasses the header and payload-protecting CRC (24 bit). Given a payload of up to 248 useful bytes, the CRC guarantees a Hamming distance of 6. For a larger payload the Hamming distance is 4.

13.5 Functions

13.5.1 xlFrSetConfiguration

Syntax

```
XLstatus xlFrSetConfiguration(
  XLportHandle    portHandle,
  XLaccess        accessMask,
  XLclusterConfig *pxlClusterConfig)
```

Description

Configures the FlexRay CC. The function must be called before xlActivateChannel (). It is not possible to change the FlexRay parameters during runtime. The function requires **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlFrClusterConfig**
  Pointer to the cluster config structure (see section XLfrClusterConfig on page 275).

Return value

Returns an error code (see section Error Codes on page 360).

13.5.2 xlFrGetChannelConfiguration

Syntax

```
XLstatus xlFrGetChannelConfiguration (
  XLportHandle       portHandle,
  XLaccess           accessMask,
  XLfrChannelConfig* pxlFrChannelConfig)
```

Description

Returns the actual cluster configuration depending on the channel.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pxlFrChannelConfig**
  Pointer the config structure (see section XLfrChannelConfig on page 280). Contains the cluster configuration parameters.

Return value

Returns an error code (see section Error Codes on page 360).

13.5.3 xlFrSetMode

Syntax

```
XLstatus xlFrSetMode(
```

```
  XLportHandle portHandle,
  Xlaccess     accessMask,
  XLfrMode     frMode)
```

Description

Sets up the operational mode for both Vector device CCs E-Ray (normal CC) and coldstart (Fujitsu CC). The function must be called before xlActivateChannel() and requires **init access**.

If the function is not called, both CCs are set to default mode `XL_FR_MODE_NORMAL` without wake up for E-Ray. The Fujitsu is completely deactivated.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **frMode**
  Structure of different operational modes (see section XLfrMode on page 280).

Return value

Returns an error code (see section Error Codes on page 360).

13.5.4 xlFrInitStartupAndSync

Syntax

```
XLstatus xlFrInitStartupAndSync(
  XLportHandle  portHandle,
  XLaccess      accessMask,
  XLfrEvent     *pEventBuffer)
```

Description

Initializes the coldstart and defines the sync frame. The function must be called before xlActivateChannel() and requires **init access**. To select the channel and CC, use the `flagsChip` parameter within the basic event structure. To setup different data for FlexRay channels A and B, call it twice. Be sure that the FlexRay config parameters `pKeySlotUsedForSync` and `pKeySlotUsedForStartup` are set! The function requires **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pEventBuffer**
  Pointer to the event buffer which includes the sync frame (see section XLfrEvent on page 283). It is an XL_FR_TX_FRAME event with set `XL_FR_FRAMEFLAG_SYNC/STARTUP` flag.

Return value

Returns an error code (see section Error Codes on page 360).

13.5.5 xlFrSetupSymbolWindow

Syntax

```
XLstatus xlFrSetupSymbolWindow(
  XLportHandle portHandle,
  XLaccess      accessMask,
  unsigned int frChannel,
  unsigned int symbolWindowMask)
```

Description

Sets up the symbol window. The function must be called before xlActivateChannel () and requires **init access**. Defines on which channel the symbol(s) can be sent. At the moment, only a MTS (Media Access Test Symbol) symbol is possible. If the function is called, the config parameter `pChannelMTS` value will be overwritten. The function requires **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **frChannel**
  FlexRay channel A, B or both e. g.:
  `XL_FR_CHANNEL_A`
  `XL_FR_CHANNEL_B`
  `XL_FR_CHANNEL_AB`
- **symbolWindowMask**
  Mask for the symbol windows which can be sent with xlFrSendSymbolWindow (). At the moment, only the MTS is supported (Media Access Symbol):
  `XL_FR_SYMBOL_MTS`

Return value

Returns an error code (see section Error Codes on page 360).

13.5.6 xlFrActivateSpy

Syntax

```
XLstatus xlFrActivateSpy(
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int mode)
```

Description

In asynchronous mode, all FlexRay frames and symbols are received by the spy, but no frame transmission is possible at all. If this mode is selected, only the baudrate has to be passed in the `pxlClusterConfig` parameter of xlFrSetConfiguration(), no further FlexRay configuration data is required.

The function call is optional. If this function is not called, the FlexRay frame reception is done by E-Ray after the Vector device node is integrated in the cluster and the cluster is synchronized.

The function may be called after xlFrSetConfiguration() and requires **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **mode**
  Mode of the Spy:
  `XL_FR_SPY_MODE_ASYNCHRONOUS`

Return value — Returns an error code (see section Error Codes on page 360).

13.5.7 xlSetTimerBasedNotify

Syntax

```
XLstatus xlSetTimerBasedNotify(
  XLportHandle portHandle,
  XLhandle      *pHandle)
```

Description — Sets up an event to notify the application based on the timerrate which can be set by xlSetTimerRate() and xlSetTimerRateAndChannel().



Note
This function is only available for VN3300 und VN760.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

Output parameters

- **phandle**
  Pointer to a WIN32 event handle.

Return value — Returns an error code (see section Error Codes on page 360).

13.5.8 xlFrReceive

Syntax

```
XLstatus xlFrReceive(
  XLportHandle portHandle,
  XLfrEvent    *pEventBuffer)
```

Description — Reads one event from the FlexRay receive queue. Calls to xlFrReceive() can be triggered by a notification event (see section xlSetNotification on page 42). An overrun of the receive queue can be determined by the message flag `XL_FR_QUEUE_OVERFLOW` in `XLfrEvent.flagsChip`.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **pEventBuffer**
  Pointer to an application buffer in which the received event is copied (see section XLfrEvent on page 283).

Return value — Returns an error code (see section Error Codes on page 360).

13.5.9 xlFrTransmit

Syntax

```
XLstatus xlFrTransmit(
  XlportHandle portHandle,
  Xlaccess      accessMask,
  XLfrEvent    *pEventBuffer)
```

Description

The function sends static and dynamic frames with the event tag Tx or can be used for updates in case of cyclic frames. Additionally, a frame payload increment can be configured. To configure different payload increment modes for different `frChannels`, the function has to be called twice (one time for every channel).

This function can be called before and after channel activation.

Basic conflict checking of the frame configuration is also done by this function. If the frame to be sent conflicts with already configured frames (repetition overlapping / cycle overlapping), the frame is not transmitted and the function returns with error. If the frame to be sent is already configured by another application, the frame is not transmitted and the functions returns with error as well.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pEventBuffer**
  Pointer to the event buffer (see section XLfrEvent on page 283).
  Buffersize: `XL_FR_MAX_EVENT_SIZE`

Return value

Returns an error code (see section Error Codes on page 360).

13.5.10 xlFrSetTransceiverMode

Syntax

```
XLstatus xlFrSetTransceiverMode (
  XLportHandle  portHandle,
  XLaccess       accessMask,
  unsigned int  frChannel,
  unsigned int  mode)
```

Description

The function sets up the transceiver modes. For example, to set a FlexRay transceiver into sleep, wake up mode etc. The function requires **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

- **frChannel**
  XL_FR_CHANNEL_A
  XL_FR_CHANNEL_B
  XL_FR_CHANNEL_AB
- **mode**
  Specifies the transceiver mode. e. g.:
  XL_TRANSCEIVER_MODE_SLEEP
  XL_TRANSCEIVER_MODE_NORMAL

Return value — Returns an error code (see section Error Codes on page 360).

13.5.11 xlFrSendSymbolWindow

Syntax

```
XLstatus xlFrSendSymbolWindow(
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int symbolWindow)
```

Description — Sends a symbol window during the next following symbol window as configured by xlFrSetupSymbolWindow(). May be called only after xlActivateChannel() and requires **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **symbolWindow**
  At the moment only:
  XL_FR_SYMBOL_MTS
  Defines the Media Access Symbol.

Return value — Returns an error code (see section Error Codes on page 360).

13.5.12 xlFrSetAcceptanceFilter

Syntax

```
XLstatus xlFrSetAcceptanceFilter(
  XLportHandle         portHandle,
  XLaccess             accessMask,
  XLfrAcceptanceFilter *pAcceptanceFilter)
```

Description — This function modifies the acceptance filter for FlexRay frames. The function requires **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pAcceptanceFilter**
  Pointer to a structure which defines range, channel mask and filter type to be added to the acceptance filter (see section XLfrAcceptanceFilter on page 282).

Return value

Returns an error code (see section Error Codes on page 360).

13.6 Structs

13.6.1 XLfrClusterConfig

Syntax

```
typedef struct s_xl_fr_cluster_configuration {
  unsigned int busGuardianEnable;
  unsigned int baudrate;
  unsigned int busGuardianTick;
  unsigned int externalClockCorrectionMode;
  unsigned int gColdStartAttempts;
  unsigned int gListenNoise;
  unsigned int gMacroPerCycle;
  unsigned int gMaxWithoutClockCorrectionFatal;
  unsigned int gMaxWithoutClockCorrectionPassive;
  unsigned int gNetworkManagementVectorLength;
  unsigned int gNumberOfMinislots;
  unsigned int gNumberOfStaticSlots;
  unsigned int gOffsetCorrectionStart;
  unsigned int gPayloadLengthStatic;
  unsigned int gSyncNodeMax;
  unsigned int gdActionPointOffset;
  unsigned int gdDynamicSlotIdlePhase;
  unsigned int gdMacrotick;
  unsigned int gdMinislot;
  unsigned int gdMiniSlotActionPointOffset;
  unsigned int gdNIT;
  unsigned int gdStaticSlot;
  unsigned int gdSymbolWindow;
  unsigned int gdTSSTransmitter;
  unsigned int gdWakeupSymbolRxIdle;
  unsigned int gdWakeupSymbolRxLow;
  unsigned int gdWakeupSymbolRxWindow;
  unsigned int gdWakeupSymbolTxIdle;
  unsigned int gdWakeupSymbolTxLow;
  unsigned int pAllowHaltDueToClock;
  unsigned int pAllowPassiveToActive;
  unsigned int pChannels;
  unsigned int pClusterDriftDamping;
  unsigned int pDecodingCorrection;
  unsigned int pDelayCompensationA;
  unsigned int pDelayCompensationB;
  unsigned int pExternOffsetCorrection;
  unsigned int pExternRateCorrection;
  unsigned int pKeySlotUsedForStartup;
  unsigned int pKeySlotUsedForSync;
  unsigned int pLatestTx;
  unsigned int pMacroInitialOffsetA;
  unsigned int pMacroInitialOffsetB;
  unsigned int pMaxPayloadLengthDynamic
  unsigned int pMicroInitialOffsetA;
  unsigned int pMicroInitialOffsetB;
  unsigned int pMicroPerCycle;
  unsigned int pMicroPerMacroNom;
  unsigned int pOffsetCorrectionOut;
  unsigned int pRateCorrectionOut;
  unsigned int pSamplesPerMicrotick;
  unsigned int pSingleSlotEnabled;
  unsigned int pWakeupChannel;
  unsigned int pWakeupPattern;
  unsigned int pdAcceptedStartupRange;
  unsigned int pdListenTimeout;
  unsigned int pdMaxDrift;
  unsigned int pdMicrotick;
  unsigned int gdCASRxLowMax;
```

```
  unsigned int gChannels;
  unsigned int vExternOffsetControl;
  unsigned int vExternRateControl;
  unsigned int pChannelsMTS;
  unsigned int reserved[16];
} XLfrClusterConfig;
```

Parameters

- **busGuardianEnable**
  For future use. Has to be set to 0.
- **baudrate**
  FlexRay baudrate. Supported values are:
  10 Mbit: 10.000
  5 Mbit: 5.000
  2,5 Mbit: 2.500
- **busGuardianTick**
  For future use. Has to be set to 0.
- **externalClockCorrectionMode**
  Not used. Has to be set to 0.
- **gColdStartAttempts**
  Maximum number of times a node in the cluster is permitted to attempt to start the cluster by initiating schedule synchronization.
  Range: 2..31
- **gListenNoise**
  Upper limit for the start up listen timeout and wake up listen timeout in the presence of noise.
  Range: 2..16
- **gMacroPerCycle**
  Number of macroticks in a communication cycle.
  Range: 10..16000.
- **gMaxWithoutClockCorrectionFatal**
  Range 1..15.
- **gMaxWithoutClockCorrectionPassive**
  Range: 1..15.
- **gNetworkManagementVectorLength**
  Length of the NM vector.
  Range: 0..12.
- **gNumberOfMinislots**
  Number of mini slots in the dynamic segment.
  Range: 0..7986.
- **gNumberOfStaticSlots**
  Number of static slots in the static segment.
  Range: 2..1023.
- **gOffsetCorrectionStart**
  Start of the offset correction phase within the NIT, expressed as the number of macro ticks from the start of cycle.
- **gPayloadLengthStatic**
  Payload length of a static frame.
  Range: 0..127.

- **gSyncNodeMax**
  Maximum number of nodes that may send frames with the sync frame indicator bit set to one.
  Range: 2..15.
- **gdActionPointOffset**
  Offset of a statical slot from slot beginning to actual StartOfFrame. In macro ticks.
  Range: 2..63.
- **gdDynamicSlotIdlePhase**
  Duration of the idle phase within a dynamic slot.
  Range: 0..2.
- **gdMacrotick**
  No used (calculated internally).
- **gdMinislot**
  Duration of a minislot.
  Range: 2..63.
- **gdMiniSlotActionPointOffset**
  Range: 1..31.
- **gdNIT**
  Duration of the Network Idle Time.
- **gdStaticSlot**
  Duration of a static slot.
  Range: 4..659 macro ticks.
- **gdSymbolWindow**
  Duration of the symbol window. Not used. Has to be set to 0.
- **gdTSSTransmitter**
  Number of bits in the Transmission Start Sequence.
  Range: 3..15
- **gdWakeupSymbolRxIdle**
  Number of bits used by the node to test the duration of the idle portion of a received wake up symbol.
  Range: 14..59.
- **gdWakeupSymbolRxLow**
  Number of bits used by the node to test the LOW portion of a received wake up symbol.
  Range: 10..55.
- **gdWakeupSymbolRxWindow**
  Range: 76..301.
- **gdWakeupSymbolTxIdle**
  Maximum dynamic mini slots.
  Range: 45..180.
- **gdWakeupSymbolTxLow**
  Number of bits used by the node to transmit the idle part of a wake up symbol.
  Range: 15..60.
- **pAllowHaltDueToClock**
  Boolean flag that controls the transition
  - 0: Disable clock halt
  - 1: Enable clock halt

- **pAllowPassiveToActive**
  Number of consecutive even/odd cycle pairs that must have valid clock correction terms.
- **pChannels**
  Channels to which the node is connected.
- **pClusterDriftDamping**
  Local cluster drift damping factor used for rate correction.
  Range: 0..20;
- **pDecodingCorrection**
  Value used by the receiver to calculate the difference between primary time reference point and secondary time reference point.
  Range: 14..143
- **pDelayCompensationA**
- Value used to compensate for reception delays for channel A.
  Range: 0..200
- **pDelayCompensationB**
  Value used to compensate for reception delays for channel B.
  Range: 0..200
- **pExternOffsetCorrection**
  Number of micro ticks added or subtracted to the NIT to carry out a host-requested external offset correction.
  Range: 0..7
- **pExternRateCorrection**
  Number of micro ticks added or subtracted to the cycle to carry out a host-requested external rate correction.
  Range: 0...7
- **pKeySlotUsedForStartup**
  Flag indicating whether the Key Slot is used to transmit a startup frame.
  Not used. Has to be set to 0.
- **pKeySlotUsedForSync**
  Flag indicating whether the Key Slot is used to transmit a sync frame.
  Not used. Has to be set to 0.
- **pLatestTx**
  Number of the last mini slot in which a frame transmission can start in the dynamic segment.
  Range: 0..7981.
- **pMicroInitialOffsetA**
  Number of micro ticks between the closest macrotick boundary on channel A.
  Range: 0..240
- **pMicroInitialOffsetB**
  Number of micro ticks between the closest macrotick boundary on channel B.
  Range: 0..240
- **pMaxPayloadLengthDynamic**
  Not used. Has to be set to 0.

- **pMacroInitialOffsetA**
  Integer number of macro ticks for channel A between the static slot boundary and the following macro tick boundary of the secondary time reference point based on the nominal macro tick duration.
  Range: 0..72.
- **pMacroInitialOffsetB**
  Integer number of macro ticks for channel B between the static slot boundary and the following macro tick boundary of the secondary time reference point based on the nominal macro tick duration.
  Range: 0..72.
- **pMicroPerCycle**
  Nominal number of micro ticks in the communication cycle of the local node. If nodes have different micro tick durations this number will differ from node to node.
  Range: 640..640000.
- **pMicroPerMacroNom**
  Number of micro ticks per nominal macro tick that all implementations must sup-port. Not used. Has to be set to 0.
- **pOffsetCorrectionOut**
  Magnitude of the maximum permissible offset correction value.
  Range: 5...15266.
- **pRateCorrectionOut**
  Magnitude of the maximum permissible rate correction value.
  Range: 2...1923.
- **pSamplesPerMicrotick**
  Number of samples per micro tick. Not used. Has to be set to 0.
- **pSingleSlotEnabled**
  Flag indicating whether or not the node shall enter single slot mode following startup. Not used. Has to be set to 0.
- **pWakeupChannel**
  Channel (A or B) used by the node to send a wake up pattern.
  `XL_FR_CHANNEL_A`
  `XL_FR_CHANNEL_B`
- **gdWakeupPattern**
  Indicates how many times the wake up symbol (WUS) is repeated to form a wake up pattern (WUP).
  Range: 2...63.
- **pdAcceptedStartupRange**
  Expanded range of measured clock deviation allowed for startup frames during integration.
  Range: 0...1875.
- **pdListenTimeout**
  Upper limit for the start up listen timeout and wake up listen timeout.
  Range: 0x504...0x139703.
- **pdMaxDrift**
  Maximum drift offset between two nodes that operate with unsynchronized clocks over one communication cycle.
  Range: 2...1923.

- **pdMicrotick**
  Duration of a micro tick. Not used. Has to be set to 0.
- **gdCASRxLowMax**
  Upper limit of the CAS acceptance window.
  Range: 67...99.
- **gChannels**
  The channels that are used by the cluster. Not used. Has to be set to 0.
- **vExternOffsetControl**
  Not used. Has to be set to 0.
- **vExternRateControl**
  Not used. Has to be set to 0.
- **pChannelsMTS**
  Setup the channels on which the MTS will be send.

13.6.2 XLfrChannelConfig

Syntax

```
struct s_xl_fr_channel_config {
  unsigned int status;
  unsigned int cfgMode;
  unsigned int reserved[6];
  XLfrClusterConfig xlFrClusterConfig;
} XLfrChannelConfig
```

Parameters

- **status**
  XL_FR_CHANNEL_CFG_STATUS_INIT_APP_PRESENT
  XL_FR_CHANNEL_CFG_STATUS_CHANNEL_ACTIVATED
  XL_FR_CHANNEL_CFG_STATUS_VALID_CLUSTER_CF
  XL_FR_CHANNEL_CFG_STATUS_VALID_CFG_MODE
- **cfgMode**
  XL_FR_CHANNEL_CFG_MODE_SYNCHRONOUS
  XL_FR_CHANNEL_CFG_MODE_COMBINED
  XL_FR_CHANNEL_CFG_MODE_ASYNCHRONOUS
- **reserved**
  Reserved for future use.
- **xlFrClusterConfig**
  The cluster config (see section XLfrClusterConfig on page 275).

13.6.3 XLfrMode

Syntax

```
struct s_xl_fr_set_modes {
  unsigned int   frMode;
  unsigned int   frStartupAttributes;
  unsigned short useSelCycle;
  unsigned short selCycle;
  unsigned int reserved[29];
} XLfrMode
```

Parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().

- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.

- **frMode**
  `XL_FR_MODE_NORMAL`
  Sets up the E-Ray CC into normal operation mode (default mode).

  Only paid version:
  `XL_FR_MODE_COLD_NORMAL`
  Sets up the coldstart CC into normal operation mode.

  `XL_FR_MODE_BUS_HALT`
  Specifies that the FlexRay bus is set to halt.
  See parameters `useSelCycle` and `selCycle` below.

- **frStartupAttributes**
  `XL_FR_MODE_NONE`
  No startup attribute set (default).

  `XL_FR_MODE_WAKEUP`
  Sets up the CC into the wakup mode. After wake up the CC goes into normal mode or does a coldstart.

  `XL_FR_MODE_COLDSTART_LEADING`
  Sets up the CC to do a coldstart leading to initiating the schedule synchronization.

  `XL_FR_MODE_COLDSTART_FOLLOWING`
  Sets up the CC to do a coldstart following and joining other coldstart nodes.

  `XL_FR_MODE_WAKEUP_AND_COLDSTART_LEADING`
  Sends Wakeup and Coldstart path initiating the schedule synchronization.

  `XL_FR_MODE_WAKEUP_AND_COLDSTART_FOLLOWING`
  Sends Wakeup and Coldstart path joining other coldstart nodes.

- **useSelCycle**
  Only valid for **frMode** `XL_FR_MODE_BUS_HALT`.
  `0`: Stops immediately.
  Any other value: Stop is executed within the specified FlexRay cycle (see below).

- **selCycle**
  Only valid for **frMode** `XL_FR_MODE_BUS_HALT`.
  Specifies in which FlexRay cycle the bus is set to halt, i. e. switched off.
  Possible values: `0...63`

- **reserved**
  Reserved for future use. Has to be set to 0.

13.6.4 XLfrAcceptanceFilter

Syntax

```
struct s_xl_fr_acceptance_filter {
  unsigned int filterStatus;
  unsigned int filterTypeMask;
  unsigned int filterFirstSlot;
  unsigned int filterLastSlot;
  unsigned int filterChannelMask;
} XLfrAcceptanceFilter;
```

Parameters

- **filterStatus**
  Defines if the specified frame should be blocked or passed.

  Matching frame passes the filter
  `XL_FR_FILTER_PASS`

  Matching frame is blocked
  `XL_FR_FILTER_BLOCK`

- **filterTypeMask**
  Specifies the frame type that should be filtered.

  Specifies a data frame
  `XL_FR_FILTER_TYPE_DATA`

  Specifies a null frame in a used cycle
  `XL_FR_FILTER_TYPE_NF`

  Specifies a null frame in an unused cycle
  `XL_FR_FILTER_TYPE_FILLUP_NF`

- **filterFirstSlot**
  Beginning of the slot range.

- **filterLastSlot**
  End of the slot range (can be the same as filterFirstSlot).

- **filterChannelMask**
  Specifies the FlexRay channel.
  `XL_FR_FILTER_CHANNEL_A`
  `XL_FR_FILTER_CHANNEL_B`

13.7 Events

13.7.1 XLfrEvent

Syntax

```
struct s_xl_fr_event {
  unsigned int           size;
  XLfrEventTag           tag;
  unsigned shor          channelIndex;
  unsigned int           userHandle;
  unsigned short         flagsChip;
  unsigned short         reserved;
  XLuint64               timeStamp;
  XLuint64               timeStampSync;
  union s_xl_fr_tag_data tagData;
} XLfrEvent;
```

Description

- **size**
  In case of events received via RX queue:
  Overall size of the event (in bytes).
  The maximum size is defined by `XL_FR_MAX_EVENT_SIZE`.

  In case of event used in `xlFrTransmit`:
  Parameter not used.

- **Tag**
  Specifies the FlexRay event type / tag.
  `XL_FR_START_CYCLE`
  `XL_FR_RX_FRAME`
  `XL_FR_TX_FRAME`
  `XL_FR_TXACK_FRAME`
  `XL_FR_INVALID_FRAME`
  `XL_FR_WAKEUP`
  `XL_FR_SYMBOL_WINDOW`
  `XL_FR_ERROR`
  `XL_FR_STATUS`
  `XL_FR_NM_VECTOR`
  `XL_FR_TRANCEIVER_STATUS`
  `XL_FR_SPY_FRAME`
  `XL_FR_SPY_SYMBOL`
  `XL_APPLICATION_NOTIFICATION`

- **channelIndex**
  Channel of the received event.

- **userHandle**
  Internal use.

