

XL Driver Library Manual

Version 9.7 | 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 2016, Vector Informatik GmbH. All rights reserved.

Contents

1 Introduction	18
1.1 About this User Manual	19
1.1.1 Certification	19
1.1.2 Warranty	20
1.1.3 Registered Trademarks	20
2 Overview	21
2.1 General Information	22
2.2 Principles of the XL Driver Library	23
2.2.1 General Information	23
2.2.2 Step 1: Driver Initialization	24
2.2.3 Step 2: Channel Setup	26
2.2.4 Step 3: On Bus/Measurement Tasks	26
2.3 Driver Files and Examples	27
2.4 System Requirements	28
2.5 Additional Information	29
3 Common Commands	30
3.1 Introduction	31
3.2 Functions	32
3.2.1 xlOpenDriver	32
3.2.2 xlCloseDriver	32
3.2.3 xlGetApplConfig	32
3.2.4 xlSetApplConfig	34
3.2.5 xlGetDriverConfig	34
3.2.6 xlGetRemoteDriverConfig	35
3.2.7 xlGetChannelIndex	36
3.2.8 xlGetChannelMask	36
3.2.9 xlOpenPort	37
3.2.10 xlClosePort	39
3.2.11 xlSetTimerRate	39
3.2.12 xlSetTimerRateAndChannel	40
3.2.13 xlResetClock	41
3.2.14 xlSetNotification	41
3.2.15 xlFlushReceiveQueue	41
3.2.16 xlGetReceiveQueueLevel	42
3.2.17 xlActivateChannel	42
3.2.18 xlReceive	43

3.2.19 xlGetString	44
3.2.20 xlGetErrorString	45
3.2.21 xlGetSyncTime	45
3.2.22 xlGetChannelTime	46
3.2.23 xlGenerateSyncPulse	46
3.2.24 xlPopupHwConfig	47
3.2.25 xlDeactivateChannel	47
3.2.26 xlGetLicenseInfo	47
3.2.27 xlSetGlobalTimeSync	48
3.2.28 xlGetKeymanBoxes	48
3.2.29 xlGetKeymanInfo	48
3.3 Structs	50
3.3.1 XLdriverConfig	50
3.3.2 XLchannelConfig	50
3.3.3 XLbusParams	53
3.3.4 XLlicenseInfo	57
3.4 Events	59
3.4.1 XLevent	59
3.4.2 XL Tag Data	60
3.4.3 XL Sync Pulse	60
3.4.4 XL Transceiver	61
3.4.5 XL Timer	61
4 CAN Commands	62
4.1 Introduction	63
4.2 Flowchart	64
4.3 Functions	65
4.3.1 xlCanSetChannelMode	65
4.3.2 xlCanSetChannelOutput	65
4.3.3 xlCanSetReceiveMode	66
4.3.4 xlCanSetChannelTransceiver	66
4.3.5 xlCanSetChannelParams	68
4.3.6 xlCanSetChannelParamsC200	69
4.3.7 xlCanSetChannelBitrate	69
4.3.8 xlCanSetChannelAcceptance	70
4.3.9 xlCanAddAcceptanceRange	71
4.3.10 xlCanRemoveAcceptanceRange	72
4.3.11 xlCanResetAcceptance	73
4.3.12 xlCanRequestChipState	74
4.3.13 xlCanTransmit	74

4.3.14 xlCanFlushTransmitQueue	75
4.4 Structs	76
4.4.1 XLchipParams	76
4.5 Events	77
4.5.1 XL CAN Message	77
4.5.2 XL Chip State	78
4.6 Application Examples	80
4.6.1 xICANDemo	80
4.6.1.1 General Information	80
4.6.1.2 Keyboard Commands	80
4.6.1.3 Functions	81
4.6.2 xICANcontrol	82
4.6.2.1 General Information	82
4.6.2.2 Classes	83
4.6.2.3 Functions	83
5 CAN FD Commands	85
5.1 Introduction	86
5.2 Flowchart	87
5.3 Functions	88
5.3.1 xlCanFdSetConfiguration	88
5.3.2 xlCanTransmitEx	88
5.3.3 xlCanReceive	89
5.3.4 xlCanGetEventString	89
5.4 Structs	90
5.4.1 XLcanFdConf	90
5.5 Events	91
5.5.1 XLcanTxEvent	91
5.5.2 XL_CAN_TX_MSG	91
5.5.3 XLcanRxEvent	92
5.5.4 XL_CAN_EV_RX_MSG	93
5.5.5 XL_CAN_EV_ERROR	94
5.5.6 XL_CAN_EV_CHIP_STATE	95
5.5.7 XL_CAN_EV_TX_REQUEST	95
5.5.8 XL_SYNC_PULSE_EV	96
6 LIN Commands	98
6.1 Introduction	99
6.2 Flowchart	100
6.3 LIN Basics	101