- **flagsChip**
  The lower 8 bit contain the channel:

  E-Ray channels:
  `XL_FR_CHANNEL_A`
  `XL_FR_CHANNEL_B`
  `XL_FR_CHANNEL_AB`

  SPY channels:
  `XL_FR_SPY_CHANNEL_A`
  `XL_FR_SPY_CHANNEL_B`

  Coldstart (Fujitsu channels) (Tx only within the paid version):
  `XL_FR_CC_COLD_A`
  `XL_FR_CC_COLD_B`

  The upper 8 bit contain special flags:
  `XL_FR_QUEUE_OVERFLOW`

  NOTE: for the `XL_FR_STATUS` event the flags will not be set.
- **reserved**
  Reserved for future use.
- **time stamp**
  Raw time stamp (starting with 0 when device is powered) with 1ns resolution and 8 µs granularity. Resetting the time stamp by xlResetClock() and time synchroni-zation has no effect on this time stamp. Use `timestamp_sync` instead.
- **timestamp_sync**
  Synchronized time stamp with 1 ns resolution and 8 µs granularity. (PC→ device). Time synchronization is applied if enabled in the Vector Hardware Manager. Offset correction is possible with xlResetClock().



Note
The time stamp of the event header is an end of frame time stamp. For spy frames, this time stamp is taken at the frame end sequence (FES), measured at recognition of the recessive bit of FES.

13.7.2 XL_FR_START_CYCLE_EV

Syntax

```
struct s_xl_fr_start_cycle {
  unsigned int cycleCount;
  int          vRateCorrection;
  int          vOffsetCorrection;
  unsigned int vClockCorrectionFailed;
  unsigned int vAllowPassivToActive;
  unsigned int reserved[3];
} XL_FR_START_CYCLE_EV;
```

Parameters

- **cycleCount**
  Current cycle count.
- **vRateCorrection**
  Rate correction in microticks.
- **vOffsetCorrection**
  Offset correction in microticks.

- **vClockCorrectionFailed**
  vAllowPassivToActive.
- **Reserved**
  For future use.

13.7.3 XL_FR_RX_FRAME_EV

Syntax

```
struct s_xl_fr_rx_frame {
  unsigned short flags;
  unsigned short headerCRC;
  unsigned short slotID;
  unsigned char  cycleCount;
  unsigned char  payloadLength;
  unsigned char  data[XL_FR_MAX_DATA_LENGTH];
} XL_FR_RX_FRAME_EV;
```

Parameters

- **flags**

  `XL_FR_FRAMEFLAG_STARTUP`
  Startup flag, set from CC frame buffer.

  `XL_FR_FRAMEFLAG_SYNC`
  Sync bit, set from CC frame buffer.

  `XL_FR_FRAMEFLAG_NULLFRAME`
  If set, the Rx frame is a null frame otherwise it contains a valid FlexRay frame.

  `XL_FR_FRAMEFLAG_PAYLOAD_PREAMBLE`
  Payload preamble bit, set from CC frame buffer.

  `XL_FR_FRAMEFLAG_FR_RESERVED`
  Reserved by the FlexRay protocol (zero in current FlexRay version V2.1)

  `XL_FR_FRAMEFLAG_SYNTAX_ERROR`

  `XL_FR_FRAMEFLAG_CONTENT_ERROR`
  A content error was observed in the assigned slot. (s. FR spec Ch.: 6.2.3)

  `XL_FR_FRAMEFLAG_SLOT_BOUNDARY_VIOLATION`
  A slot boundary violation (channel active at the start or at the end of the assigned slot) was observed.

  `XL_FR_FRAMEFLAG_TX_CONFLICT`
  The transmission conflict indication is set if a transmission conflict has occurred. E. g. if both channels try to send on the same slot (only used for XL_FR_TXACK_FRAME).

  `XL_FR_FRAMEFLAG_FRAME_TRANSMITTED`
  Tx frame has been transmitted. If the flag is not set after a transmission, an error has occurred (only used for XL_FR_TXACK_FRAME).

  `XL_FR_FRAMEFLAG_TXACK_SS`
  Indicates TxAck of SingleShot (only used for XL_FR_TXACK_FRAME).

  `XL_FR_FRAMEFLAG_NEW_DATA_TX`
  Will be set by the CC after the frame has been sent the first time with updated data (only used for XL_FR_TXACK_FRAME).

  `XL_FR_FRAMEFLAG_DATA_UPDATE_LOST`
  Indication that data update has been lost (only used for XL_FR_TXACK_FRAME).

- **headerCRC**
  Frame header CRC.

- **cycleCount**
  Cycle in which the frame has been received.

- **slotID**
  ID from CC receive buffer.

- **payloadLength**
  Payload in words. (0…127 words). One word -> 16bit.

- **data**
  XL_FR_MAX_DATA_LENGTH (here 254).

13.7.4 XL_FR_TX_FRAME_EV

Syntax

```
struct s_xl_fr_tx_frame {
  unsigned short flags;
  unsigned short slotID;
  unsigned char  offset;
  unsigned char  repetition;
  unsigned char  payloadLength;
  unsigned char  txMode;
  unsigned char  incrementSize;
  unsigned char  incrementOffset;
  unsigned char  reserved0;
  unsigned char  reserved1;
  unsigned char  data[XL_FR_MAX_DATA_LENGTH];
} XL_FR_TX_FRAME_EV;
```

Parameters

- **flags**
  XL_FR_FRAMEFLAG_NULLFRAME
  If set, the Tx frame is a null frame, otherwise it contains a valid FlexRay frame.

  XL_FR_FRAMEFLAG_SYNC
  Sync bit, set from CC frame buffer. (Only in coldstart mode).

  XL_FR_FRAMEFLAG_STARTUP
  Startup flag, set from CC frame buffer. (Only in coldstart mode).

  XL_FR_FRAMEFLAG_PAYLOAD_PREAMBLE
  Payload preamble bit, set from CC frame buffer.

  XL_FR_FRAMEFLAG_FR_RESERVED
  Reserved by the FlexRay protocol (zero in current FlexRay version V2.1)

  XL_FR_FRAMEFLAG_REQ_TXACK
  Flag may be set for requesting Tx acknowledge events. (Only used for XL_FR_TX_FRAME).
- **slotID**
  Slot ID of the transmitted frame.
- **offset**
  Offset of the Tx frame.
- **repetition**
  Repetition of the Tx frame.
- **payloadLength**
  Payload in words. (0…127 words). Word -> 16bit

- **txMode**
  `XL_FR_TX_MODE_CYCLIC`
  Sets up the E-Ray to send the frame cyclic.

  `XL_FR_TX_MODE_SINGLE_SHOT`
  The frame will be sent only once. After sending, null frames will be sent.

  `XL_FR_TX_MODE_NONE`
  Turns off the sending of FlexRay frames.
- **incrementSize** (ADVANCED VERSION ONLY)
  If this is unequal to NULL, payload increment is done. The values listed below are used to specify the size of the value to be incremented and start payload increment; the chosen definition has to be set for every data update not in-tended to stop the payload increment. The increment value will be one after a successfully transmission.

  `XL_FR_PAYLOAD_INCREMENT_8BIT`
  `XL_FR_PAYLOAD_INCREMENT_16BIT`
  `XL_FR_PAYLOAD_INCREMENT_32BIT`
- **incrementOffset** (ADVANCED VERSION ONLY)
  Byte offset of the value to be incremented. For an increment size of 8 bit a byte alignment of the value to be incremented is possible, for an increment size of 16 bit the value has to be 16 bit aligned, for an increment size of 32 bit the value has to be 32 bit aligned.
- **reserved0**
  For future extensions – has to be set to “0”.
- **reserved1**
  For future extensions – has to be set to “0”.
- **data**
  `XL_FR_MAX_DATA_LENGTH` (here 254).

13.7.5 XL_FR_TXACK_FRAME

Reference
Same as `XL_FLEXRAY_RX_FRAME`.

13.7.6 XL_FR_INVALID_FRAME

Reference
Same as `XL_FLEXRAY_RX_FRAME`.

13.7.7 XL_FR_WAKEUP_EV

Syntax

```
struct s_xl_fr_wakeup {
  unsigned char cycleCount;
  unsigned char wakeupStatus;
  unsigned char reserved[6];
```

```
} XL_FR_WAKEUP_EV;
```

Parameters

- **cycleCount**
  Current cycle count.
- **wakeupStatus**
  XL_FR_WAKEUP_UNDEFINED
  No wake up attempt since POC-state XL_FR_STATUS_CONFIG was left. On a received wake up pattern on frChannel A|B, this value will be set.

  XL_FR_WAKEUP_RECEIVED_HEADER
  Set when the CC finishes wake up due to the reception of a frame header without coding violation on either channel in WAKEUP_LISTEN state.

  XL_FR_WAKEUP_RECEIVED_WUP
  Set when the CC finishes wake up due to the reception of a valid wake up pattern on the configured wake up channel in WAKEUP_LISTEN state.

  XL_FR_WAKEUP_COLLISION_HEADER
  Set when the CC stops wake up due to a detected collision during wake up pattern transmission by receiving a valid header on either channel.

  XL_FR_WAKEUP_COLLISION_WUP
  Flag is set if the CC stops wake up due to a detected collision or during wake up pattern transmission by receiving a valid wake up pattern on the configured wake up channel.

  XL_FR_WAKEUP_COLLISION_UNKNOWN
  Set when the CC stops wake up by leaving WAKEUP_DETECT state after expira-tion of the wake up timer without receiving a valid wakeup pattern or a valid frame header.

  XL_FR_WAKEUP_TRANSMITTED
  Set when the CC has successfully completed the transmission of the wakeup pattern.

  XL_FR_WAKEUP_RESERVED
- **reserved**
  For future use.

13.7.8 XL_FR_SYMBOL_WINDOW_EV

Syntax

```
struct s_xl_fr_symbol_window {
  unsigned int  symbol;
  unsigned int  flags;
  unsigned char cycleCount;
  unsigned char reserved[7];
} XL_FR_SYMBOL_WINDOW_EV;
```

Parameters

- **symbol**
  XL_FR_SYMBOL_MTS
  Media Access Test Symbol
- **cycleCount**
  Current cycle count.

- **reserved**
  Reserved for future use.
- **flags**
  E-Ray: SWNIT register:
  `XL_FR_SYMBOL_STATUS_SESA`
  Syntax Error in Symbol Window Channel A.

  `XL_FR_SYMBOL_STATUS_SBSA`
  Slot Boundary Violation in Symbol Window Channel A.

  `XL_FR_SYMBOL_STATUS_TCSA`
  Transmission Conflict in Symbol Window Channel A.

  `XL_FR_SYMBOL_STATUS_SESB`
  Syntax Error in Symbol Window Channel B.

  `XL_FR_SYMBOL_STATUS_SBSB`
  Slot Boundary Violation in Symbol Window Channel B.

  `XL_FR_SYMBOL_STATUS_TCSB`
  Transmission Conflict in Symbol Window Channel B.

13.7.9 XL_FR_ERROR_EV

Syntax

```
struct s_xl_fr_error {
  unsigned char            tag;
  unsigned char            cycleCount;
  unsigned char            reserved[6];
  union s_xl_fr_error_info errorInfo;
} XL_FR_ERROR_EV;
```

Parameters

- **tag**
  Error tag for errorInfo:
  `XL_FR_ERROR_POC_MODE`
  `XL_FR_ERROR_SYNC_FRAMES_BELOWMIN`
  `XL_FR_ERROR_SYNC_FRAMES_OVERLOAD`
  `XL_FR_ERROR_CLOCK_CORR_FAILURE`
  `XL_FR_ERROR_NIT_FAILURE`
  `XL_FR_ERROR_CC_ERROR`
- **cycleCount**
  Current cycle count.
- **reserved**
  Reserved for future use.
- **errorInfo**
  Union for further error information.

13.7.10 XL_FR_ERROR_POC_MODE_EV

Syntax

```
struct s_xl_fr_error_poc_mode {
  unsigned char errorMode;
  unsigned char reserved[3];
} XL_FR_ERROR_POC_MODE_EV;
```

Parameters

- **errorMode**
  Indicates the actual error mode of the POC:
  `XL_FR_ERROR_POC_ACTIVE`
  `XL_FR_ERROR_POC_PASSIVE`
  `XL_FR_ERROR_POC_COMM_HALT`
- **reserved**
  For future use.

13.7.11 XL_FR_ERROR_SYNC_FRAMES_BELOWMIN

Description

Not enough sync frames received in cycle.

13.7.12 XL_FR_ERROR_SYNC_FRAMES_EV

Syntax

```
struct s_xl_fr_error_sync_frames {
  unsigned short evenSyncFramesA;
  unsigned short oddSyncFramesA;
  unsigned short evenSyncFramesB;
  unsigned short oddSyncFramesB;
  unsigned int   reserved;
} XL_FR_ERROR_SYNC_FRAMES_EV;
```

Parameters

- **evenSyncFramesA**
  Valid Rx/Tx sync frames on frCh A for even cycles.
- **oddSyncFramesA**
  Valid Rx/Tx sync frames on frCh A for odd cycles.
- **evenSyncFramesB**
  Valid Rx/Tx sync frames on frCh B for even cycles.
- **oddSyncFramesB**
  Valid Rx/Tx sync frames on frCh B for odd cycles.
- **Reserved**
  For future use.

13.7.13 XL_FR_ERROR_CLOCK_CORR_FAILURE_EV

Syntax

```
struct s_xl_fr_error_clock_corr_failure {
  unsigned short evenSyncFramesA;
  unsigned short oddSyncFramesA;
  unsigned short evenSyncFramesB;
  unsigned short oddSyncFramesB;
  unsigned int   flags;
  unsigned int   clockCorrFailedCounter;
  unsigned int   reserved;
} XL_FR_ERROR_CLOCK_CORR_FAILURE_EV;
```

Parameters

- **evenSyncFramesA**
  Valid Rx/Tx sync frames on frCh A for even cycles.
- **oddSyncFramesA**
  Valid Rx/Tx sync frames on frCh A for odd cycles.
- **evenSyncFramesB**
  Valid Rx/Tx sync frames on frCh B for even cycles.

- **oddSyncFramesB**
  Valid Rx/Tx sync frames on frCh B for odd cycles.
- **flags**
  XL_FR_ERROR_MISSING_OFFSET_CORRECTION
  XL_FR_ERROR_MAX_OFFSET_CORRECTION_REACHED
  XL_FR_ERROR_MISSING_RATE_CORRECTION
  XL_FR_ERROR_MAX_RATE_CORRECTION_REACHED
- **clockCorrFailedCounter**
  E-Ray: CCEV register (CCFC value).
- **reserved**
  For future use.

13.7.14 XL_FR_ERROR_NIT_FAILURE_EV

Syntax

```
struct s_xl_fr_error_nit_failure {
  unsigned int flags;
  unsigned int reserved;
} XL_FR_ERROR_NIT_FAILURE_EV;
```

Parameters

- **flags**
  XL_FR_ERROR_NIT_SENA
  Syntax Error during NIT Channel A.

  XL_FR_ERROR_NIT_SBNA
  Slot Boundary Violation during NIT Channel A.

  XL_FR_ERROR_NIT_SENB
  Syntax Error during NIT Channel B.

  XL_FR_ERROR_NIT_SBNB
  Slot Boundary Violation during NIT Channel B.
- **reserved**
  For future use.

13.7.15 XL_FR_ERROR_CC_ERROR_EV

Syntax

```
struct s_xl_fr_error_cc_error {
  unsigned int ccError;
  unsigned int reserved;
} XL_FR_ERROR_CC_ERROR_EV;
```

Parameters

- **ccError**
  E-Ray EIR register:
  `XL_FR_ERROR_CC_PERR`
  The flag signals a parity error to the Host.

  `XL_FR_ERROR_CC_IIBA`
  Illegal Input Buffer Access.

  `XL_FR_ERROR_CC_IOBA`
  Illegal Output buffer Access.

  `XL_FR_ERROR_CC_MHF`
  Message Handler Constraints Flag.

  `XL_FR_ERROR_CC_EDA`
  Error Detected on Channel A.

  `XL_FR_ERROR_CC_LTVA`
  Latest Transmit Violation Channel A.

  `XL_FR_ERROR_CC_TABA`
  Transmission Across Boundary Channel A.

  `XL_FR_ERROR_CC_EDB`
  Error Detected on Channel B.

  `XL_FR_ERROR_CC_LTVB`
  Latest Transmit Violation Channel B.

  `XL_FR_ERROR_CC_TABB`
  Transmission Across Boundary Channel B.

- **reserved**
  For future use

13.7.16 XL_FR_STATUS_EV

Syntax

```
struct s_xl_fr_status {
  unsigned int statusType;
  unsigned int reserved;
} XL_FR_STATUS_EV;
```

Parameters

- **statusType**
  Indicates the actual state of the POC in operation control:
  XL_FR_STATUS_DEFAULT_CONFIG
  XL_FR_STATUS_READY
  XL_FR_STATUS_NORMAL_ACTIVE
  XL_FR_STATUS_NORMAL_PASSIVE
  XL_FR_STATUS_HALT
  XL_FR_STATUS_MONITOR_MODE
  XL_FR_STATUS_CONFIG

  Indicates the actual state of the POC in the wake up path:
  XL_FR_STATUS_WAKEUP_STANDBY
  XL_FR_STATUS_WAKEUP_LISTEN
  XL_FR_STATUS_WAKEUP_SEND
  XL_FR_STATUS_WAKEUP_DETECT

  Indicates the actual state of the POC in the startup path:
  XL_FR_STATUS_STARTUP_PREPARE
  XL_FR_STATUS_COLDSTART_LISTEN
  XL_FR_STATUS_COLDSTART_COLLISION_RESOLUTION
  XL_FR_STATUS_COLDSTART_CONSISTENCY_CHECK
  XL_FR_STATUS_COLDSTART_GAP
  XL_FR_STATUS_COLDSTART_JOIN
  XL_FR_STATUS_INTEGRATION_COLDSTART_CHECK
  XL_FR_STATUS_INTEGRATION_LISTEN
  XL_FR_STATUS_INTEGRATION_CONSISTENCY_CHECK
  XL_FR_STATUS_INITIALIZE_SCHEDULE
  XL_FR_STATUS_ABORT_STARTUP

- **reserved**
  For future use.

13.7.17 XL_FR_NM_VECTOR_EV

Syntax

```
struct s_xl_fr_nm_vector {
  unsigned char nmVector[12];
  unsigned char cycleCount;
  unsigned char reserved[3];
} XL_FR_NM_VECTOR_EV;
```



Note
The NM vector will be sent in combination with the XL_FR_START_CYCLE event on every change.

Parameters

- **cycleCount**
  Current cycle count. Will be set only on cycle changes.
- **nmVector**
  Network management vector.
  The length is depending on `gNetworkManagementVectorLength` (see section XLfrClusterConfig on page 275).

13.7.18 XL_FR_SPY_FRAME_EV

Syntax

```
typedef struct s_xl_fr_spy_frame {
  unsigned int   frameLength;
  unsigned char  frameError;
  unsigned char  tssLength;
  unsigned short headerFlags;
  unsigned short slotId;
  unsigned short headerCRC;
  unsigned char  payloadLength;
  unsigned char  cycleCount;
  unsigned short reserved;
  unsigned int   frameCRC;
  unsigned char  data[254];
} XL_FR_SPY_FRAME_EV;
```

Parameters

- **frameLength**
  Overall length of frame in sample clock ticks.
- **frameError**
  frameError = 0 : valid frame
  frameError != 0 : invalid frame
  `XL_FR_FRAMEFLAG_FRAMING_ERROR`
  `XL_FR_FRAMEFLAG_HEADER_CRC_ERROR`
  `XL_FR_FRAMEFLAG_FRAME_CRC_ERROR`
- **tssLength**
  Length of TSS in bits (transmission start sequence, 3 .. 15 bit)
- **headerFlags**
  `XL_FR_FRAMEFLAG_STARTUP`
  `XL_FR_FRAMEFLAG_SYNC`
  `XL_FR_FRAMEFLAG_NULLFRAME`
  `XL_FR_FRAMEFLAG_PAYLOAD_PREAMBLE`
  `XL_FR_FRAMEFLAG_FR_RESERVED`
  (same flags as for E-Ray RxFrame / TxAckFrame)
- **slotId**
  headerCRC
- **payloadLength**
  Payload length in words. (0…127 words). One word → 16bit.
- **cycleCount**
- **reserved**
  Reserved for future use.
- **frameCRC**
  CRC computed over the header segment and the payload segment of the frame.
- **data**

13.7.19 XL_FR_SPY_SYMBOL_EV

Syntax

```
typedef struct s_xl_fr_spy_symbol {
  unsigned short lowLength;
  unsigned short reserved;
} XL_FR_SPY_SYMBOL_EV;
```

Parameters

- **lowLength**
  Length of low part of symbol (WUS, CAS, MTS) in bits.
- **reserved**
  Reserved for future use.

13.7.20 XL_APPLICATION_NOTIFICATION_EV

Syntax

```
typedef struct s_xl_application_notification {
  unsigned int notifyReason;
  unsigned int reserved[7];
} XL_APPLICATION_NOTIFICATION_EV;
```

Parameters

- **notifyReason**
  XL_NOTIFY_REASON_CHANNEL_ACTIVATION
  XL_NOTIFY_REASON_CHANNEL_DEACTIVATION
  XL_NOTIFY_REASON_PORT_CLOSED
- **reserved**
  Reserved for future use.

13.8 Application Examples

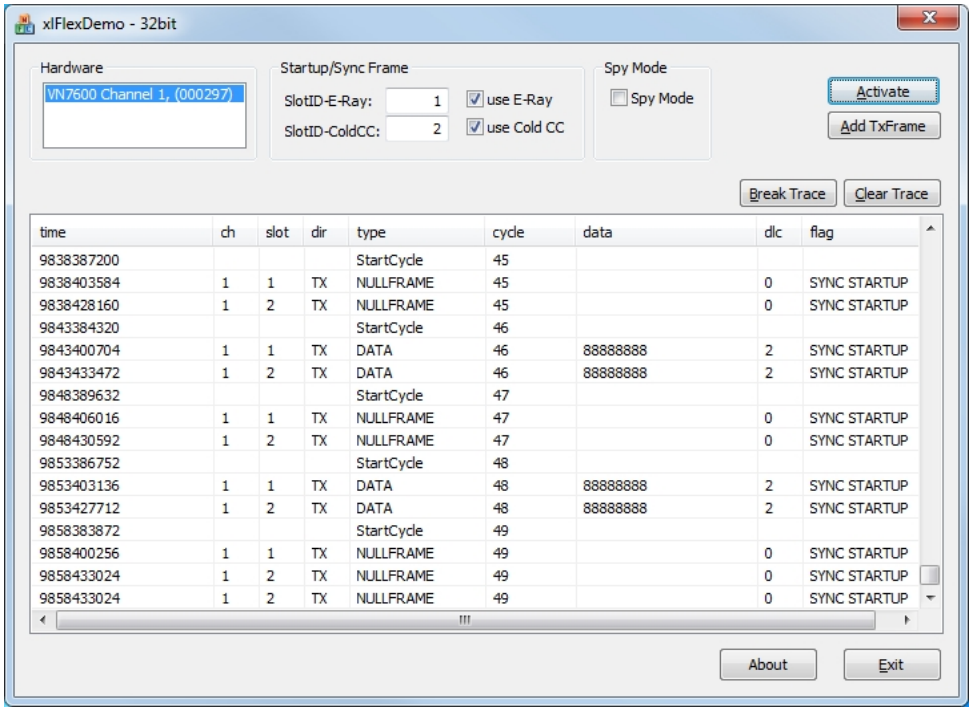
13.8.1 xlFlexDemo

13.8.1.1 General Information

Description

This example demonstrates the basic FlexRay XL API handling. The demo searches for available FlexRay devices on the system and shows them within the **Hardware** listbox. When **[Activate]** is clicked, the amplification tries to start-up the FlexRay bus.

If the FREE library is used, only the E-Ray communication controller will be used. In this case the bus must be started externally. To include a new TxFrame click **[Add TxFrame]**. A dialog box appears to setup the frame parameters (like channel, offset, repetition, slotId…).



13.8.1.2 Classes

Description

The example has the following class structure:

- **CFrFunctions**
  Implementation of all library functions.
- **CFrParseEvent**
  Contains an event parser to display the received events.

13.8.1.3 Functions

Description

- **FrInit**
  Opens the driver and checks the FlexRay channels.
- **FrToggleTrace**
  Switches on/off the “Trace Window”.
- **FrAddTxFrame**
  Calls xlFrTransmit() to add a FlexRay Tx frame.
- **FrActivate**
  Activates the selected FlexRay channel.
- **FrDeactivate**
  Deactivates the active FlexRay channel.

Private

- **frGetChannelMask**
  Gets the device channel masks.
- **frInit**
  Opens the port.
- **frSetConfig**
  Sets up the FlexRay cluster configuration.
- **frStartUpSync**
  Sets up the StartUpAndSync frames depending on the library license.
- **frCreateRxThread**
  Creates the Rx thread to readout the FlexRay message queue.

13.8.1.4 Events

Description

- **parseEvent**
  Filter the events

Private

- **printRxEvent**
  Writes the FlexRay Rx events into the “Trace Window”.
- **printStartOfCycleEvent**
  Writes the FlexRay StartOfCycle events into the “Trace Window”.
- **printValue**
  Writes the values to the tables.
- **printEvent**
  Writes event without any description to the “Trace Window”.

13.8.2 xlFlexDemoCmdLine

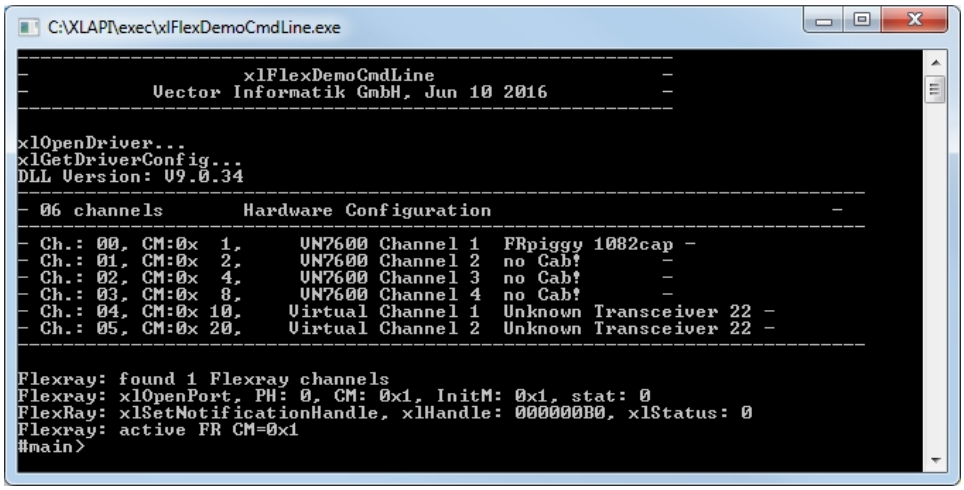
13.8.2.1 General Information

Description

This example demonstrates basic FlexRay XLAPI handling in a console application. Press <h> to show an overview of all available keyboard commands.

For starting up a cluster press <c> to specify a valid Fibex file. If the command succeeded, press <g> to initialize the FlexRay controller and to acti-vate the channels. First, enter a valid slot number for the ERay sync frame, then specify the coldstart-controller (Fujitsu) sync slot number. If all succeeded, the cluster should start up and run. The frames are printed into the window. With the key <v>, the printing can be switched off and on. Press the <Esc> key to exit the application.

Note
In order to compile the example, the Microsoft XML parser package MSXML is required.



13.8.2.2 Functions

Description

- **Main()**
  Main function.
- **RxThread()**
  Independent Rx thread for receiving and processing all events from device.
- **viewFrEvent()**
  Prints all received events in human-readable form.
- **frStartupAndSync()**
  Sets the FlexRay cluster parameters. Initializes and syncs the FlexRay cluster.

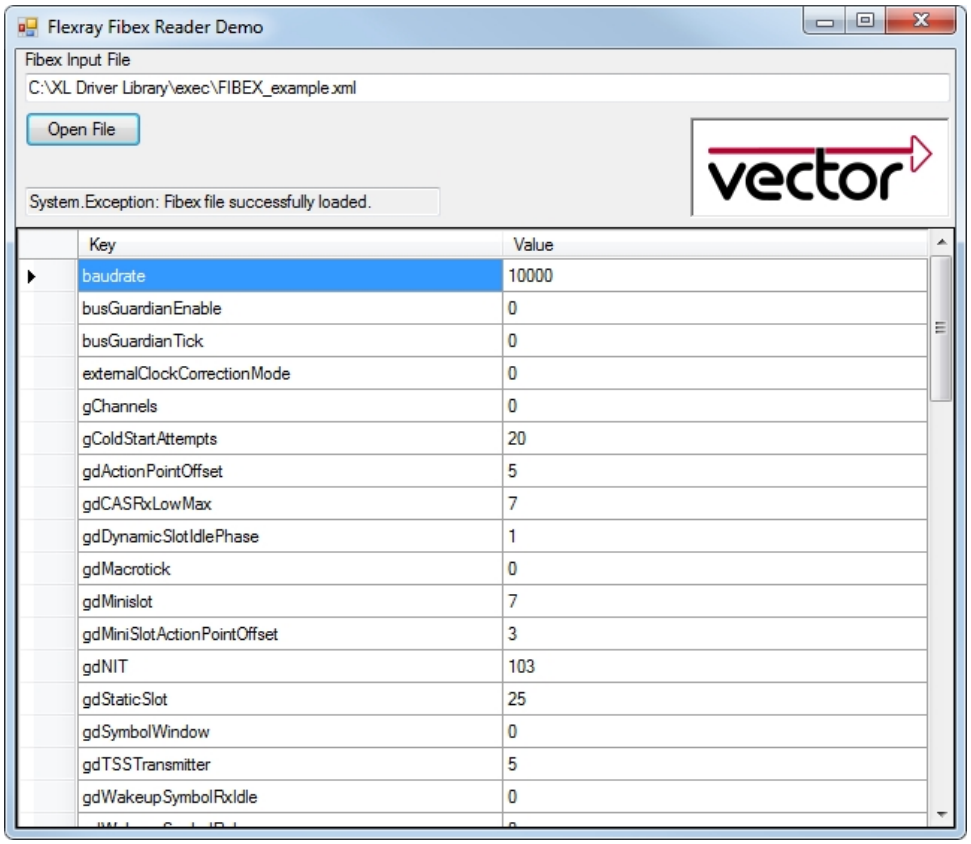
13.8.3 Fibex2CSharpReaderDemo

13.8.3.1 General Information

Description

This example demonstrates the usage of the Fibex Parser example files. In the main form of this application, the input file can be manually specified in the top text field or selected via button **[Open File]**. If the Fibex file is successfully loaded, the content of the Fibex file is shown in the result pane of the main form.

Note
The `FibexParser.cs` implementation supports only Fibex Version 2.0.1 files!



13.8.3.2 Classes

Description

- **Program.cs**
  Contains the code for starting and initializing the application.
- **Form1.cs**
  Contains all code for loading the Form and starting the convertion of the Fibex file.
- **FibexParser.cs**
  Contains the code for parsing Fibex Version 2.0.1 files.

14 ARINC 429 Commands

In this chapter you find the following information:

14.1 Introduction **302**
14.2 Flowchart **303**
14.3 Functions **304**
14.4 Structs **309**
14.5 Events **317**
14.6 Application Examples **325**

14.1 Introduction

Description

The **XL Driver Library** enables the development of ARINC 429 applications for supported Vector devices (see section System Requirements on page 25).

Depending on the channel property **init access** (see page 22), the application's main features are as follows:

With init access

- Common access.

Without init access

- Not supported. If the application gets no **init access** on a specific channel, no further function call is possible on the according channel.



Note
Multi-application is not supported. A single ARINC 429 channel can be used by one application exclusively. The used ARINC 429 channel requires **init access**. A second application gets no **init access** for the assigned channel. In this case `xlOpenPort()` returns an error and no messages can be sent or transmitted.



Reference
See the flowchart on the next page for all available functions and the according calling sequence.

14.2 Flowchart

Calling sequence

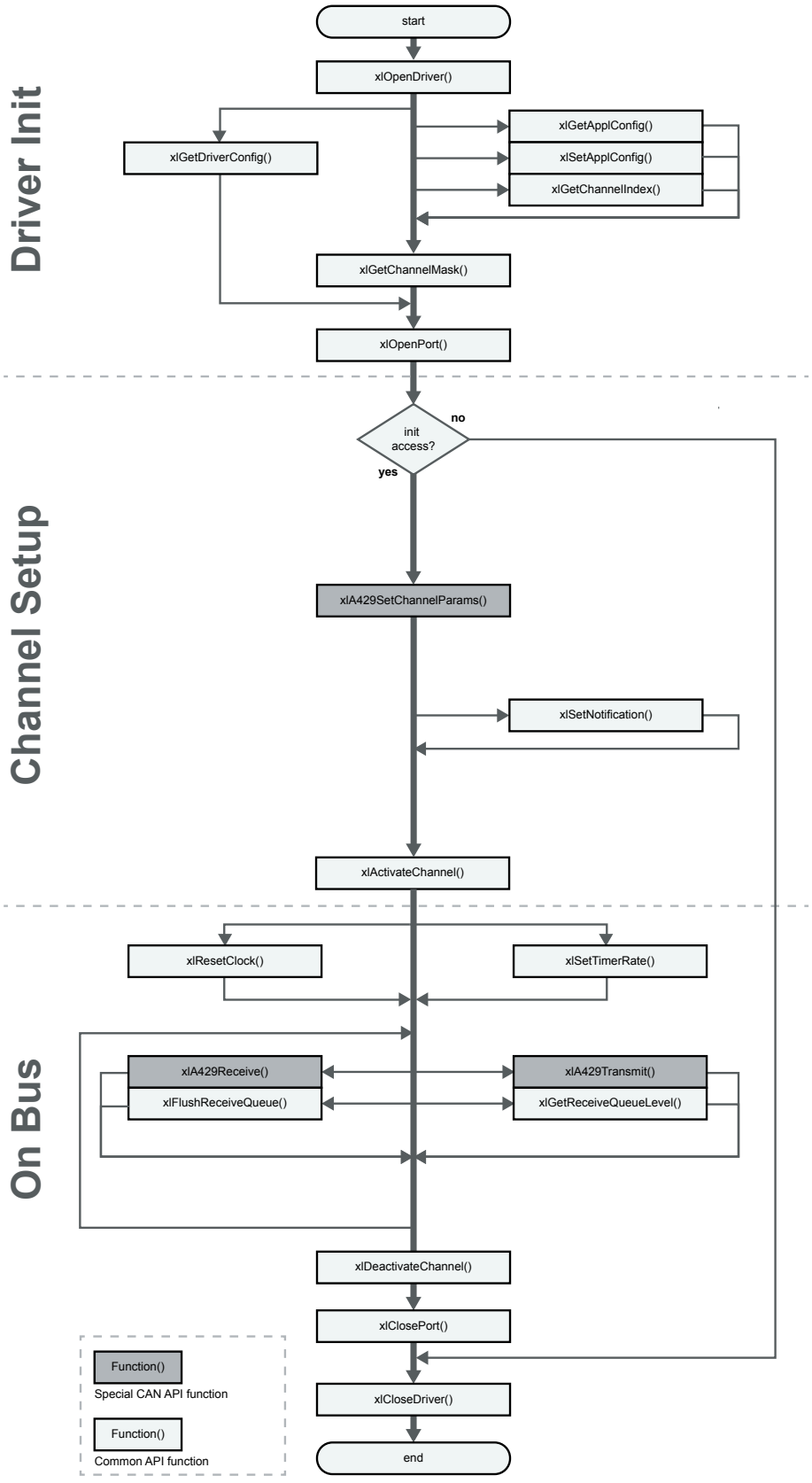


Figure 52: Function calls for ARINC 429 applications

14.3 Functions

14.3.1 xlA429SetChannelParams

Syntax

```
XLstatus xlA429SetChannelParams (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  XL_A429_PARAMS* pXlA429Params
)
```

Description

Configures basic ARINC 429 parameters. Note that the device does not keep those settings after a restart. This is a synchronous operation and function needs **init access**.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **pXlA429Params**
  ARINC 429 configuration structure (see section XL_A429_PARAMS on page 309).



Note

Each `xlA429SetChannelParams()` call has to be called before xlActivateChannel() function call. Parameter changes after xlActivateChannel() calls (e. g. bitrate) are not supported. After xlDeactivateChannel(), `xlA429SetChannelParams()` can be called again.



Example

Configures an A429 channel in Tx channel direction with a bitrate of 100000 bit/s, parity calculation disabled and a default `minGap` of 4 bit.

```
XL_A429_PARAMS xlA429Params;

memset(&xlA429Params, 0, sizeof(XL_A429_PARAMS));
xlA429Params.channelDirection = XL_A429_MSG_CHANNEL_DIR_TX;
xlA429Params.data.tx.bitrate  = 100000;
xlA429Params.data.tx.minGap   = XL_A429_MSG_GAP_4BIT;
xlA429Params.data.tx.parity   = XL_A429_MSG_PARITY_DISABLED;

xlStatus = xlA429SetChannelParams(xlPortHandle,
                                  xlChannelMask,
                                  &xlA429Params);
```



Example

Configures an A429 channel in Rx channel direction with enabled bitrate detection (expected bitrate should be between minimum bitrate and maximum bitrate), parity calculation disabled and a default `minGap` of 4 bit.

```
XL_A429_PARAMS xlA429Params;

memset(&xlA429Params, 0, sizeof(XL_A429_PARAMS));
xlA429Params.channelDirection = XL_A429_MSG_CHANNEL_DIR_RX;
xlA429Params.data.rx.autoBaudrate = XL_A429_MSG_AUTO_BAUDRATE_
ENABLED;
xlA429Params.data.rx.minBitrate = 97500;
xlA429Params.data.rx.maxBitrate = 102500;
xlA429Params.data.rx.parity     = XL_A429_MSG_PARITY_DISABLED;
xlA429Params.data.rx.minGap     = XL_A429_MSG_GAP_4BIT;

xlStatus = xlA429SetChannelParams(xlPortHandle,
                                  xlChannelMask,
                                  &xlA429Params);
```

14.3.2 xlA429Transmit

Syntax

```
XLstatus xlA429Transmit(
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    msgCnt,
  unsigned int*   pMsgCntSent,
  XL_A429_MSG_TX* pXlA429MsgTx
)
```

Description

The function writes ARINC 429 messages from host PC to the A429 interface. It writes the transmit data to a transmit queue and the hardware interface handles the message queue until all messages are transmitted. It is possible to write more than one message to the message queue with one `xlA429Transmit()` call. This function is an asynchronous operation.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **accessMask**
  The access mask specifies the channels to be accessed (**channel masks** support up to 64 channels, **channel indexes** support more than 64 channels). For more information, please refer to section Principles of the XL Driver Library on page 20.
- **msgCnt**
  Amount of messages to be transmitted.
- **pXlA429MsgTx**
  Points to a user buffer with messages to be transmitted, e. g. `XL_A429_MSG_TX xlA429MsgTx[100]`. At least the buffer must have the size of `msgCnt` multiplied with the size of XL_A429_MSG_TX structure (see section XL_A429_PARAMS on page 309).

Output parameters

- **pMsgCntSent**
  Number of messages successfully transferred to the transmit queue.

Return value

Returns an error code (see section Error Codes on page 360).

If `msgCnt` value is greater than the output parameter `pMsgCntSent` value, not all messages could be written to message queue and the return value `XL_ERR_QUEUE_IS_FULL` is reported.



Example

Transmit one ARINC 429 frame. The message should be sent immediately without the cyclic hardware scheduler. The global parity setting is used and a gap time of 4 bit is configured.

```
XL_A429_MSG_TX xlA429MsgTx;
unsigned int msgCnt     = 1;
unsigned int msgCntSent = 0;
memset(&xlA429MsgTx, 0, sizeof(XL_A429_MSG_TX));
xlA429MsgTx.userHandle = 0;
xlA429MsgTx.flags = XL_A429_MSG_FLAG_ON_REQUEST;
xlA429MsgTx.label = 0x04;
xlA429MsgTx.gap = 32;
xlA429MsgTx.parity = XL_A429_MSG_PARITY_DEFAULT;
xlA429MsgTx.data = 0xAABBCC;
xlStatus = xlA429Transmit(portHandle,
                          accessMask,
                          msgCnt,
                          &msgCntSent,
                          &xlA429MsgTx);
```



Example

Setup the hardware scheduler with one ARINC 429 message. This message is triggered every 100000 us (100 ms). The global parity setting is used and a gap time of 8 bit is configured.

```
XL_A429_MSG_TX xlA429MsgTx;
unsigned int msgCnt = 1;
unsigned int msgCntSent = 0;
memset(&xlA429MsgTx, 0, sizeof(XL_A429_MSG_TX));
xlA429MsgTx.userHandle = 0;
xlA429MsgTx.cycleTime = 100000;
xlA429MsgTx.flags = XL_A429_MSG_FLAG_CYCLIC;
xlA429MsgTx.label = 0x04;
xlA429MsgTx.gap = 64;
xlA429MsgTx.parity = XL_A429_MSG_PARITY_DEFAULT;
xlA429MsgTx.data = 0xAABBCC;
xlStatus = xlA429Transmit(portHandle,
                                        accessMask,
                                        msgCnt,
                                        &msgCntSent,
                                        &xlA429MsgTx);
```



Example

Transmit a burst of ARINC 429 messages. Messages are sent immediately without cyclic hardware scheduler. The global minimum gap time is used and the parity setting is odd for every single message (not used from global settings).

```
XL_A429_MSG_TX xlA429MsgTx[100];
unsigned int msgCnt = 100;
unsigned int msgCntSent = 0;
memset(xlA429MsgTx, 0, sizeof(XL_A429_MSG_TX));
for (i=0; i<nMsgCnt;i++) {
xlA429MsgTx[i].userHandle = 0;
xlA429MsgTx[i].flags = XL_A429_MSG_FLAG_ON_REQUEST;
xlA429MsgTx[i].label = 0x04;
xlA429MsgTx[i].gap = XL_A429_MSG_GAP_DEFAULT;
xlA429MsgTx[i].parity = XL_A429_MSG_PARITY_ODD;
xlA429MsgTx[i].data = 0xAABBCC;
}
xlStatus = xlA429Transmit(portHandle,
                          accessMask,
                          msgCnt,
                          &msgCntSent,
                          xlA429MsgTx);
```

14.3.3 xlA429Receive

Syntax

```
XLstatus xlA429Receive (
  XLportHandle portHandle,
  XLa429Event* pXlA429Event
)
```

Description

Retrieves one event from the event queue. This operation is synchronous.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort().
- **pXlA429Event**
  Pointer to the application allocated receive event buffer (see section XLa429Event on page 317).

Return value

Returns an error code (see section Error Codes on page 360).



Example

Read each message from the message queue

```
XLa429Event xlA429Event;

xlStatus = xlA429Receive(portHandle, &xlA429Event);

if (xlStatus != XL_ERR_QUEUE_IS_EMPTY ) {
  switch(xlA429Event.tag) {
    case XL_A429_EV_TAG_TX_OK:
    // do something with received message data
    break;

    case XL_A429_EV_TAG_TX_ERR:
    break;

    case XL_A429_EV_TAG_RX_OK:
    break;

    case XL_A429_EV_TAG_RX_ERR:
    break;

    case XL_A429_EV_TAG_BUS_STATISTIC:
    break;

   default:
break;
  }
}
```

14.4 Structs

14.4.1 XL_A429_PARAMS

Syntax

```
typedef struct s_xl_a429_params {
  unsigned short channelDirection;
  unsigned short res1;

  union {
    struct {
      unsigned int bitrate;
      unsigned int parity;
      unsigned int minGap;
    } tx;

   struct {
     unsigned int bitrate;
     unsigned int minBitrate;
     unsigned int maxBitrate;
     unsigned int parity;
     unsigned int minGap;
     unsigned int autoBaudrate;
   } rx;

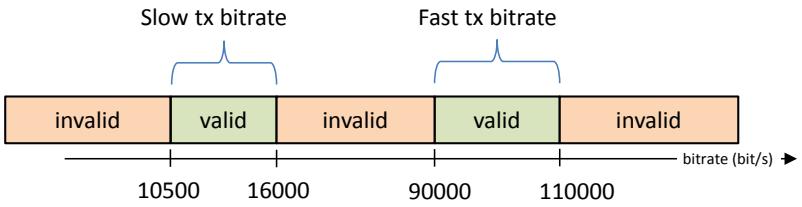
   unsigned char raw[28];
  } data;
} XL_A429_PARAMS;
```

Parameters

- **channelDirection**
  Selects the channel direction for each channel parameter. If Tx channel direction is selected, Tx struct members have to be used. If Rx channel direction is selected, Rx struct members have to be used:
  `XL_A429_MSG_CHANNEL_DIR_TX`
  `XL_A429_MSG_CHANNEL_DIR_RX`
- **res1**
  Reserved for future use.
- **tx.bitrate**
  Specifies the desired Tx channel bitrate. This value is recalculated by the A429 interface for internal clock usage (64 MHz) with a guaranteed bitrate precision of +/- 15,625 ns. Following value ranges are allowed for slow and fast bitrate settings:

  `XL_A429_MSG_BITRATE_SLOW_MIN` (10500 kbit/s)
  `XL_A429_MSG_BITRATE_SLOW_MAX` (6000 bit/s)

  `XL_A429_MSG_BITRATE_FAST_MIN` (90000 bit/s)
  `XL_A429_MSG_BITRATE_FAST_MAX` (110000 bit/s)

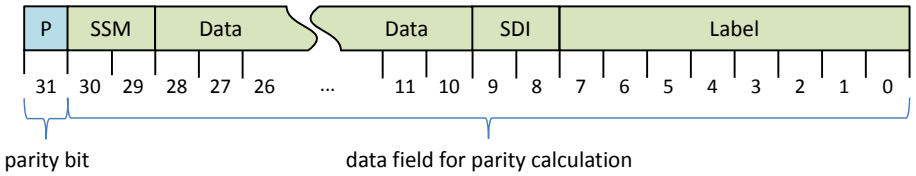


- **tx.parity**
  Global parity calculation for each Tx channel. There are three options available. It is also possible to overwrite the parity settings for every ARINC word separately. This is done in the parity field of the structure XL_A429_MSG_TX.

  `XL_A429_MSG_PARITY_DISABLED`
  Disables the parity calculation. For each transmitted Tx message the parity information has to be passed in the data field parameter separately.

  `XL_A429_MSG_PARITY_ODD`
  Enables parity bit calculation by hardware (hardware parity support). The number of bits with value 1 in bit 0 – 30 of an ARINC word is counted. If the result of the counted values is odd the parity data field is set to 0 otherwise to 1.

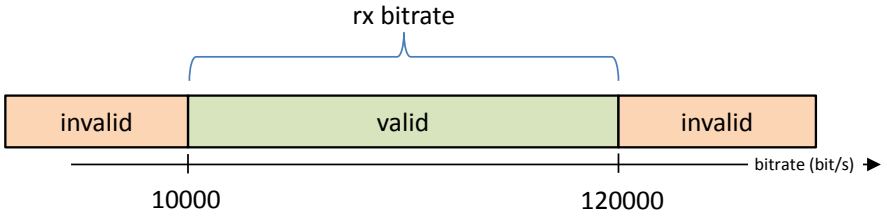
  `XL_A429_MSG_PARITY_EVEN`
  Enables parity bit calculation by hardware (hardware parity support). The parity calculation is done by hardware interface (hardware parity support). The number of bits with value 1 in bit 0 – 30 of an ARINC word is counted. If the result of the counted values is odd the parity data field is set to 1 otherwise to 0.



31 bits of data	count of 1 bits	Odd (parity bit)	Even (parity bit)
000 0000 0000 0000 0000 0000 0000 0000	0	1	0
000 0100 1000 0010 0010 1111 1000 0000	9	0	1
111 0000 1111 1111 1111 0000 1000 0000	16	1	0
111 1111 1111 1111 1111 1111 1111 1111	31	0	1

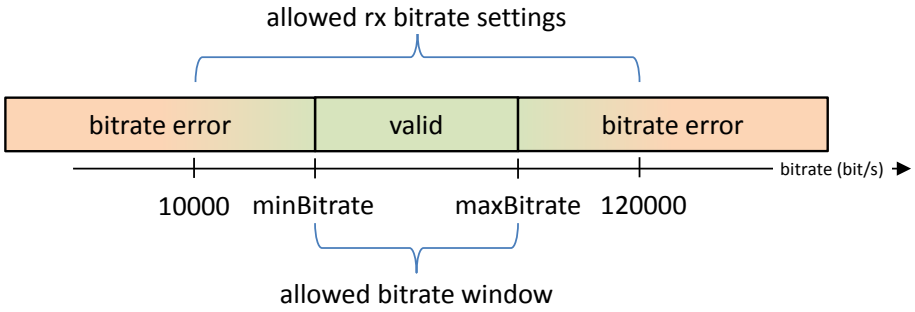
- **tx.minGap**
  Specifies the global minimum gap time between two consecutive messages. The configured gap time is inserted before a message is transmitted. Minimum Gap between two messages is defined in 1/8 bit time steps. This value is limited to 2047 (a min gap time of 255 bit). At a bitrate of 100000 bit/s the bit time is equivalent to 10 us. A setting of 32 (4 bit gap time) corresponds to a minimum gap time of 40 us. It is also possible to overwrite the minGap settings for every ARINC word separately. This is done in the gap field of the structure XL_A429_MSG_TX.

- **rx.bitrate**
  Specifies the desired Rx channel bitrate. This value is recalculated by hardware interface for internal clock usage (64 MHz) with a guaranteed bitrate precision of +/- 15,625 ns. If `autoBaudrate` is disabled this value is needed for Rx channel settings, otherwise this value is ignored. Following value ranges is allowed for bitrate settings:
  `XL_A429_MSG_BITRATE_SLOW_MIN` (10000 bit/s)
  `XL_A429_MSG_BITRATE_FAST_MINFAST_MIN` (120000 bit/s)



- **rx.minBitrate**
  Specifies the minimum allowed bitrate for Rx channels. This value is recalculated by hardware interface for internal clock usage (64 MHz) with a guaranteed bitrate precision of +/- 15,625 ns. If measured value is below this value a bitrate error is reported. Minimum allowed bitrate is 10000 bit/s (`XL_A429_MSG_BITRATE_RX_MIN`). The bitrate error check is done for every bit.

- **rx.maxBitrate**
  Specifies the maximum allowed bitrate for Rx channels. This value is recalculated by hardware interface for internal clock usage (64 MHz) with a guaranteed bitrate precision of +/- 15,625 ns. If measured value is above this value a bitrate error is reported. Maximum allowed bitrate is 120000 bit/s (`XL_A429_MSG_BITRATE_RX_MAX`). The bitrate error check is done for every bit.



- **rx.parity**
  Global parity calculation for each Rx channel. There are three options available.

  `XL_A429_MSG_PARITY_DISABLED`
  Disables the hardware parity check. There is no parity error generated.

  `XL_A429_MSG_PARITY_ODD`
  Enables odd hardware parity check. If the parity check result is even, an error is generated.

  `XL_A429_MSG_PARITY_EVEN`
  enables even hardware parity check. If the parity check result is odd, an error is generated.

- **rx.minGap**
  Specifies the global minimum gap time between consecutively received messages. Minimum Gap between two messages is defined in 1/8 bit time steps. This value is limited to 2047 (a min gap time of 255 bit). At a bitrate of 100000 bit/s the bit time is equivalent to 10 us. A setting of 32 (4 bit gap time) corresponds to a min gap time of 40 us. A gap error is reported if the measured gap time between two ARINC words is below this configured value.

- **rx.autoBaudrate**
  Enables or disables the automatic bitrate detection for Rx channels.

  `XL_A429_MSG_AUTO_BAUDRATE_DISABLED`
  Disables the automatic bitrate detection. The expected bitrate has to be set and a valid range for minimum and maximum bitrate has to be configured.

  `XL_A429_MSG_AUTO_BAUDRATE_ENABLED`
  For automatic bitrate detection the minimum and maximum bitrate has to be set. The Rx bitrate settings will be ignored. It is possible to use the complete range for minimum and maximum bitrate (XL_A429_MSG_BITRATE_RX_MIN … XL_A429_MSG_BITRATE_RX_MIN). In this mode the “average bitrate error” and “duty factor” error situation are neither checked nor reported.

- **raw**
  raw data of the data union.

- **rx.parity**
  Global parity calculation for each Rx channel. There are three options available.

  `XL_A429_MSG_PARITY_DISABLED`
  Disables the hardware parity check. There is no parity error generated.

  `XL_A429_MSG_PARITY_ODD`
  Enables odd hardware parity check. If the parity check result is even, an error is generated.

  `XL_A429_MSG_PARITY_EVEN`
  enables even hardware parity check. If the parity check result is odd, an error is generated.

- **rx.minGap**
  Specifies the global minimum gap time between consecutively received messages. Minimum Gap between two messages is defined in 1/8 bit time steps. This value is limited to 2047 (a min gap time of 255 bit). At a bitrate of 100000 bit/s the bit time is equivalent to 10 us. A setting of 32 (4 bit gap time) corresponds to a min gap time of 40 us. A gap error is reported if the measured gap time between two ARINC words is below this configured value.

- **rx.autoBaudrate**
  Enables or disables the automatic bitrate detection for Rx channels.

  `XL_A429_MSG_AUTO_BAUDRATE_DISABLED`
  Disables the automatic bitrate detection. The expected bitrate has to be set and a valid range for minimum and maximum bitrate has to be configured.

  `XL_A429_MSG_AUTO_BAUDRATE_ENABLED`
  For automatic bitrate detection the minimum and maximum bitrate has to be set. The Rx bitrate settings will be ignored. It is possible to use the complete range for minimum and maximum bitrate (`XL_A429_MSG_BITRATE_RX_MIN` … `XL_A429_MSG_BITRATE_RX_MIN`). In this mode the “average bitrate error” and “duty factor” error situation are neither checked nor reported.

- **raw**
  Raw data of the data union.



Note
Successful configured ARINC parameters can be retrieved by xlGetDriverConfig. Depending on bus type `XLbusParams` contains ARINC configured parameters. These values are the configured parameters of xlA429SetChannelParams() and not the measured/configured values of the hardware interface.



Example

Configures an A429 channel in Rx channel direction with enabled bitrate detection (expected bitrate should be between minimum bitrate and maximum bitrate), parity check is enabled (odd parity) and a default minimum gap setting of 4 bit is used.

```
XL_A429_PARAMS xlA429Params;

memset(&xlA429Params, 0, sizeof(XL_A429_PARAMS));

xlA429Params.channelDirection
= XL_A429_MSG_CHANNEL_DIR_RX;

xlA429Params.data.rx.autoBaudrate
= XL_A429_MSG_AUTO_BAUDRATE_ENABLED;

xlA429Params.data.rx.minBitrate = 97500;
xlA429Params.data.rx.maxBitrate = 102500;
xlA429Params.data.rx.parity = XL_A429_MSG_PARITY_ODD;
xlA429Params.data.rx.minGap = XL_A429_MSG_GAP_4BIT;

xlStatus = xlA429SetChannelParams(xlPortHandle, xlChannelMask,
                                  &xlA429Params);
```



Example

Configures an A429 channel in x channel direction with disabled bitrate detection (bitrate is configured to 10500 bit/s and should be between minimum bitrate and maximum bitrate), parity check is enabled (odd parity) and a default minimum gap setting of 4 bit is used.

```
XL_A429_PARAMS xlA429Params;

memset(&xlA429Params, 0, sizeof(XL_A429_PARAMS));

xlA429Params.channelDirection
= XL_A429_MSG_CHANNEL_DIR_RX;

xlA429Params.data.rx.autoBaudrate
= XL_A429_MSG_AUTO_BAUDRATE_DISABLED;

xlA429Params.data.rx.bitrate = 10500;
xlA429Params.data.rx.minBitrate = 10000;
xlA429Params.data.rx.maxBitrate = 11500;
xlA429Params.data.rx.parity = XL_A429_MSG_PARITY_ODD;
xlA429Params.data.rx.minGap = XL_A429_MSG_GAP_4BIT;

xlStatus = xlA429SetChannelParams(xlPortHandle, xlChannelMask,
                                  &xlA429Params);
```

14.4.2 XL_A429_MSG_TX

Syntax

```
typedef struct s_xl_a429_msg_tx {
  unsigned short userHandle;
  unsigned shor res1;
  unsigned int flags;
  unsigned int cycleTime;
  unsigned int gap;
  unsigned char label;
  unsigned char parity;
  unsigned short res2;
  unsigned int data;
} XL_A429_MSG_TX;
```

Parameters

- **userHandle**
  The handle is provided by the application and is used for the event assignment to the corresponding transmit request.
- **res1**
  Reserved for future use.

- **flags**
  Message flag of ARINC 429 transmit message. This flag indicates if message is transmitted on request (is written directly to message queue), cyclically (is registered in hardware scheduler) or deleted (removed from hardware scheduler).

  `XL_A429_MSG_FLAG_ON_REQUEST`
  Transmit message immediatley without writing data to hardware scheduler (data is transferred to message queue). On request messages could interfere with cyclic messages on the same channel. This message has a higher precedence than a cyclic called message.

  `XL_A429_MSG_FLAG_CYCLIC`
  Adds or modifies an entry for the hardware scheduler.
  `cycleTime` has to be defined in microseconds. If a message is entered initially to the hardware scheduler it is sent immediately. Afterwards the message is scheduled based on the given `cycleTime`. On subsequent cyclic calls all data fields of the corresponding label (including `cycleTime`) are updated. If `cycleTime` changes, the actual timer is cancelled and restarted with the new `cycleTime` value. If `cyclicTime` is zero, only the payload data is updated.

  `XL_A429_MSG_FLAG_DELETE_CYCLIC`
  Removes an ARINC word entry from the hardware scheduler.

- **cycleTime**
  Cycle time in microseconds. The value is evaluated only for `flags = XL_A429_MSG_FLAG_CYCLIC`. The maximum allowed value for `cycleTime` is `XL_A429_MSG_CYCLE_MAX` (approx. 17 minutes).

- **gap**
  Gap time between two messages. Gap time is inserted before the message is transmitted. Gap is defined in 1/8 bit time steps. At a bitrate of 100000 bit/s the bit time is equivalent to 10 us. A setting of 32 (4 bit gap time) corresponds to a gap time of 40 us. The maximum setting for this value is `XL_A429_MSG_GAP_MAX` (131071 bit gap time) corresponds to a gap time of 1,31071 s at a bitrate of 100000 bit/s.

  XL_A429_MSG_GAP_DEFAULT
  Enables global setting. If this value for gap is selected, global minGap (for Tx channel direction) setting of XL_A429_PARAMS is used.

- **label**
  Label of ARINC word.

- **parity**
  Parity bit calculation of message.

  `XL_A429_MSG_PARITY_DEFAULT`
  Enables the global setting of parity. The global setting of XL_A429_PARAMS is used.

  `XL_A429_MSG_PARITY_DISABLED`
  Disables the hardware parity generation. The user controls the parity by setting label and data.

  `XL_A429_MSG_PARITY_ODD`
  Odd parity is generated by hardware.

  `XL_A429_MSG_PARITY_EVEN`
  Even parity is generated by hardware.

- **res2**
  Reserved for future use.

- **data**
  Data field of ARINC word. Contains SSM, SDI and data field. If parity field is set to `XL_A429_MSG_PARITY_DISABLED` the data field contains the parity information.

14.5 Events

14.5.1 XLa429Event

Syntax

```
typedef struct s_xl_a429_event {
  unsigned int              size;
  XLa429EventTag            tag;
  unsigned short            channelIndex;
  unsigned int              userHandle;
  unsigned short            flagsChip;
  unsigned short            reserved;
  XLuint64                  time stamp;
  XLuint64                  timestampSync;
  union s_xl_a429_tag_data tagData;
} XLa429Event;

typedef unsigned short      XLa429EventTag;

union s_xl_a429_tag_data {
  XL_A429_EV_TX_OK          a429TxOkMsg;
  XL_A429_EV_TX_ERR         a429TxErrMsg;
  XL_A429_EV_RX_OK          a429RxOkMsg;
  XL_A429_EV_RX_ERR         a429RxErrMsg;
  XL_A429_EV_BUS_STATISTIC  a429BusStatistic;
  XL_SYNC_PULSE_EV          a429SyncPulse;
};
```

Description

All XL API ARINC 429 events are transmitted and indicated via this event structure.

Parameters

- **size**
  Size of the complete ARINC 429 event, including header and payload data.
- **tag**
  Event tag of this event.

  `XL_A429_EV_TAG_TX_OK`
  when a message was transmitted completely.

  `XL_A429_EV_TAG_TX_ERR`
  when an error was detected by the transmitter.

  `XL_A429_EV_TAG_RX_OK`
  when a message was received entirely.

  `XL_A429_EV_TAG_RX_ERR`
  when an error is detected by the receiver.

  `XL_A429_EV_TAG_BUS_STATISTIC`
  when a bus statistic is requested.

  `XL_SYNC_PULSE`
  when a sync pulse is requested.
- **channelIndex**
  Contains the logical channel number.

- **userHandle**
  Application specific handle that may be used to link associated events, e. g. a transmit confirmation to the original send request. On cyclic messages the user handle contains always the value that is configured by xlA429Transmit() function.
- **flagsChip**
  Special flags of an event, e. g. indicates an overrun of the application receive queue. This value is set once if overrun is detected.
  `XL_QUEUE_OVERFLOW`
- **time stamp**
  Raw time stamp (starting with 0 when device is powered) in nanoseconds. This value is not touched with xlResetClock() and time synchronization has no effect on this time stamp.
- **timestampSync**
  Synchronized time stamp in nanoseconds (PC → device). Time synchronization is applied if enabled in Vector Hardware Manager time synchronization settings. Offset correction is possible with xlResetClock().

14.5.2 XL_A429_EV_TX_OK

Syntax

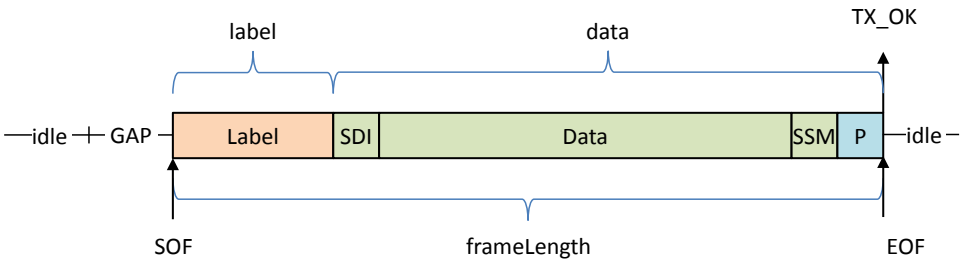
```
typedef struct s_xl_a429_ev_tx_ok {
  unsigned int   frameLength;
  unsigned int   bitrate;
  unsigned char  label;
  unsigned char  msgCtrl;
  unsigned short res1;
  unsigned int   data;
} XL_A429_EV_TX_OK;
```

Description

This event signalizes a transmitted ARINC 429 message.

Tag

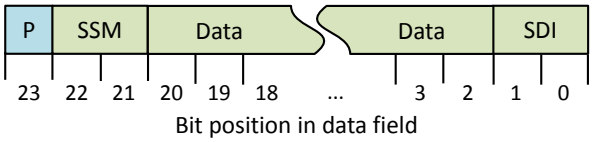
`XL_A429_EV_TAG_TX_OK`



Parameters

- **frameLength**
  Time between start of frame and end of frame in nanoseconds.
- **bitrate**
  Bitrate of transmitted message. This value is the configured bitrate for transmission (calculated by hardware interface) and not the measured value.
- **label**
  Label of ARINC word.

- **msgCtrl**
  Indicates event is generated on request (requested by user application) or cyclic (scheduled by network interface).
  `XL_A429_MSG_CTRL_ON_REQUEST`
  `XL_A429_MSG_CTRL_CYCLIC`
- **res1**
  Reserved for future use.
- **data**
  Data field of ARINC word. Contains parity, SSM, SDI and data field.



14.5.3 XL_A429_EV_TAG_TX_ERR

Syntax

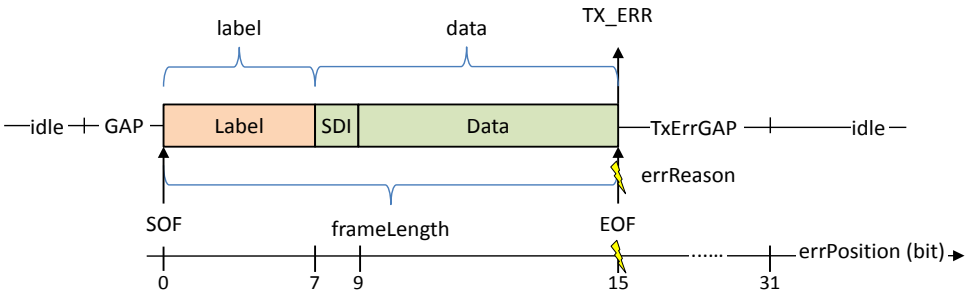
```
typedef struct s_xl_a429_ev_tx_err {
  unsigned int  frameLength;
  unsigned int  bitrate;
  unsigned char errorPosition;
  unsigned char errorReason
  unsigned char label
  unsigned char res1;
  unsigned int  data;
} XL_A429_EV_TX_ERR;
```

Description

This event informs about a failed transmission.

Tag

`XL_A429_EV_TAG_TX_ERR`



Parameters

- **frameLength**
  Time between start of frame and end of frame in nanoseconds. In case of error this is the time between start of frame and detected error.
- **bitrate**
  Bitrate of transmitted message. This value is the configured bitrate for transmission (calculated by hardware interface) and not the measured value.
- **errorPosition**
  Bit position of error. Valid range is between bit position 0 and 31.

- **errorReason**
  Error reason of event. Following error reasons are possible:

  `XL_A429_EV_TX_ERROR_ACCESS_DENIED`
  Transmission is not possible because of missing “null” state on bus (bus is not idle).

  `XL_A429_EV_TX_ERROR_TRANSMISSION_ERROR`
  Transmitter detected wrong bus pattern at end of half bit.

- **label**
  Label of ARINC word. If error position > 7 the value is valid.

- **res1**
  Reserved for future use.

- **data**
  Data field of ARINC word. Contains parity, SSM, SDI and data field. It depends on the error position which data fields are valid.

14.5.4 XL_A429_EV_TAG_RX_OK

Syntax

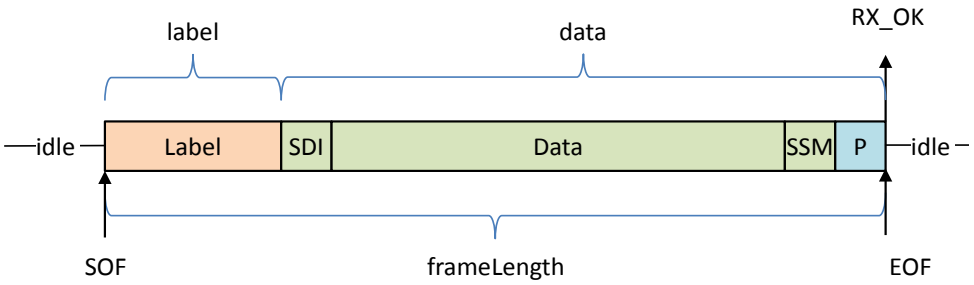
```
typedef struct s_xl_a429_ev_rx_ok {
  unsigned int   frameLength;
  unsigned int   bitrate;
  unsigned char  label;
  unsigned char  res1[3];
  unsigned int   data;
} XL_A429_EV_RX_OK;
```

Description

This event signalizes an error free received ARINC 429 message.

Tag

`XL_A429_EV_TAG_RX_OK`



Parameters

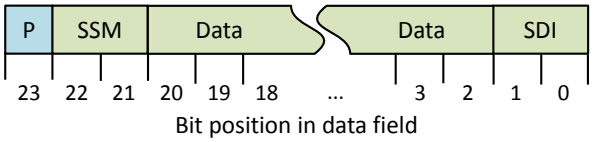
- **frameLength**
  Time between start of frame and end of frame in nanoseconds.

- **bitrate**
  Bitrate of received message. This value is the measured bitrate for reception. The bitrate is the average value through the complete reception of ARINC word.

- **label**
  label of ARINC word.

- **res1**
  Reserved for future use.

- **data**
  Data field of ARINC word. Contains parity, SSM, SDI and data field.



14.5.5 XL_A429_EV_TAG_RX_ERR

Syntax

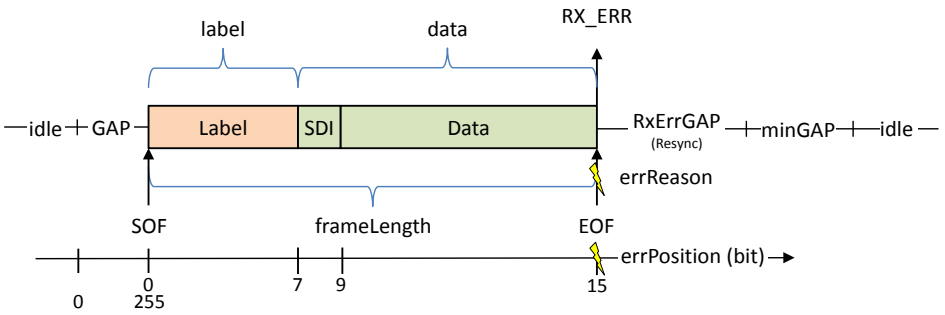
```
typedef struct s_xl_a429_ev_rx_err {
  unsigned int frameLength;
  unsigned int bitrate;
  unsigned int bitLengthOfLastBit;
  unsigned char errorPosition;
  unsigned char errorReason;
  unsigned char label;
  unsigned char res1;
  unsigned int  data;
} XL_A429_EV_RX_ERR;
```

Description

This event signalizes an error related to a received ARINC 429 message.

Tag

`XL_A429_EV_TAG_RX_ERR`



Parameters

- **frameLength**
  Time between start of frame and end of frame in nanoseconds. This is the time between start of frame and detected error.

- **bitLengthOfLastBit**
  Time between start of last bit and end of frame (error detection) in nano-seconds. This value is only valid for the following error reasons:

  `XL_A429_EV_RX_ERROR_BITRATE_LOW`
  Measured time is below configured minimum bitrate limit. This value gives the erroneous received bit length and corresponds to the channel parameter `minBitrate`.

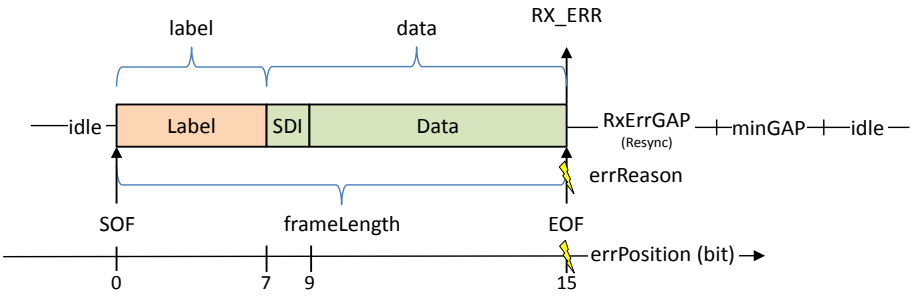
  `XL_A429_EV_RX_ERROR_BITRATE_HIGH`
  Measured time is above configured maximum bitrate limit. This value gives the erroneous received bit length.

  `XL_A429_EV_RX_ERROR_FRAME_FORMAT`
  Measured time for frame format violation. This value gives the timely position of the error in the bit.

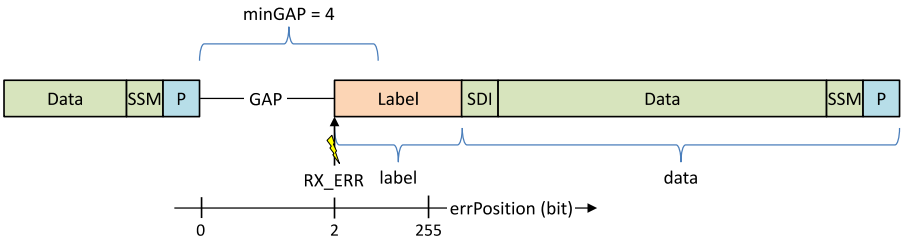
  `XL_A429_EV_RX_ERROR_CODING_RZ`
  Measured time for level violation. This value gives the timely position of the error in the bit.

- **bitrate**
  Bitrate of received message. This value is the measured bitrate for reception. The bitrate is the average value through the complete reception of ARINC word. This value is only valid for the following error reasons:
  `XL_A429_EV_RX_ERROR_PARITY`
  `XL_A429_EV_RX_ERROR_DUTY_FACTOR`
  `XL_A429_EV_RX_ERROR_AVG_BIT_LENGTH`

- **errorPosition**
  Bit position of error. For all reception errors (except `minGap` violation error) the error position range is from 0 to 31 (bit position of error occurred in ARINC word):



  For `minGap` violation the error position range is from 0 to 255 (bit position of error occurred in gap field). Label and data field does not contain valid values:



- **errorReason**
  Error reason of event.

  `XL_A429_EV_RX_ERROR_GAP_VIOLATION`
  Is reported after a violation of configured `minGap` (edge was detected on bus while running `minGap` time).

  `XL_A429_EV_RX_ERROR_PARITY`
  Received parity value doesn't match to calculated or configured parity value.

  `XL_A429_EV_RX_ERROR_BITRATE_LOW`
  Received bit length ecxeeded the configured `minBitrate` in XL_A429_PARAMS. Each bit length is checked while reception of ARINC word.

  `XL_A429_EV_RX_ERROR_BITRATE_HIGH`
  Received bit length is below configured `maxBitrate` in XL_A429_PARAMS. Each bit length is checked while reception of ARINC word.

  `XL_A429_EV_RX_ERROR_FRAME_FORMAT`
  Edge received on bus in last half bit of ARINC word.

  `XL_A429_EV_RX_ERROR_CODING_RZ`
  Unexpected edge received on bus violating RZ code e. g. voltage switching from -10 V ... 10 V or vice versa.

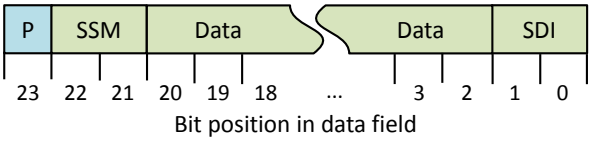
  `XL_A429_EV_RX_ERROR_DUTY_FACTOR`
  Duty Factor errors are reported at the end of the frame if the duty factor of a single bit was wrong (edge not in expected range). Range of duty factor is defined between 40% and 60% of the configured bitrate. Error position defines the first bit with the duty factor error.

  `XL_A429_EV_RX_ERROR_AVG_BIT_LENGTH`
  Average bit length error is reported if the deviation of the average bit length of the complete frame is outside the defined range (±1.0%).

- **label**
  Label of ARINC word. If error position > 7 the value is valid (except `minGap` violation error). Label field does not contain valid values for `minGap` violation error.

- **res1**
  Reserved for future use.

- **data**
  Data field of ARINC word. Contains parity, SSM, SDI and data field if available. Data field does not contain valid values for `minGap` violation error.



14.5.6 XL_A429_EV_BUS_STATISTIC

Syntax

```
typedef struct s_xl_a429_ev_bus_statistic {
  unsigned int busLoad;
  unsigned int res1[3];
```

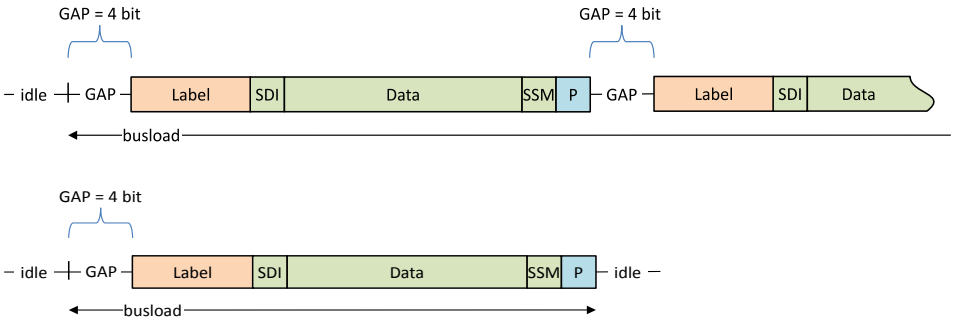
```
} XL_A429_EV_BUS_STATISTIC;
```

Description

This event is generated every second after activation of channel and reports bus statistic information.

Tag

`XL_A429_EV_TAG_BUS_STATISTIC`



Parameters

- **busLoad**
  In percent (resolution is 0.01 percent per digit).
  `busLoad` calculation includes data frame with a fixed gap of 4 bit.
- **res1**
  Reserved for future use.

14.6 Application Examples

14.6.1 xlA429Control

14.6.1.1 General Information

Description

The ARINC 429 Control is a small MFC GUI tool that demonstrates how to configure an ARINC 429 device, how to activate a channel and how to receive data indications.

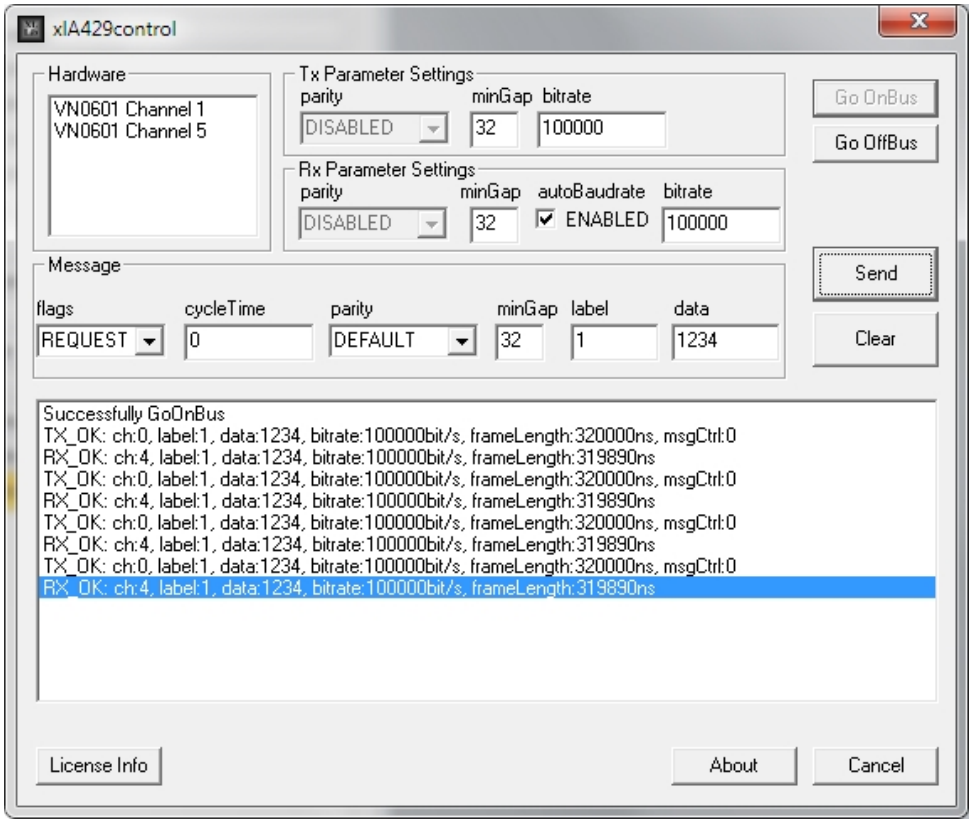


Figure 53: Running xlA429Control

15 .NET Wrapper

In this chapter you find the following information:

15.1 Introduction **327**

15.2 Vector.XlApi **328**

15.3 vxlapi_NET **334**

15.4 Application Examples **341**

15.1 Introduction

General information

The XL Driver Library ships with two .NET Wrappers:

- **Vector.XlApi** is a modernized wrapper implemented as a .NET Standard 2.0 library. Currently, it supports Ethernet in network-based mode and CAN, LIN, and DAIO in channel-based mode.
- **vxlapi_NET** is the legacy wrapper that is based on .NET Framework 3.5. It supports CAN, LIN, DAIO, FlexRay, ARINC 429, K-Line and Ethernet in channel-based mode.

If necessary, the two wrappers can be used together in one application. **Vector.XlApi** is intended to replace **vxlapi_NET** in the future.

15.2 Vector.XlApi

15.2.1 Overview

Description

Vector.XlApi enables a .NET application to access the native XL Driver Library. It is primarily intended to be used with the C# programming language.

As a wrapper, it attempts to remain close to both the C-language API and the conventions that have been established by its predecessor **vxlapi_NET**.

Compared to **vxlapi_NET**, it has following advantages:

- Integration with todays .NET ecosystem, for example .NET Core and NuGet package management.
- Support for new XL API features like `xlCreateDriverConfig` and the network-based Ethernet mode.
- Shorter naming that more closely resembles the conventions of the .NET environment.
- Exception based error handling.
- Reduced wrapping overhead.

Framework support

Vector.XlApi is a .NET Standard 2.0 library. As such, it is supported by:

- .NET Framework 4.6.1 or later. Note that Visual Studio or the NuGet package manager will pull several compatibility libraries if a .NET Standard 2.0 Library is referenced in a .NET Framework project before version 4.7.2.
- .NET Core 2.0 or later.

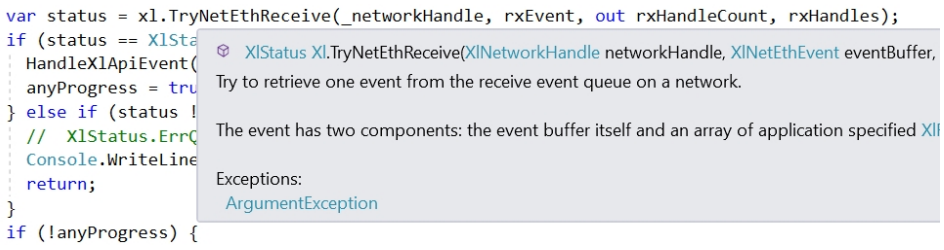
Documentation

As a starting point, we recommend looking at one of the **Vector.XlApi** samples:

- `xlCanDemo`
- `xlLinDemo`
- `xlLinDemo Single`
- `xlNetEthDemo`

These samples were written to introduce the Vector.XlApi wrapper and the network-based Ethernet mode.

Vector.XlApi contains embedded code documentation that is displayed by Visual Studio IntelliSense.



IntelliSense also shows matching functions and constants while typing. Click on a type and press **<F12>** to display the interface of the type, including its embedded documentation.

15.2.2 Design

General

The wrapper is contained in the single assembly `Vector.XlApi.dll` and all exported types are part of the `Vector.XlApi` namespace. Every type name starts with "`Xl`".

Xl class

The `Xl` class is the main class of **Vector.XlApi**. Every `Xl` function exported by the wrapper is a method of this class. Additionally, the `Xl` class exports all constants that are not members of an enum.

As part of the construction of the first Xl instance, the wrapper calls `xlOpenPort` and throws an `XlException` if that call failed. Apart from that, the Xl object is stateless. The application may allocate additional Xl objects wherever it is convenient.

Functions

Functions have the same name as in the native API, except that the `Xl` prefix is stripped and occasionally replaced with "`Try`":

- Functions that do not start with `Try` throw an `XlException` if the native function returns a different status than `XL_SUCCESS`. The wrapper function sets the `Status` property of the `XlException` object to the status returned by the native function. Examples are:
- `void NetActivateNetwork(XlNetworkHandle networkHandle)`
- `void SetApplConfig(XlApplConfig applConfig)`
- Functions that start with `Try` return the status of the native function. Examples are:
  - `XlStatus TryNetRequestMACAddress(XlNetworkHandle networkHandle, out XlEthMacAddress macAddress)`
  - `XlStatus TryGetApplConfig(XlApplConfig applConfig)`

Most functions that start with `Try` have an exception-throwing counterpart for use cases where handling exceptions is preferred over interpreting status codes. In general, functions that have no `Try` variant are expected to fail only if used incorrectly or under exceptional conditions, such as a device disconnect.

Enums

The **XL API** has many enumeration types, although the enumeration is often only defined implicitly by a common prefix.

Vector.XlApi exposes all enumerations as explicit types with camel case naming. The type name is usually derived from the common prefix.

Examples are:

- `XlStatus.Success, XlStatus.ErrQueueIsEmpty, XlStatus.Error, …`
- `XlHwType.None, XlHwType.Virtual, XlHwType.Vn1630, …`
- `XlBusType.Can, XlBusType.Ethernet, …`
- `XlEthEventTag.FramerxSimulation, XlEthEventTag.ChannelStatus …`



Note
Enums are just integers that have names for certain values. Programmers should expect that the wrapper may return unnamed values in an enum. For example, the only named value of `XlNetworkHandle` is `Invalid`(-1), but the wrapper will return handle values like 0, 1, 2, …

Constants

Constants that specify integer values like the size of a buffer are not part of an enum. **Vector.XlApi** exposes them as constants of the `Xl` class instead. Examples are:

- `Xl.EthRxFifoQueueSizeMin`
- `Xl.EthMacaddrOctets`
- `Xl.ApplconfigMaxChannels`

Structures

Vector.XlApi allows the application to work directly on the buffer that is shared with the native library. Native structures are wrapped by correspondingly named classes that hold the shared buffer and expose properties that access the fields of the native structure. Therefore, **Vector.XlApi** does not need to convert between the managed and native structure on each function call.

Class names are in camel case. Examples are:

- `XlIdriverConfig`
- `XlNetEthEvent`

Field names start with lower letters. Arrays implement the `IList<T>` interface and can be accessed with indexers. Sub structures within a larger structure are directly accessible from the larger structure but not with the usual dot syntax but instead with underscores.

For example, if `e` is an object of type `XlNetEthEvent`, the following code is valid:

- `e.tag = XlEthEventTag.ChannelStatus;`
- `e.tagData_rawData[42] = 1;`
- `e.tagData_frameSimRx_dataLen = 100;`

Structures, sub structures and arrays can be copied into and out of the underlying buffer with copy constructors and the `CopyFrom()`, `CopyTo()` and `ToArray()` methods.

For example:

- `var x = new XlNetEthDataframeRx(e.tagData_frameSimRx);`
- `x.CopyTo(e.tagData_frameSimRx);`
- `var y = e.tagData_rawData.ToArray();`
- `e.tagData_rawData.CopyFrom(y, 0);`

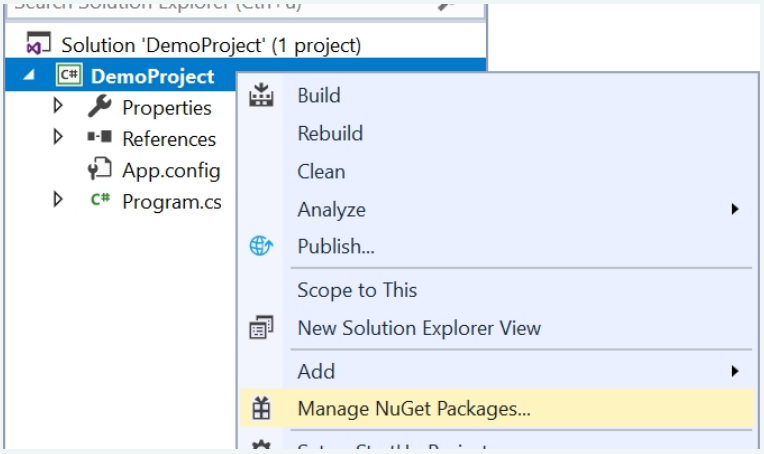
15.2.3 Including the Wrapper in a Project

NuGet

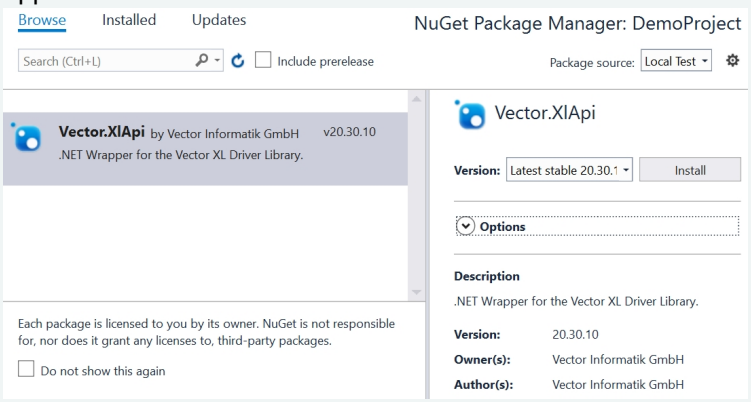
Vector.XlApi ships as a NuGet package. If you use the NuGet package management as part of your development process, it is preferred to add `Vector.XlApi` as a package to you project. The advantage is, that the package contains both the wrapper and the native `vxlapi.dll`/`vxlapi64.dll`. The package copies the native libraries to the application output folder as part of the build process.

Step by Step Procedure

1. Copy the file `bin\Vector.XlApi-<VersionNumber>.nupkg` to your teams private NuGet feed. Do not copy it to a public repository such as Nuget.org.
2. In Visual Studio, right-click on your project and select **Manage NuGet Packages…**.



3. If the package source is correctly set up, the **Vector.XlApi** package should appear under the **Browse** tab.



4. Select the package and click **[Install]**.

Assembly reference

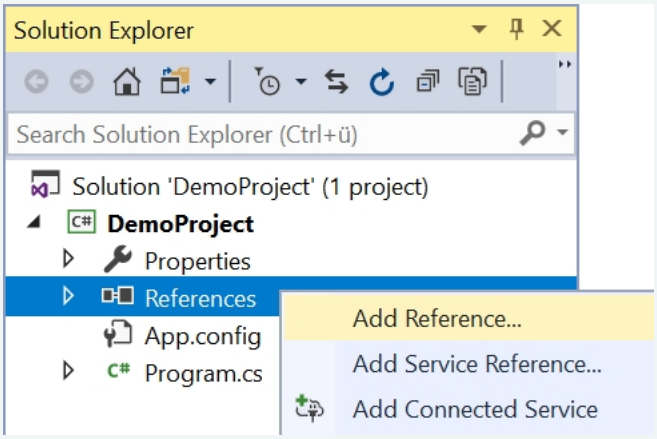
Alternatively, **Vector.XlApi** can be added to a project by referencing the `Vector.XlApi.dll` assembly. This is same installation method as is used by `vxlapi_NET`. In this case, you must ensure that the native`vxlapi.dll`/`vxlapi64.dll` are available in the same or a more recent version than the referenced `Vector.XlApi` assembly.

This can be achieved either by:

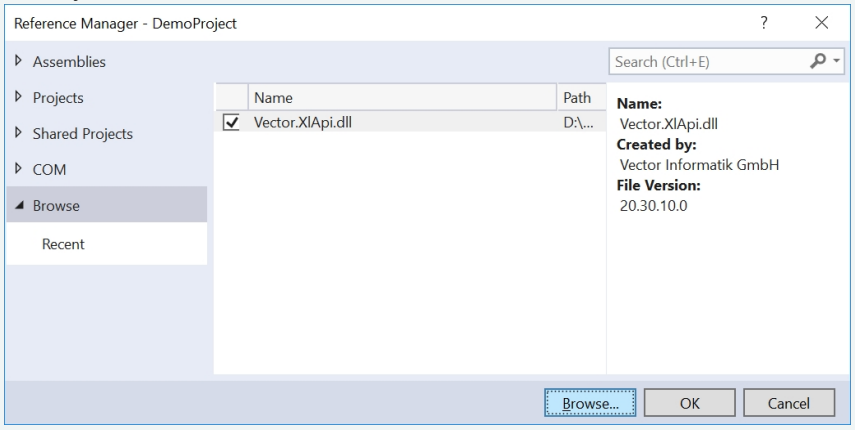
- Ensuring that users of your application have the latest XL Driver Library Setup installed, as it copies the `vxlapi.dll`/`vxlapi64.dll` to system folders.
- Shipping the `vxlapi.dll`/`vxlapi64.dll` with your application.

Step by Step Procedure

1. In Visual Studio, right-click on **References** (Solution Explorer, below your project) and select **Add Reference**.



2. The **Reference Manager** opens. Click **Browse** and choose the `Vector.XlApi.dll` file in the `bin\netstandard2.0` folder of the XI Driver Library installation.



3. Close the dialog with **[OK]**.

15.2.4 Relationship with vxlapi_NET

Combined use

The following restrictions apply if **Vector.XlApi** and **vxlapi_NET** are used within the same application:

- The application must construct the first instance of the `Vector.XlApi.Xl` class before it calls any method of **vxlapi_NET**. The constructor of the `Xl` class opens the driver.
- Therefore, the methods `vxlapi_NET.XLDriver.XL_OpenDriver()` and `vxlapi_NET.XLDriver.XL_CloseDriver()` must not be called.

As both wrappers operate on the same native library instance, values may be exchanged between the wrappers. This will, however, require a lot of casts. For example, the application may use `Vector.XlApi.Xl.CreateDriverConfig()` in favor of `vxlapi_NET.XLDriver.XL_GetDriverConfig()` to find a channel and later open that channel with `vxlapi_NET.XLDriver.XL_OpenPort()`.

15.3 vxlapi_NET

Note
Please note that this wrapper is deprecated and is no longer being developed. Please use the available alternative, Vector.XlApi.

15.3.1 Overview

Description

The **XL API .NET Wrapper** allows an easy integration of the **XL Driver Library** in any .NET environment. This means that Vector device can be accessed in any .NET programming language, for example in C# or Visual Basic .NET.

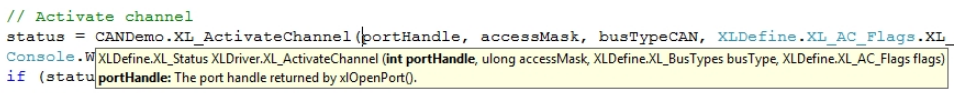
The **XL API .NET Wrapper** consists of the single .NET assembly `vxlapi_NET.dll` which offers the functionality of the XL Driver Library by using three major classes:

- **XLDriver**
  .NET methods accessing the XL API.
- **XLClass**
  Predefined classes/parameters required by XLDriver.
- **XLDefine**
  Predefined values that are required by XLDriver/XLClass.

The usage of the **XL API .NET Wrapper** is similar to the native XL API. It i recommended to look up the flowcharts in the general XL API description and to use the according .NET methods. Compared to the native XL API, the .NET method names differs only in the prefix, e. g.

Wrapper: `XL_OpenDriver()`
Native XL API: `xlOpenDriver()`

The required parameters of the .NET methods can be looked up by using the IntelliSense feature of the IDE, for example:



Examples

The **XL Driver Library** setup also contains a few examples in different .NET languages that explain the usage in each environment.

WARNING!
THE INCLUDED WRAPPER IS PROVIDED “AS-IS”. NO LIABILITY OR RESPONSIBILITY FOR ANY ERRORS OR DAMAGES.

Note
The .NET Wrapper only supports CAN, LIN, DAIO, FlexRay, Ethernet and ARINC 429 and can be found in the XL Driver Library in `\Drivers\XL Driver Library\bin`.

In order to run the .NET wrapper with your application, the general libraries `vxlapi.dll`/`vxlapi64.dll` have also to be copied into the execution folder of your application.

Note
To run the XL API .NET Wrapper, framework .NET 3.5 or higher is required.

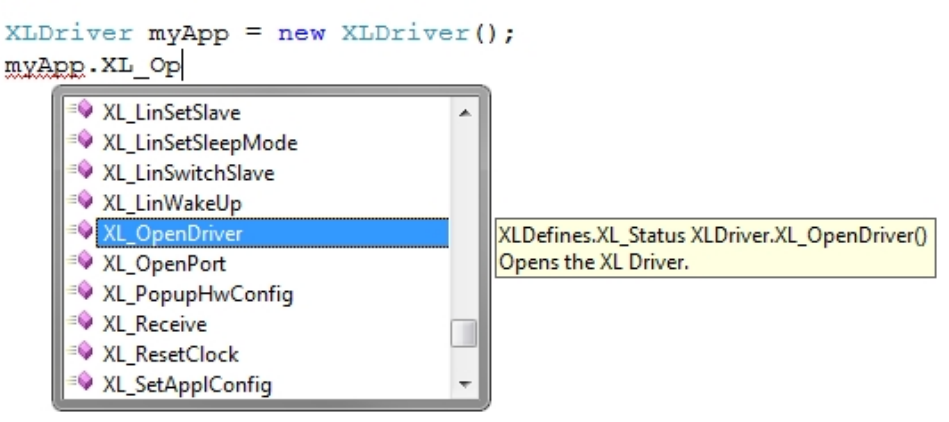
15.3.2 XLDriver - Accessing Driver

Accessing XL API

In .NET, the native XL API can be accessed by the major class `XLDriver` which supports most of the native functions.

Note

Please refer to the general **XL Driver Library** documentation for further information on available functions or use the IntelliSense feature in your IDE to find all available .NET methods provided by `XLDriver`.



15.3.3 XLClass - Storing Data/Parameters

Predefined classes

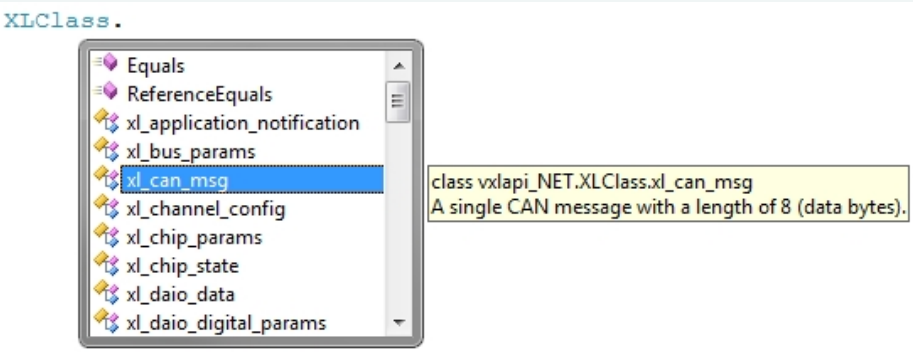
Some of the XL API .NET methods expect objects (parameters). For this case, all required classes are predefined in the class `XLClass` and ready to use. Most of these classes are clones of the XL API structures. Please refer to the general **XL Driver Library** documentation for further information.

Here are some examples of these predefined classes:

- **xl_driver_config**
  For storing the driver configuration.
  Required by method `XL_GetDriverConfig()`.
- **xl_event**
  Contains data to be transmitted.
  Required by method `XL_CanTransmit()`.
- **xl_event_collection**
  For storing one or more xl_events.
  Required by method `XL_CanTransmit()`.
- **xl_bus_params**
  Used by subclass `xl_channel_config`.
- **xl_channel_config**
  Used by subclass `xl_driver_config`.
- **xl_can_message**
  Used by subclass `xl_event`.
- **xl_chip_params**
  Used by method `XL_CanSetChannelParams()`.
- xl_**linStatPar**
  Used by method `XL_LinSetChannelParams()`.

Note

More predefined classes in `XLClass` can be found via the IntelliSense feature in your IDE.



15.3.4 XLDefine - Using Predefined Values

Definitions and enumerations

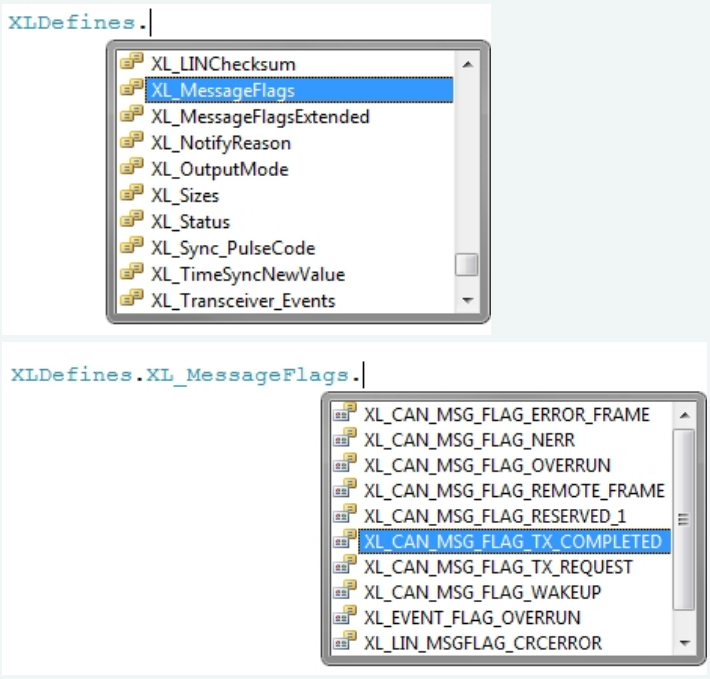
The class `XLDefine` offers a wide range of enumerations for easy access to values and definitions. Most of these definitions can be found in `vxlapi.h` of the native XL API.

Here are some examples of these predefined definitions:

- **XLDefine.XL_Status**
  ```
  .XL_SUCCESS
  .XL_PENDING
  .XL_ERR_QUEUE_IS_EMPTY
  .XL_ERR_QUEUE_IS_FULL
  .XL_ERROR
  …
  ```
- **XLDefine.XL_HardwareType**
  ```
  .XL_HWTYPE_NONE
  .XL_HWTYPE_VIRTUAL
  .XL_HWTYPE_VN1630
  .XL_HWTYPE_VN1640
  …
  ```
- **XLDefine.XL_BusType**
  ```
  .XL_BUS_TYPE_CAN
  .XL_BUS_TYPE_DAIO
  .XL_BUS_TYPE_FLEXRAY
  …
  ```

Note

More definitions can be found via the IntelliSense feature in your IDE.
Example: `XL_MessageFlags.`

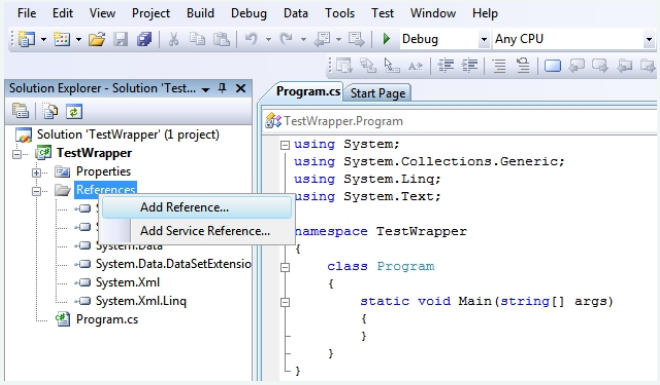


15.3.5 Including the Wrapper in a New .NET Project

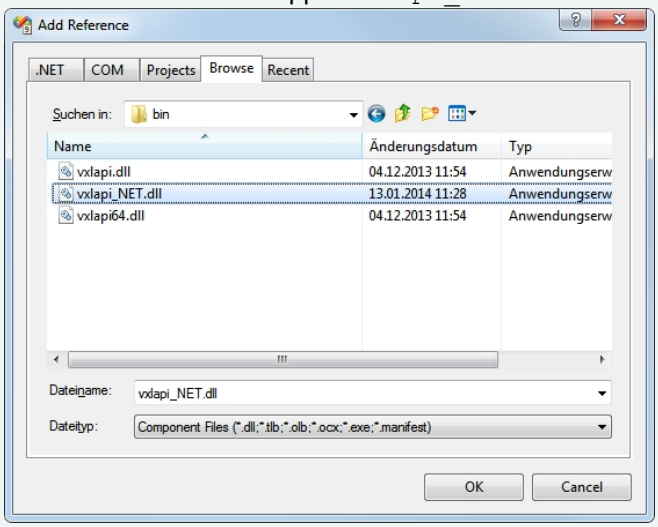


Step by Step Procedure

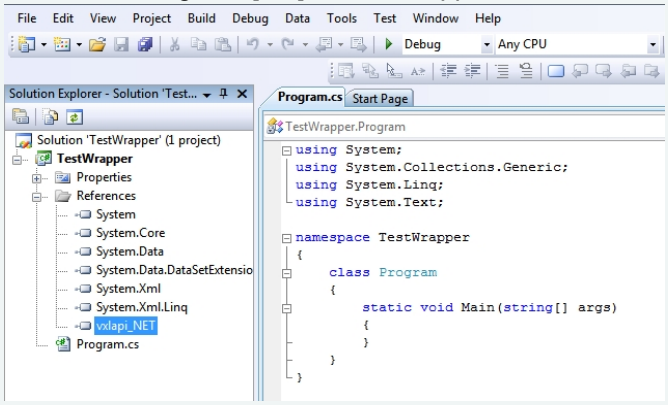
1. Copy the general XL Driver Library `vxlapi.dll`/`vxlapi64.dll` to your execution folder of your project (`\Debug` or `\Release`).
2. In VS2008, right-click on **References** (Solution Explorer) and select **Add Reference…**



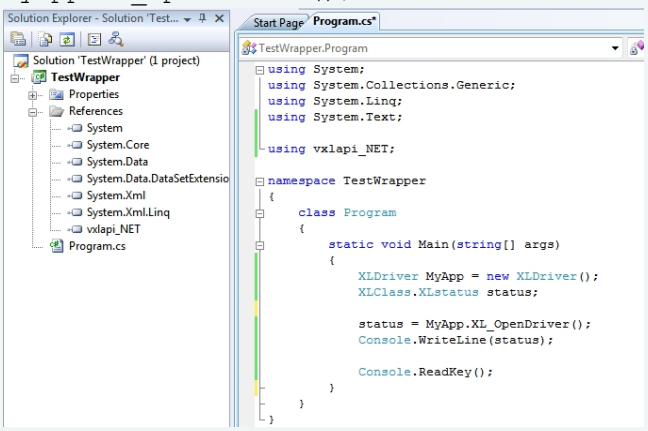
3. Browse for the .NET wrapper `vxlapi_NET.dll`.



4. Close the dialog with [OK]. The DLL appears in the Solution Explorer.



5. Enter the following line in the top of your source code to access the wrapper:
   `using vxlapi_NET;`
6. Now you are able to instantiate a main object from class XLDriver:
   `XLDriver MyApp = new XLDriver();`
7. Try to open the port by entering the line:
   `MyApp.XL_OpenDriver();`



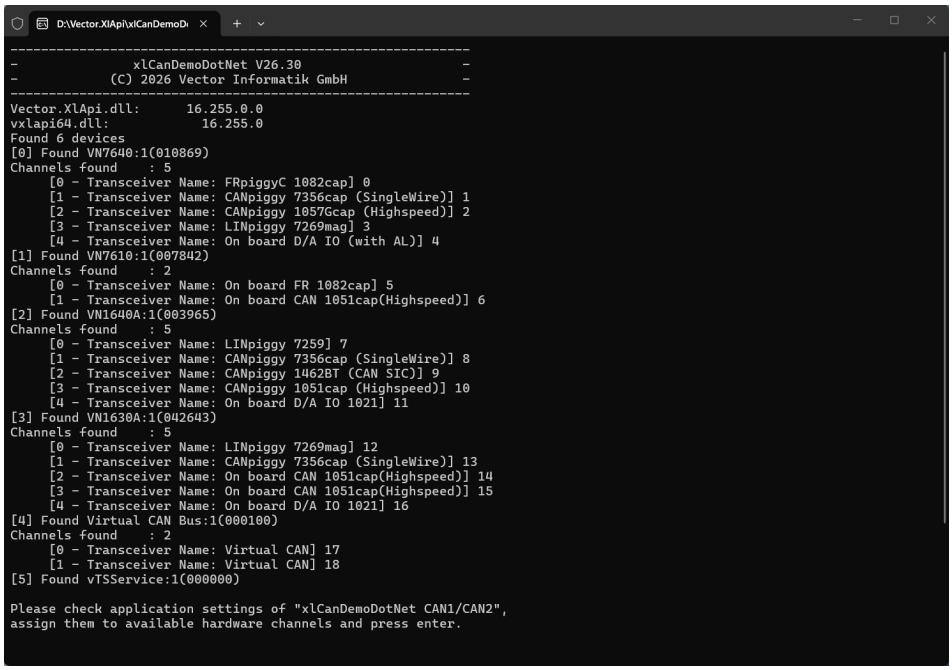
Note
Take a look at our examples (source code) in the XL Driver Library for further information.

15.4 Application Examples

15.4.1 xlCanDemo Vector.XlApi

Description

This C# example demonstrates how to access a Vector CAN interface.



Starting the example

When the example starts, it looks for the **xlCanDemoDotNet** application in the Vector Hardware Manager. Since this application name is not registered initially, the example automatically creates it. Then, the application channels CAN 1 and CAN 2 must be manually assigned to a physical Vector CAN interface, such as the VN1630A. Both channels must also be physically connected, e. g., via CANcable1.

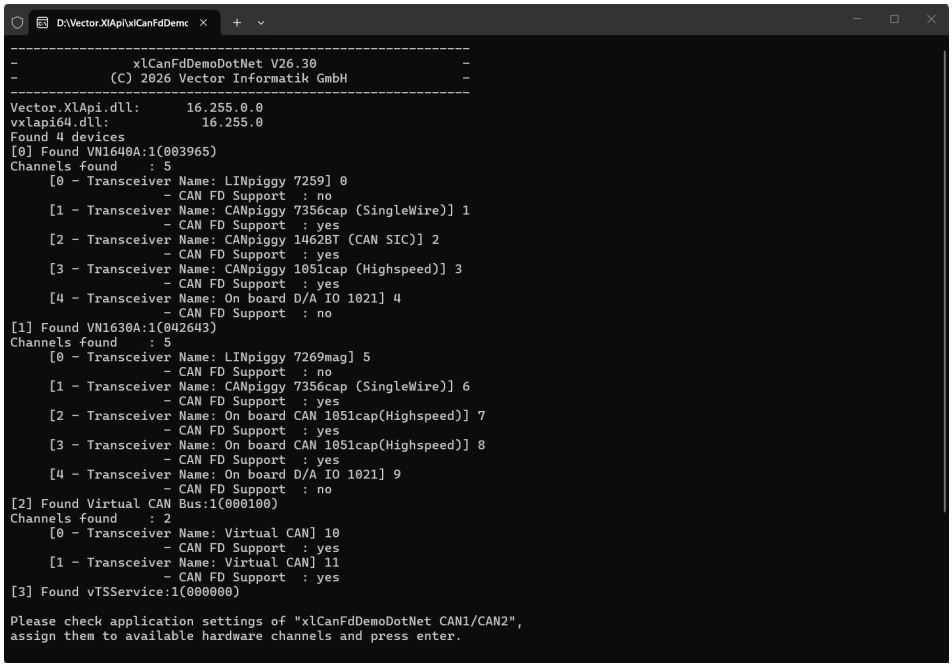
Send and receive messages

By pressing the **[ENTER]** key, the example sends and receives CAN messages. The message is sent over the first configured channel and is received by the second one.

15.4.2 xlCanFdDemo Vector.XlApi

Description

This example demonstrates how to access the XL API for CAN FD.



Starting the example

When the example starts, it looks for the **xlCanFdDemoDotNet** application in the Vector Hardware Manager. Since the application is not registered the first time, it is automatically created by the example. Then, the application channels CAN 1 and CAN 2 must be manually assigned to a physical Vector CAN interface, such as the VN1630A. The two channels must also be physically connected, e. g., via a CANcable 2Y and a CANcable 1.

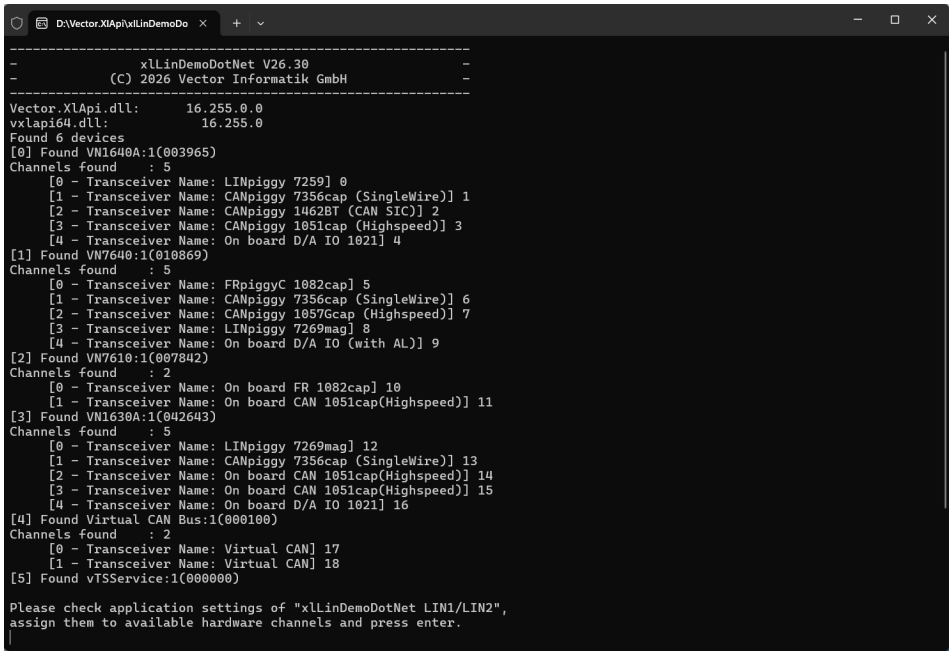
Send and receive messages

By pressing the **[ENTER]** key, the example sends and receives CAN FD messages. The message is sent over the first configured channel and is received by the second one.

15.4.3 xlLinDemo Vector.XlApi

Description

This example demonstrates how to access a Vector LIN interface.



Starting the example

When the example starts, it looks for the **xlLinDemoDotNet** application in the Vector Hardware Manager. Since the application is not registered the first time, it is automatically created by the example. Then, the application channels LIN 1 and LIN 2 must be manually assigned to a physical Vector LIN interface, such as the VN1630A. Both channels must also be physically connected, e. g., with a CANcable0.

Send and receive messages

By pressing the **[ENTER]** key, the example sends and receives LIN messages. The message is sent over the first configured channel and is received by the second one.

15.4.4 xlLinDemo Single Vector.XlApi

Description

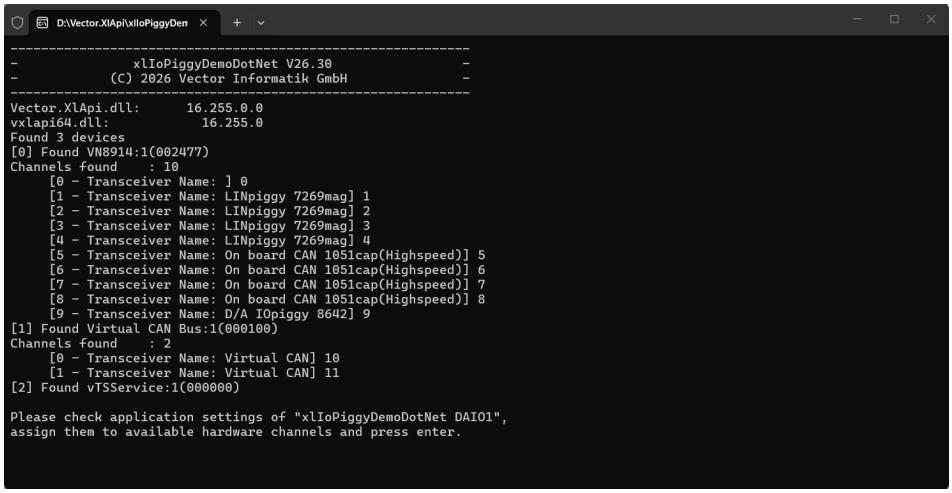
This example is similar to xlLinDemo Vector.XlApi on page 343, but uses only one LIN channel.

15.4.5 xlIoPiggyDemo Vector.XlApi

15.4.5.1 General Information

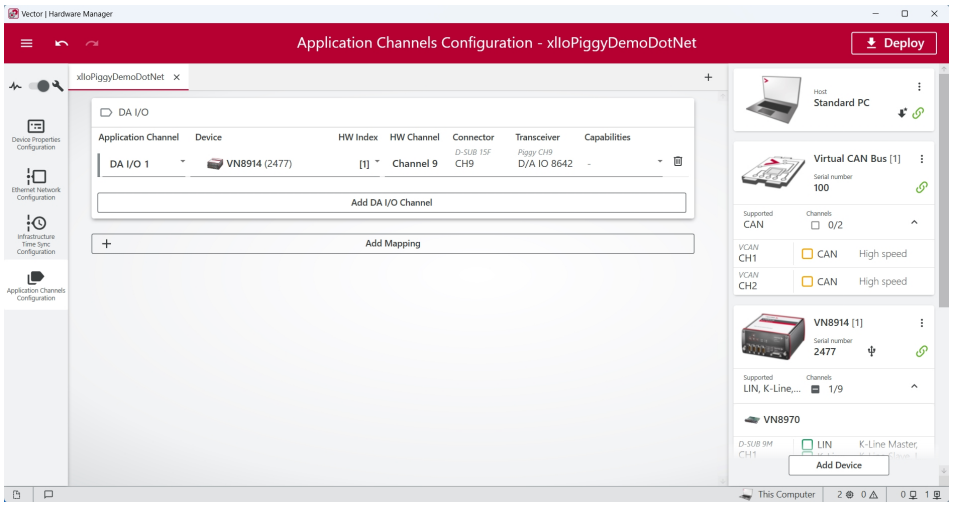
Description

This example demonstrates how to access the IOpiggy 8642 for analog measurements.



Starting the example

When the example starts, it looks for the **xlIoPiggyDemoDotNet** application in the Vector Hardware Manager. Since the application is not registered the first time, it is automatically created by the example. Then, the application channel DAIO 1 must be manually assigned to an IOpiggy 8642 (e. g., inserted into a VN8970).



15.4.5.2 Setup

Pin definition

The following pins of the IIOpiggy 8642 are used in this example:

Signal	Pin	Type
AIO0	14	Analog output
AIO1	7	Analog input
AIO2	15	Analog input
AIO3	8	Analog input
DIO0	1	Digital output
DIO1	9	Digital input
DIO2	2	Digital input
DIO3	10	Digital input
DIO4	3	MOS switch
DIO5	11	MOS switch

Analog measurement

By pressing the **[ENTER]** key, the example toggles the analog output level at AIO0 which can be measured at AIO1…A3. The output level at AIO0 cannot be read back at the same time and remains at 0 mV.

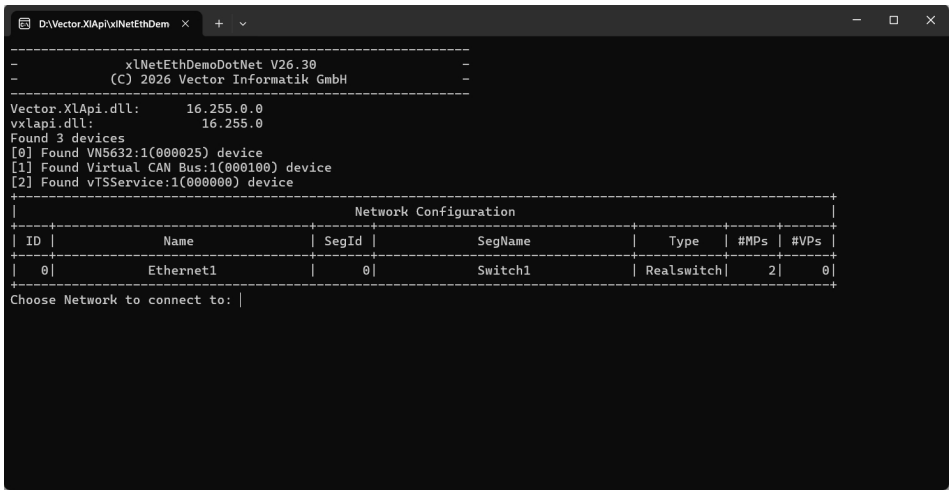
Digital measurement

By pressing the **[ENTER]** key once, DIO0 outputs 5 V. Afterwards, pressing **[ENTER]** toggles the switch between DIO4 and DIO5.

15.4.6 xlNetEthDemo Vector.XlApi

Description

This C# application demonstrates how to transmit and receive Ethernet packets using network-based mode. It is functionally equivalent to the C++ version described in section xlNetEthDemo on page 159.



15.4.7 xlTimeSyncMonitor .NET

15.4.7.1 General Information

Description

This example is a C# application that demonstrates how to use the time sync commands and is functionally similar to the C++ version `xlVTssDemo`. This example uses a modern C# framework for the user interface.

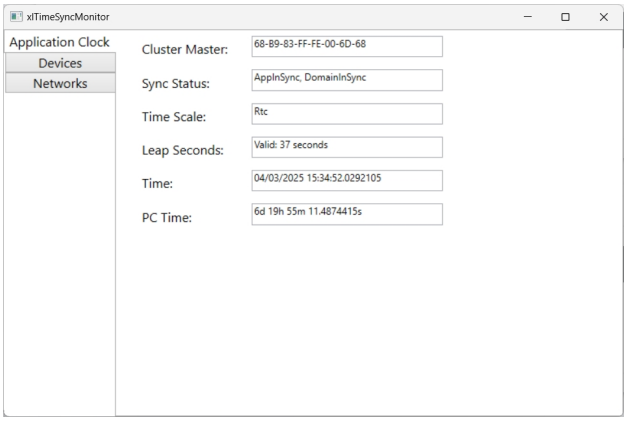


Figure 54: Application clock details

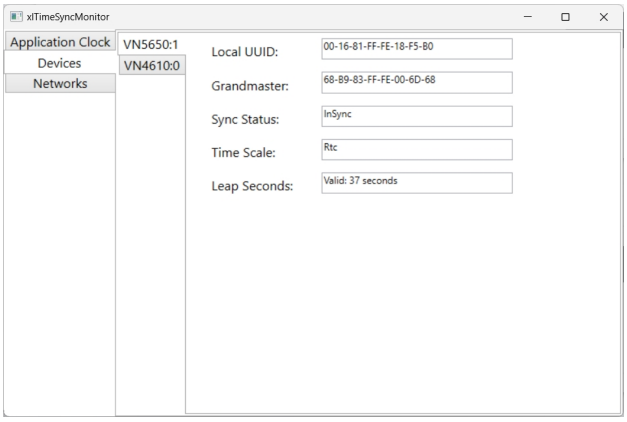


Figure 55: Device details (example)

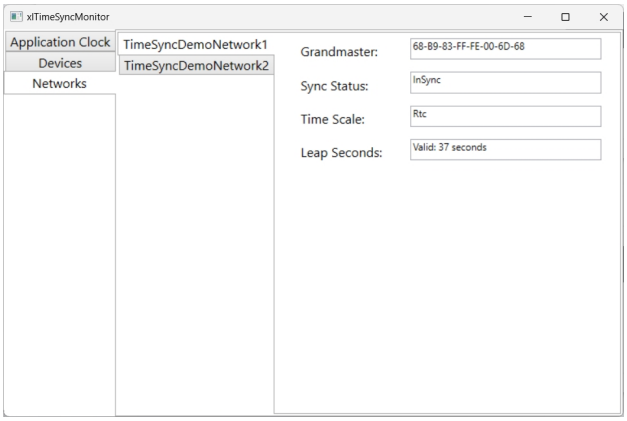


Figure 56: Vector Hardware Manager network details (example)

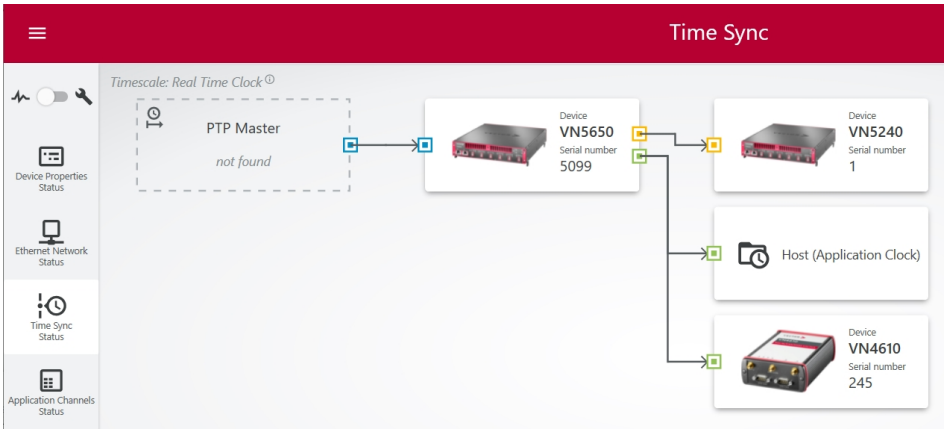


Figure 57: Vector Hardware Manager time sync config (example)

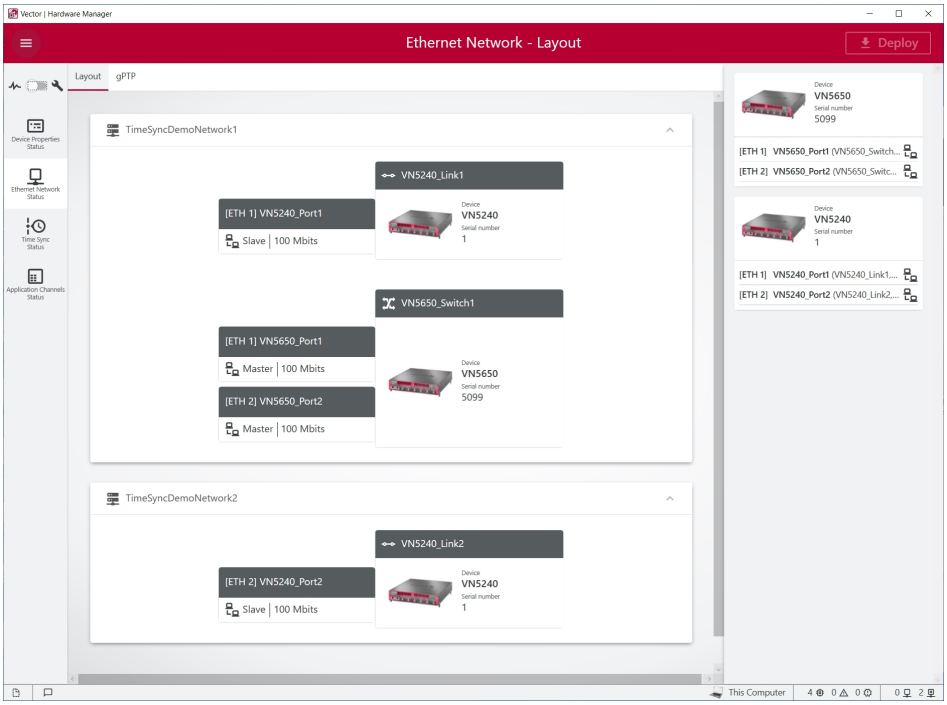
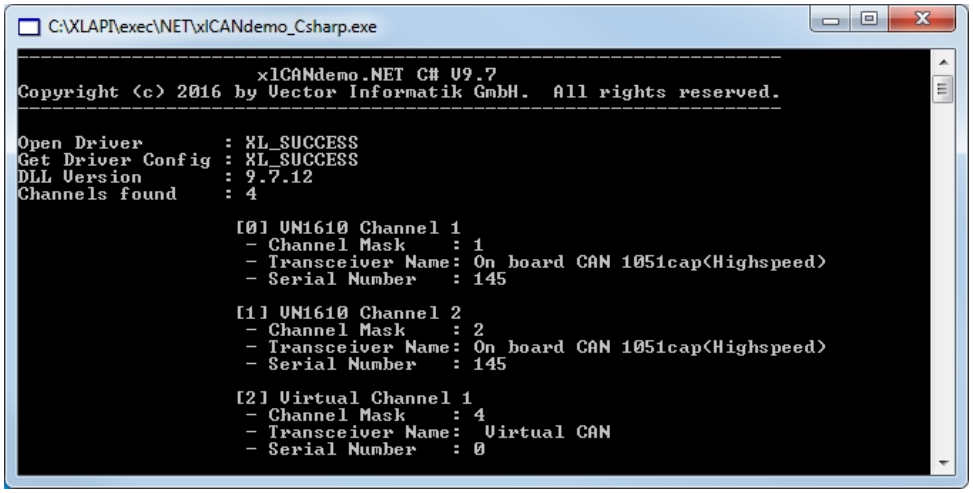


Figure 58: Vector Hardware Manager network config (example)

15.4.8 xlCANdemo .NET

Description

This example shows how to access a Vector CAN interface.



Starting the example

When the example starts, it looks for the application **xlCANdemoNET** in the Vector Hardware Manager. Since this application name is not registered at the very first time, it is automatically created by the example. Afterwards, the application (channels **CAN 1** and **CAN 2**) has to be manually assigned to a real CAN interface such as the CANcardXLe or the VN1630A. Both channels have also to be physically connected, e. g. via CANcable1.

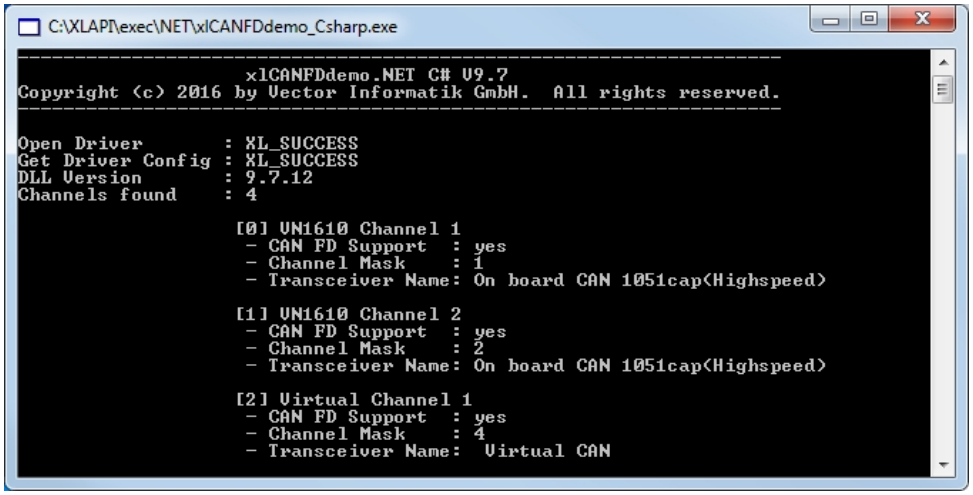
Send and receive messages

By pressing the **[ENTER]** key, the example sends and receives CAN messages. The message is sent over the first configured channel and is received by the second one.

15.4.9 xlCANFDdemo .NET

Description

This example shows how to access the XL API for CAN FD.



Starting the example

When the example starts, it looks for the application **xlCANdemoNET** in the Vector Hardware Manager. Since this application name is not registered at the very first time, it is automatically created by the example. Afterwards, the application (channels **CAN 1** and **CAN 2**) has to be manually assigned to a real CAN interface such as the VN1630A. Both channels have also to be physically connected, e. g. via CANcable 2Y and a CANcable1.

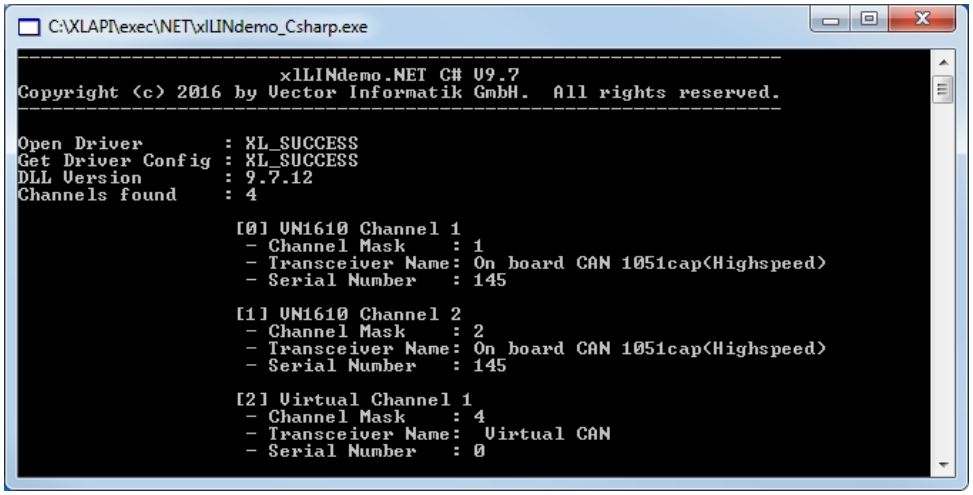
Send and receive messages

By pressing the **[ENTER]** key, the example sends and receives CAN FD messages. The message is sent over the first configured channel and is received by the second one.

15.4.10 xlLINDemo .NET

Description

This example shows how to access a Vector LIN interface.



Starting the example

When the example starts, it looks for the application **xlLINdemoNET** in the Vector Hardware Manager. Since this application name is not registered at the very first time, it is automatically created by the example. Afterwards, the application (channels **LIN 1** and **LIN 2**) has to be manually assigned to a real LIN interface such as the VN1630A. Both channels have also to be physically connected, e. g. with a CANcable0.

Send and receive messages

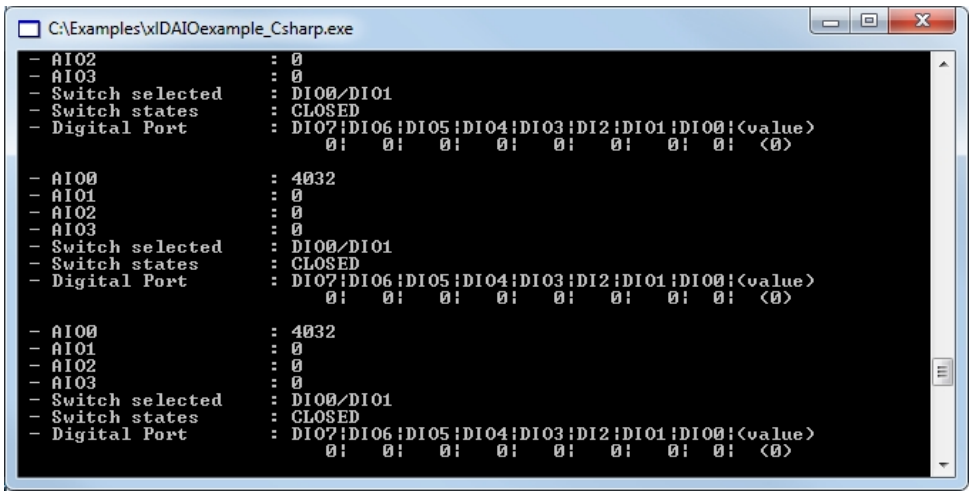
By pressing the **[ENTER]** key, the example sends and receives LIN messages. The message is sent over the first configured channel and is received by the second one.

15.4.11 xlDAIOexample .NET

15.4.11.1 General Information

Description

This example demonstrates how to access a IOcab 8444opto for cyclical measurement.



Starting the example

When the example starts, it looks for the application **xlDAIOexampleNET** in the Vector Hardware Manager. Since this application name is not registered at the very first time, it is automatically created by the example. Afterwards, the application (channel **DAIO 1**) has to be manually assigned to a real DAIO interface such as the CANcardXLe with IOcab 8444opto.

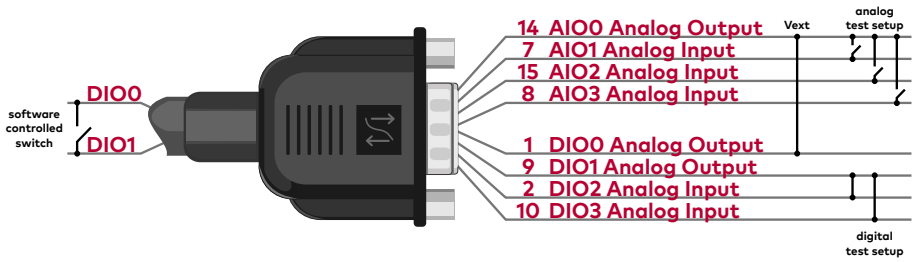
15.4.11.2 Setup

Pin definition

The following pins of the IOcab 8444opto are used in this example:

Signal	Pin	Type
AIO0	14	Analog output
AIO1	7	Analog input
AIO2	15	Analog input
AIO3	8	Analog input
DIO0	1	Digital output (shared electronic switch with DIO1).
DIO1	9	Digital output (supplied by DIO0, when switch is closed).
DIO2	2	Digital input.
DIO3	10	Digital input.

Setup



Note
The internal switch between DIO0 (supplied by AIO0) and DIO1 is closed/opened with `xlDAIOSetDigitalOutput()`. If the switch is closed, the applied voltage at DIO0 can be measured at DIO1.

15.4.11.3 Keyboard commands

The running application can be controlled via the following keyboard commands:

Key	Command
<ENTER>	Toggle digital output.
<x>	Closes application.

15.4.11.4 Output Examples

Example

```
AIO0:             4032mV
AIO1:             0mV
AIO2:             0mV
AIO3:             0mV
Switch selected: DIO0/DI01
Switch states: OPEN
Digital Port:    DIO7 DIO6 DIO5 DIO4 DIO3 DIO2 DIO1 DIO0 val
                    0    0    0    0    0    0    0    1 (1)
```

Explanation

- “AIO0” displays 4032mV, since it is set to output with maximum output level.
- "AIO1” displays 0mV, since there is no applied voltage at this input.
- “AIO2” displays 0mV, since there is no applied voltage at this input.
- “AIO3” displays 0mV, since there is no applied voltage at this input.
- ”Switch selected” displays DIO0/DIO1 (first switch)
- ”Switch states” displays the state of switch between DIO0/DIO1
- “Digital Port” shows the single states of DIO7…DIO0:
  - DIO0: displays ‘1’ (always ‘1’, due the voltage supply)
  - DIO1: displays ‘0’ (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays ‘0’ (output of DIO1)
  - DIO3: displays ‘0’ (output of DIO1)
  - DIO4: displays ‘0’ (n.c.)
  - DIO5: displays ‘0’ (n.c.)
  - DIO6: displays ‘0’ (n.c.)
  - DIO7: displays ‘0’ (n.c.)

Example

```
AIO0:             4032mV
AIO1:             0mV
AIO2:             4032mV
AIO3:             0mV
Switch selected: DIO0/DI01
Switch states: CLOSED
Digital Port:    DIO7 DIO6 DIO5 DIO4 DIO3 DIO2 DIO1 DIO0 val
                    0    0    0    0    1    1    1    1 (1)
```

Explanation

- "AIO0" displays 4032mV, since it is set to output with maximum output level.
- "AIO1" displays 0mV, since there is no applied voltage at this input.
- "AIO0" displays 4032mV, since it is connected to AIO0.
- "AIO3" displays 0mV, since there is no applied voltage at this input.
- "Switch selected" displays DIO0/DIO1 (first switch)
- "Switch state" displays the state of switch between DIO0/DIO1
- "Digital Port" shows the single states of DIO7…DIO0:
  - DIO0: displays '1' (always '1', due the voltage supply)
  - DIO1: displays '1' (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays '1' (output of DIO1)
  - DIO3: displays '1' (output of DIO1)
  - DIO4: displays '0' (n.c.)
  - DIO5: displays '0' (n.c.)
  - DIO6: displays '0' (n.c.)
  - DIO7: displays '0' (n.c.)

Note

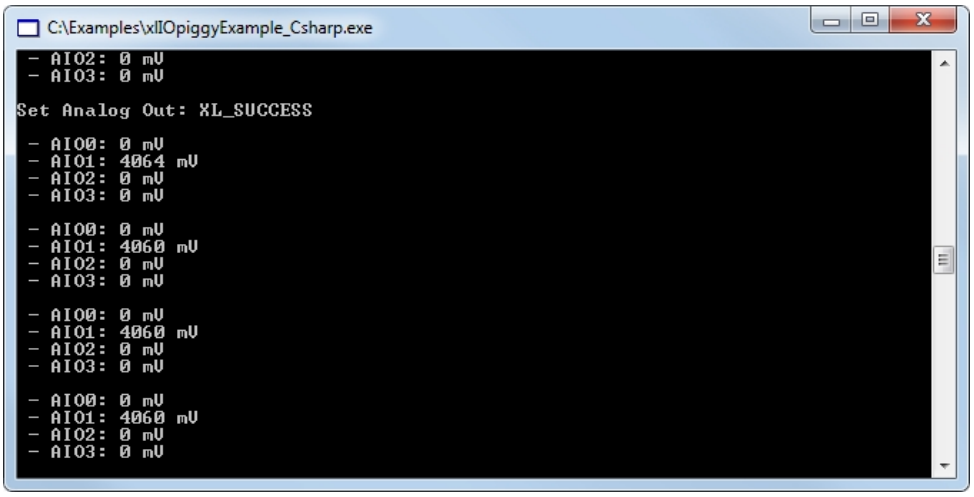
If you try to connect DIO1 (when output is '1') to one of the inputs DIO4…DIO7, you will notice no changes on the screen. The digital output is supplied by the IOcab 8444opto itself, where the maximum output is 4.096 V. Due to different thresholds, the inputs DIO4…DIO7 needs higher voltages (>=4.7 V) to toggle from '0' to '1'.

15.4.12 xlIOpiggyExample .NET

15.4.12.1 General Information

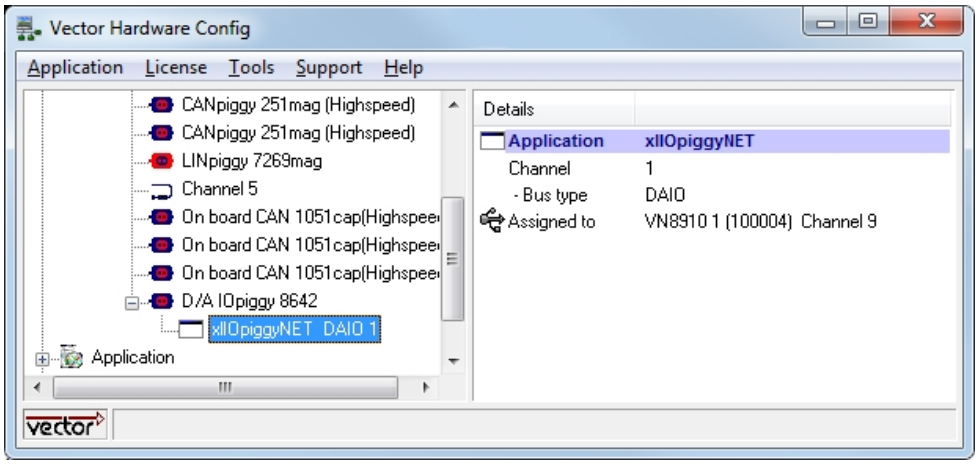
Description

This example shows how to access the IOpiggy 8642 for analog measurement.s



Starting the example

When the example starts, it looks for the application **xlIOpiggyNET** in the Vector Hardware Manager. Since this application name is not registered at the very first time, it is automatically created by the example. Afterwards, the application (channel **DAIO 1**) has to be manually assigned to an IOpiggy 8642 (e. g. inserted on a VN8970).



15.4.12.2 Setup

Pin definition

The following pins of the IIOpiggy 8642 are used in this example:

Signal	Pin	Type
AIO0	14	Analog output
AIO1	7	Analog input
AIO2	15	Analog input
AIO3	8	Analog input
DIO0	1	Digital output
DIO1	9	Digital input
DIO2	2	Digital input
DIO3	10	Digital input
DIO4	3	MOS switch
DIO5	11	MOS switch

Analog measurement

By pressing the **[ENTER]** key, the example toggles the analog output level at AIO0 which can be measured at AIO1…A3. The output level at AIO0 cannot be read back at the same time and remains at 0 mV.

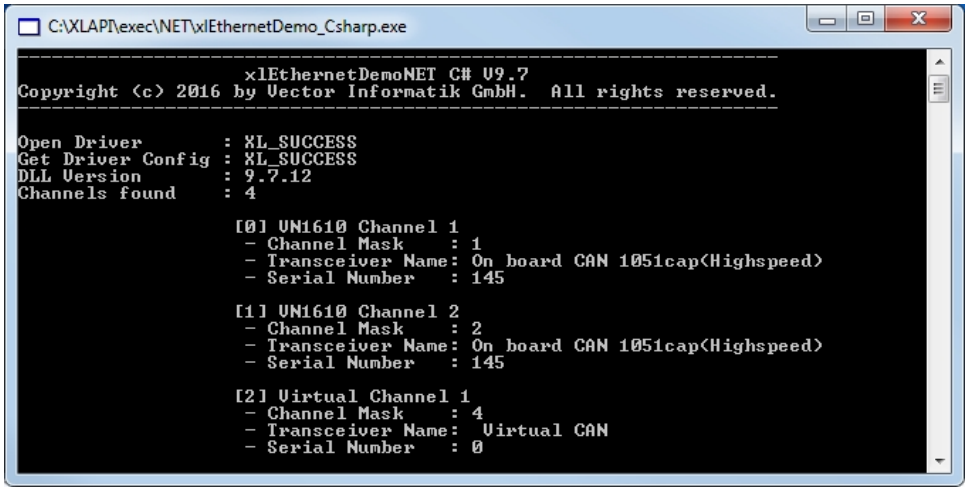
Digital measurement

By pressing the **[ENTER]** key once, DIO0 outputs 5 V. Afterwards, pressing **[ENTER]** toggles the switch between DIO4 and DIO5.

15.4.13 xlEthernetDemo .NET

Description

This example shows how to access the XL API for Ethernet.



Starting the example

When the example starts, it looks for the application **xlEthernetDemoNET** in the Vector Hardware Manager. Since this application name is not registered at the very first time, it is automatically created by the example. Afterwards, the application (channels **Ethernet 1** and **Ethernet 2**) has to be manually assigned to a real Ethernet interface such as the VN5610. Both channels have also to be physically connected via an Ethernet cable (RJ45).

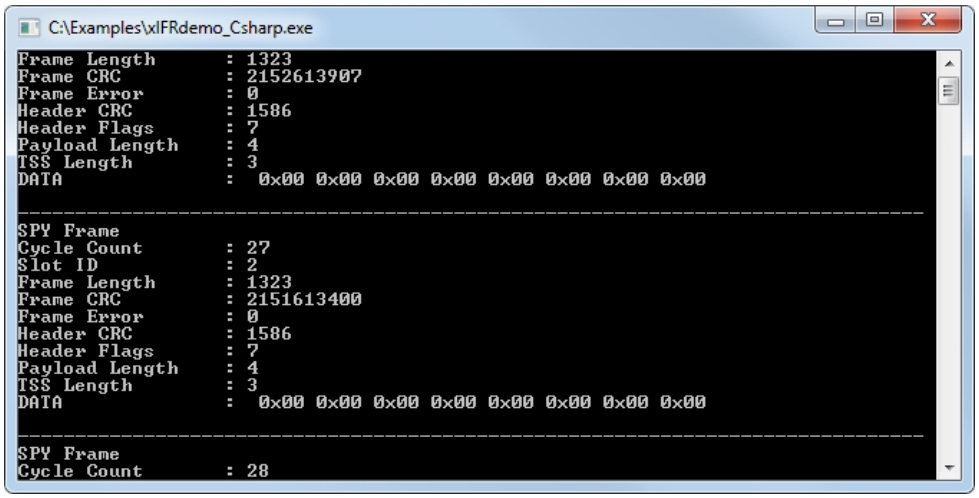
Send and receive messages

By pressing the **[ENTER]** key, the example sends and receives Ethernet packets. The message is sent over the first configured channel and is received by the second one.

15.4.14 xlFRdemo .NET

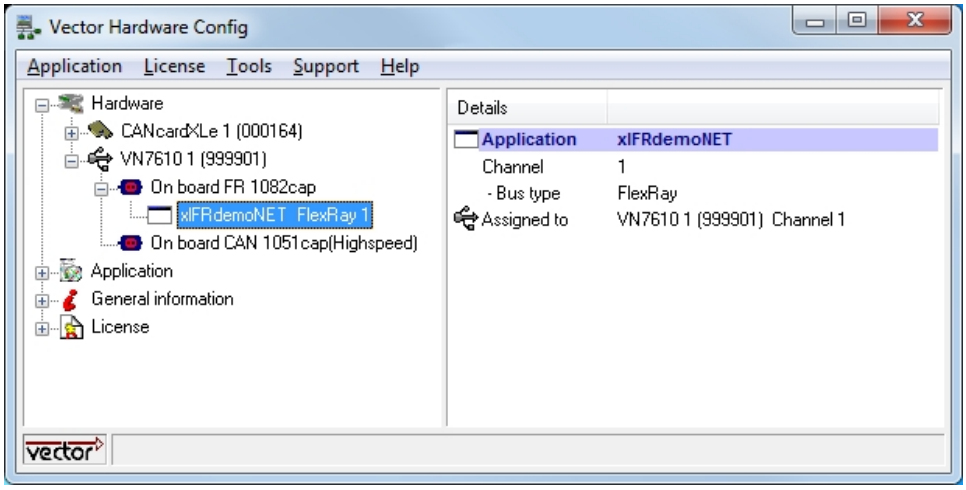
Description

This example shows how to access FlexRay interface (e. g. VN7610) for COLD CC.



Starting the example

When the example starts, it looks for the application **xlFRdemoNET** in the Vector Hardware Manager. Since this application name is not registered at the very first time, it is automatically created by the example. Afterwards, the application (channel **FlexRay 1**) has to be manually assigned to a real FlexRay interface such as the VN7610.



Send and receive frames

By pressing a key, the example sends and receives frames.

16 Error Codes

In this chapter you find the following information:

16.1 Common Error Codes **361**

16.2 Time Sync Error Codes **363**

16.3 CAN FD Error Codes **364**

16.4 FlexRay Error Codes **365**

16.5 Ethernet Error Codes **366**

16.1 Common Error Codes

Code (dec)	Error	Description
0	XL_SUCCESS	The driver call was successful.
1	XL_PENDING	The driver call returned but the requested operation is still pending.
10	XL_ERR_QUEUE_IS_EMPTY	The receive queue of the port is empty. The user can proceed normally
11	XL_ERR_QUEUE_IS_FULL	The transmit queue of a channel is full. The transmit event will be lost.
12	XL_ERR_TX_NOT_POSSIBLE	The hardware is busy and not able to transmit an event at once
14	XL_ERR_NO_LICENSE	The user requested an operation that requires a license to unlock (see section License Management on page 27).
101	XL_ERR_WRONG_PARAMETER	At least one parameter passed to the driver was wrong or invalid.
110	XL_ERR_TWICE_REGISTER	The user attempted to register something that is already registered or otherwise in use.
111	XL_ERR_INVALID_CHAN_INDEX	The driver attempted to access a channel with an invalid index.
112	XL_ERR_INVALID_ACCESS	The user made a call to a port specifying channel(s) for which he had not declared access at opening of the port.
113	XL_ERR_PORT_IS_OFFLINE	The user called a port function whose execution must be online, but the port is offline.
116	XL_ERR_CHAN_IS_ONLINE	The user called a function whose desired channels must be offline, but at least one channel is online.
117	XL_ERR_NOT_IMPLEMENTED	The user called a feature which is not implemented.
118	XL_ERR_INVALID_PORT	The driver attempted to access a port by an invalid pointer or index.
120	XL_ERR_HW_NOT_READY	The accessed hardware is not ready.
121	XL_ERR_CMD_TIMEOUT	The timeout condition occurred while waiting for the response event of a command.
122	XL_ERR_CMD_HANDLING	An internal error concerning command execution occured.
129	XL_ERR_HW_NOT_PRESENT	The hardware is not present (or could not be found) at a channel. This may occur with removable hardware or faulty hardware.
131	XL_ERR_NOTIFY_ALREADY_ACTIVE	This error code exists for historic reasons.
132	XL_ERR_INVALID_TAG	The driver refuses a tag in an event structure.
133	XL_ERR_INVALID_RESERVED_FLD	The user passed a structure that has reserved bits set to one.
134	XL_ERR_INVALID_SIZE .	The driver refuses a size value in an event structure
135	XL_ERR_INSUFFICIENT_BUFFER	The user passed a buffer that is too small to receive all data.
136	XL_ERR_ERROR_CRC	A checksum calculation failed.
137	XL_ERR_BAD_EXE_FORMAT	The OS failed to load an executable, e.g. a DLL is compiled for a different architecure.
138	XL_ERR_NO_SYSTEM_RESOURCES	The OS has insufficient resources (e.g.

Code (dec)	Error	Description
		memory) to perform the requested operation.
139	XL_ERR_NOT_FOUND	The driver could not find the specified object.
140	XL_ERR_INVALID_ACCESS	The driver does not permit the specifed access, e.g. wrong index, wrong interface version or insufficient privileges.
141	XL_ERR_REQ_NOT_ACCEP	The driver refused the operation for an unspecified reason.
142	XL_ERR_INVALID_LEVEL	An internal error concering queue levels occured.
143	XL_ERR_NO_DATA_DETECTED	The driver expects data that is not available.
144	XL_ERR_INTERNAL_ERROR	An unspecified internal error occured.
145	XL_ERR_UNEXP_NET_ERROR	An unexpected networking error occured.
146	XL_ERR_INVALID_USER_BUFFER	The driver refused a buffer provided by user-space.
147	XL_ERR_INVALID_PORT_ACCESS_TYPE	See XL_ERR_INVALID_ACCESS.
152	XL_ERR_NO_RESOURCES	The driver has insufficient resources (e.g. memory) to perform the requested operation.
153	XL_ERR_WRONG_CHIP_TYPE	The relevant chip on the device does not support the requested operation.
154	XL_ERR_WRONG_COMMAND	The user issued a command that was rejected.
155	XL_ERR_INVALID_HANDLE	The user passed an invalid handle.
157	XL_ERR_RESERVED_NOT_ZERO	See XL_ERR_INVALID_RESERVED_FLD.
158	XL_ERR_INIT_ACCESS_MISSING	Function call requires init access.
160	XL_ERR_WRONG_VERSION	A version mismatch was detected.
201	XL_ERR_CANNOT_OPEN_DRIVER	The attempt to load or open the driver failed. Reason could be the driver file which cannot be found, is already loaded or part of a previously unloaded driver.
202	XL_ERR_WRONG_BUS_TYPE	The user called a function with the wrong bus type. (e. g. try to activate a LIN channel for CAN).
203	XL_ERR_DLL_NOT_FOUND	The XL API dll could not be found.
204	XL_ERR_INVALID_CHANNEL_MASK	Invalid channel mask.
205	XL_ERR_NOT_SUPPORTED	Function call not supported.
210	XL_ERR_CONNECTION_BROKEN	The driver lost connection to a remote device.
211	XL_ERR_CONNECTION_CLOSED	The connection is already closed.
212	XL_ERR_INVALID_STREAM_NAME	An internal error concerning remote devices occured.
213	XL_ERR_CONNECTION_FAILED	An internal error concerning remote devices occured.
214	XL_ERR_STREAM_NOT_FOUND	An internal error concerning remote devices occured.
215	XL_ERR_STREAM_NOT_CONNECTED	An internal error concerning remote devices occured.
216	XL_ERR_QUEUE_OVERRUN	A queue overrun occured.
255	XL_ERROR	An unspecified error occurred.

16.2 Time Sync Error Codes

Code (dec)	Error	Description
1282	XL_ERR_TS_DOMAIN_NOT_SYNC	The port is not in synchronous state.
1283	XL_ERR_TS_INVALID_CLUSTER_MASTER	The cluster master of at least one measurement clock is undefined.
1284	XL_ERR_TS_CLOCK_NOT_FOUND	There were no clocks found that can be reset.
1285	XL_ERR_TS_AGGREGATE_STATUS	At least the time scale of one measurement clock is undefined.
1292	XL_ERR_TS_RESET_CLOCK	At least one clock could not be reset or its `timeOffset` could not be read.
1293	XL_ERR_TS_SET_CLOCK_OFFSET	The `timeOffset` for at least one clock could not be set.
1296	XL_ERR_TS_SYNC_TO_LOCAL	GNSS is being used, but the cluster master is not receiving a satellite signal.
1297	XL_ERR_TS_SYNC_OFF	All synchronization protocols are disabled.
1298	XL_ERR_TS_CLOCK_NOT_SYNC	The clock is not synchronized with the cluster master. Used when querying the status of a single clock.

16.3 CAN FD Error Codes

Code (hex)	Error	Description
0x0201	XL_ERR_INVALID_DLC	DLC with invalid value.
0x0202	XL_ERR_INVALID_CANID	CAN Id has invalid bits set.
0x0203	XL_ERR_INVALID_FDFLAG_MODE20	Flag set that must not be set when configured for CAN20 (e.g. EDL).
0x0204	XL_ERR_EDL_RTR	RTR must not be set in combination with EDL.
0x0205	XL_ERR_EDL_NOT_SET	EDL is not set but BRS and/or ESICTRL is.
0x0206	XL_ERR_UNKNOWN_FLAG	Unknown bit in flags field is set.

16.4 FlexRay Error Codes

Code (hex)	Error	Description
0x0104	XL_ERR_PDU_OUT_OF_MEMORY	Too many PDUs configured or too less system memory free.
0x0105	XL_ERR_FR_CLUSTERCONFIG_MISSING	No cluster configuration has been sent to the driver but is needed for the command which failed.
0x0106	XL_ERR_PDU_OFFSET_REPET_INVALID	Invalid offset and/or repetition value specified.
0x0107	XL_ERR_PDU_PAYLOAD_SIZE_INVALID	Specified PDU payload size is invalid (e.g. size is too large) Frame-API: size is different than static payload length configured in cluster config.
0x0109	XL_ERR_FR_NBR_FRAMES_OVERFLOW	Too many frames specified in parameter.
0x010B	XL_ERR_FR_SLOT_ID_INVALID	Specified slot-ID exceeds biggest possible ID specified by the cluster configuration.
0x010C	XL_ERR_FR_SLOT_ALREADY_OCCUPIED_BY_ERAY	Specified slot cannot be used by Coldstart-Controller because it's already in use by the eRay.
0x010D	XL_ERR_FR_SLOT_ALREADY_OCCUPIED_BY_ COLDC	Specified slot cannot be used by eRay because it's already in use by the Coldstart-Controller.
0x010E	XL_ERR_FR_SLOT_OCCUPIED_BY_OTHER_APP	Specified slot cannot be used because it's already in use by another application.
0x010F	XL_ERR_FR_SLOT_IN_WRONG_SEGMENT	Specified slot is not in correct segment. E.g.: A dynamic slot was specified for startup&sync.
0x0110	XL_ERR_FR_FRAME_CYCLE_MULTIPLEX_ERROR	The given frame-multiplexing rule (specified by offset and repetition) cannot be done because some of the slots are already in use.
0x0116	XL_ERR_PDU_NO_UNMAP_OF_SYNCFRAME	Unmapping of eRay startup/sync frames is not allowed.
0x0123	XL_ERR_SYNC_FRAME_MODE	Wrong txMode in sync frame.

16.5 Ethernet Error Codes

Code (hex)	Error	Description
0x1100	XL_ERR_ETH_PHY_ACTIVATION_FAILED	The driver failed to activate an Ethernet PHY.
0x1103	XL_ERR_ETH_PHY_CONFIG_ABORTED	The driver aborted the configuration of an Ethernet PHY.
0x1104	XL_ERR_ETH_RESET_FAILED	The driver failed to reset the device.
0x1105	XL_ERR_ETH_SET_CONFIG_DELAYED	Requested config was stored but could not be immediately activated.
0x1106	XL_ERR_ETH_UNSUPPORTED_FEATURE	Requested feature/function not supported by device.
0x1107	XL_ERR_ETH_MAC_ACTIVATION_FAILED	The driver failed to activate the Ethernet MAC .
0x110C	XL_ERR_NET_ETH_SWITCH_IS_ONLINE	Switch has already been activated.

17 Important Notes - Details

In this chapter you find the following information:

17.1 Safety Instructions and Hazard Warnings 368
17.1.1 Proper Use and Intended Purpose 368
17.1.2 Hazards 369
17.2 Disclaimer 370
17.3 Disposal of Vector Hardware 371

17.1 Safety Instructions and Hazard Warnings



WARNING!
In order to avoid personal injuries and damage to property, you have to read and understand the following safety instructions and hazard warnings prior to installation and use of this interface. Keep this documentation (manual) always near the interface.

17.1.1 Proper Use and Intended Purpose



WARNING!
The interface is designed for analyzing, controlling and otherwise influencing control systems and electronic control units. This includes, inter alia, bus systems like CAN, LIN, K-Line, MOST, FlexRay, Ethernet, BroadR-Reach and/or ARINC 429.

The interface may only be operated in a closed state. In particular, printed circuits must not be visible. The interface may only be operated (i) according to the instructions and descriptions of this manual; (ii) with the electric power supply designed for the interface, e.g. USB-powered power supply; and (iii) with accessories manufactured or approved by Vector.

The interface is exclusively designed for use by skilled personnel as its operation may result in serious personal injuries and damage to property. Therefore, only those persons may operate the interface who (i) have understood the possible effects of the actions which may be caused by the interface; (ii) are specifically trained in the handling with the interface, bus systems and the system intended to be influenced; and (iii) have sufficient experience in using the interface safely.

The knowledge necessary for the operation of the interface can be acquired in work-shops and internal or external seminars offered by Vector. Additional and interface specific information, such as „Known Issues“, are available in the „Vector KnowledgeBase“ on Vector´s website at www.vector.com. Please consult the „Vector KnowledgeBase“ for updated information prior to the operation of the interface.

17.1.2 Hazards



WARNING!
The interface may control and/or otherwise influence the behavior of control systems and electronic control units. Serious hazards for life, body and property may arise, in particular, without limitation, by interventions in safety relevant systems (e.g. by deactivating or otherwise manipulating the engine management, steering, airbag and/or braking system) and/or if the interface is operated in public areas (e.g. public traffic, airspace). Therefore, you must always ensure that the interface is used in a safe manner. This includes, inter alia, the ability to put the system in which the interface is used into a safe state at any time (e.g. by „emergency shutdown“), in particular, without limitation, in the event of errors or hazards.

Comply with all safety standards and public regulations which are relevant for the operation of the system. Before you operate the system in public areas, it should be tested on a site which is not accessible to the public and specifically prepared for performing test drives in order to reduce hazards.

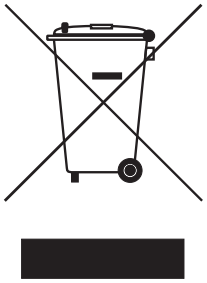
17.2 Disclaimer



WARNING!
Claims based on defects and liability claims against Vector are excluded to the extent damages or errors are caused by improper use of the interface or use not according to its intended purpose. The same applies to damages or errors arising from insufficient training or lack of experience of personnel using the interface.

17.3 Disposal of Vector Hardware

Please handle old devices responsibly and observe the environmental laws applicable in your country. Please dispose of the Vector hardware only at the designated places and not with the household waste.



Within the European Community, the Directive on Waste Electrical and Electronic Equipment (WEEE Directive) and the Directive on the Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment (RoHS Directive) apply.

For Germany and other EU countries, we offer free take-back of old Vector hardware.

Please carefully check the Vector hardware to be disposed of before shipping. Please remove all items that are not part of the original scope of delivery, e.g. storage media. The Vector hardware must also be free of licenses and must no longer contain any personal data. Vector does not perform any checks in this regard.
Once the hardware has been shipped, it cannot be returned to you. By shipping the hardware to us, you have relinquished your rights to the hardware.
Before shipping, please register your old device via:

https://www.vector.com/int/en/support-downloads/return-registration-for-the-disposal-of-vector-hardware/

18 Wichtige Hinweise

In diesem Kapitel finden Sie die folgenden Informationen:

18.1 Sicherheits- und Gefahrenhinweise 373
18.1.1 Sach- und bestimmungsgemäßer Gebrauch 373
18.1.2 Gefahren 374
18.2 Haftungsausschluss 375
18.3 Entsorgung von Vector Hardware 376

18.1 Sicherheits- und Gefahrenhinweise



WARNUNG!
Um Personen- und Sachschäden zu vermeiden, müssen Sie vor der Installation und dem Einsatz dieses Interfaces die nachfolgenden Sicherheits- und Gefahrenhinweise lesen und verstehen. Bewahren Sie diese Dokumentation (Handbuch) stets in der Nähe dieses Interfaces auf.

18.1.1 Sach- und bestimmungsgemäßer Gebrauch



WARNUNG!
Das Interface ist für die Analyse, die Steuerung sowie für die anderweitige Beeinflussung von Regelsystemen und Steuergeräten bestimmt. Das umfasst unter anderem die Bussysteme CAN, LIN, K-Line, MOST, FlexRay, Ethernet, BroadR-Reach oder ARINC 429.

Der Betrieb des Interfaces darf nur im geschlossen Zustand erfolgen. Insbesondere dürfen keine Leiterplatten sichtbar sein. Das Interface ist entsprechend den Anweisungen und Beschreibungen dieses Handbuchs einzusetzen. Dabei darf nur die dafür vorgesehene Stromversorgung, wie z. B. USB-powered, Netzteil, und das Originalzubehör von Vector bzw. das von Vector freigegebene Zubehör verwendet werden.

Das Interface ist ausschließlich für den Einsatz durch geeignetes Personal bestimmt, da der Gebrauch dieses Interfaces zu erheblichen Personen- und Sachschäden führen kann. Deshalb dürfen nur solche Personen dieses Interface einsetzen, welche die möglichen Konsequenzen der Aktionen mit diesem Interface verstanden haben, speziell für den Umgang mit diesem Interface, den Bussystemen und dem zu beeinflussenden System geschult worden sind und ausreichende Erfahrung im sicheren Umgang mit dem Interface erlangt haben.

Die notwendigen Kenntnisse zum Einsatz dieses Interfaces können bei Vector über interne oder externe Seminare und Workshops erworben werden. Darüber hinausgehende und Interface-spezifische Informationen wie z. B. „Known Issues“ sind auf der Vector Webseite unter www.vector.com in der „Vector KnowledgeBase“ verfügbar. Bitte informieren Sie sich dort vor dem Betrieb des Interfaces über aktualisierte Hinweise.

18.1.2 Gefahren



WARNUNG!
Das Interface kann das Verhalten von Regelsystemen und Steuergeräten steuern und in anderweitiger Weise beeinflussen. Insbesondere durch Eingriffe in sicherheitsrelevante Bereiche (z. B. durch Deaktivierung oder sonstige Manipulation der Motorsteuerung, des Lenk-, Airbag-, oder Bremssystems) und/oder der Einsatz des Interfaces in öffentlichen Räumen (z. B. Straßenverkehr, Luftraum) können erhebliche Gefahren für Leib, Leben und Eigentum entstehen. Stellen Sie daher in jedem Fall eine gefahrfreie Verwendung sicher. Hierzu gehört unter anderem auch, dass das System, in dem das Interface eingesetzt wird, jederzeit, insbesondere bei Auftreten von Fehlern oder Gefahren, in einen sicheren Zustand geführt werden kann (z. B. durch Not-Abschaltung).

Beachten Sie alle sicherheitstechnischen Richtlinien und öffentlich-rechtlichen Vorschriften, die für den Einsatz des Systems relevant sind. Zur Verminderung von Gefahren sollte das System vor dem Einsatz in öffentlichen Räumen auf einem nicht-öffentlich zugänglichen und für Testfahrten bestimmten Gelände erprobt werden.

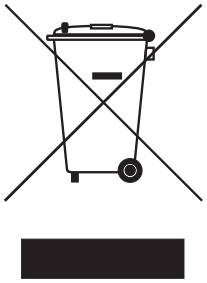
18.2 Haftungsausschluss



WARNUNG!
Soweit das Interface nicht sach- oder bestimmungsgemäß eingesetzt wird, übernimmt Vector keine Gewährleistung oder Haftung für dadurch verursachte Schäden oder Fehler. Das Gleiche gilt für Schäden oder Fehler, die auf einer mangelnden Schulung oder Erfahrung derjenigen Personen beruhen, die das Interface einsetzen.

18.3 Entsorgung von Vector Hardware

Bitte gehen Sie verantwortungsvoll mit Altgeräten um und beachten Sie die in Ihrem Land geltenden Umweltgesetze. Entsorgen Sie die Vector Hardware bitte nur bei den dafür vorgesehenen Stellen und nicht über den Hausmüll.



Innerhalb der Europäischen Gemeinschaft gelten die Richtlinie über Elektro- und Elektronik-Altgeräte (WEEE-Richtlinie) und die Richtlinie zur Beschränkung der Verwendung bestimmter gefährlicher Stoffe in Elektro- und Elektronikgeräten (RoHS-Richtlinie).

Für Deutschland und andere EU-Länder bieten wir Ihnen eine kostenlose Rücknahme der alten Vector Hardware an.

Bitte überprüfen Sie die zu entsorgende Vector Hardware vor dem Versand sorgfältig. Bitte entfernen Sie alle Gegenstände, die nicht zum ursprünglichen Lieferumfang gehören, z. B. Speichermedien. Die Vector Hardware muss außerdem frei von Lizenzen sein und darf keine personenbezogenen Daten mehr enthalten. Vector führt keine Kontrollen diesbezüglich durch. Sobald die Hardware versandt wurde, kann sie nicht mehr an Sie zurück geliefert werden. Mit dem Versand der Hardware an uns haben Sie Ihre Rechte an der Hardware abgetreten. Bitte melden Sie vor dem Versand Ihr Altgerät an über:

https://www.vector.com/de/de/support-downloads/return-registration-for-the-disposal-of-vector-hardware/

VECTOR

Visit our website for:

- News
- Products
- Demo software
- Support
- Training classes
- Addresses