6.4 Functions	102
6.4.1 xILinSetChannelParams	102
6.4.2 xILinSetDLC	102
6.4.3 xILinSetChecksum	103
6.4.4 xILinSetSlave	104
6.4.5 xILinSwitchSlave	105
6.4.6 xILinSendRequest	106
6.4.7 xILinWakeUp	106
6.4.8 xILinSetSleepMode	106
6.5 Structs	108
6.5.1 XLinStatPar	108
6.6 Events	109
6.6.1 XL LIN Message API	109
6.6.2 XL LIN Message	109
6.6.3 XL LIN Error Message	110
6.6.4 XL LIN Sync Error	110
6.6.5 LIN No Answer	110
6.6.6 LIN Wake Up	110
6.6.7 LIN Sleep	111
6.6.8 LIN CRC Info	111
6.7 Application Examples	112
6.7.1 xILINExample	112
6.7.1.1 General Information	112
6.7.1.2 Classes	112
6.7.1.3 Functions	113
7 D/A IO Commands (IOcab)	114
7.1 Introduction	115
7.2 Flowchart	116
7.3 Functions	117
7.3.1 xIDAIOSetAnalogParameters	117
7.3.2 xIDAIOSetAnalogOutput	118
7.3.3 xIDAIOSetAnalogTrigger	119
7.3.4 xIDAIOSetDigitalParameters	119
7.3.5 xIDAIOSetDigitalOutput	120
7.3.6 xIDAIOSetPWMOutput	121
7.3.7 xIDAIOSetMeasurementFrequency	122
7.3.8 xIDAIORRequestMeasurement	122
7.4 Events	124
7.4.1 XL DAIO Data	124

7.5 Application Examples	126
7.5.1 xIDAIExample	126
7.5.1.1 General Information	126
7.5.1.2 Setup	126
7.5.1.3 Keyboard commands	127
7.5.1.4 Output Examples	127
7.5.1.5 Functions	128
7.5.2 xIDAOdemo	128
7.5.2.1 General Information	128
7.5.2.2 Classes	129
8 D/A IO Commands (IOpiggy)	130
8.1 Introduction	131
8.2 Flowchart	132
8.3 Functions	133
8.3.1 xlIoSetTriggerMode (IOpiggy)	133
8.3.2 xlIoConfigurePorts	133
8.3.3 xlIoSetDigInThreshold	134
8.3.4 xlIoSetDigOutLevel	134
8.3.5 xlIoSetDigitalOutput	135
8.3.6 xlIoSetAnalogOutput	135
8.3.7 xlIoStartSampling	135
8.4 Structs	137
8.4.1 XLdaioTriggerMode	137
8.4.2 XLdaioDigitalParams	138
8.4.3 XLdaioDigitalParams (IOpiggy)	140
8.4.4 XLdaioAnalogParams	140
8.5 Events	142
8.5.1 XL DAIO Piggy Data	142
8.5.2 XL IO Analog Data	142
8.5.3 XL IO Digital Data	143
9 D/A IO Commands (VN1600)	144
9.1 Introduction	145
9.2 Flowchart	146
9.3 Functions	147
9.3.1 xlIoSetTriggerMode (VN1600)	147
9.3.2 xlIoSetDigitalOutput	147
9.4 Structs	149
9.4.1 XLdaioDigitalParams (VN1600)	149
9.5 Events	150

9.5.1 XL DAIO Piggy Data	150
9.5.2 XL IO Analog Data	150
9.5.3 XL IO Digital Data	151
10 MOST Commands	152
10.1 Introduction	153
10.2 Flowchart	154
10.3 MOST Analysis Library and Node Functions	156
10.4 Specific OS8104 Registers	159
10.5 Functions	160
10.5.1 xlMostSwitchEventSources	160
10.5.2 xlMostSetAllBypass	161
10.5.3 xlMostGetAllBypass	162
10.5.4 xlMostSetTimingMode	162
10.5.5 xlMostGetTimingMode	163
10.5.6 xlMostSetFrequency	164
10.5.7 xlMostGetFrequency	164
10.5.8 xlMostWriteRegister	165
10.5.9 xlMostReadRegister	166
10.5.10 xlMostWriteRegisterBit	166
10.5.11 xlMostCtrlTransmit	167
10.5.12 xlMostAsyncTransmit	168
10.5.13 xlMostSyncGetAllocTable	168
10.5.14 xlMostCtrlSyncAudio	169
10.5.15 xlMostCtrlSyncAudioEx	170
10.5.16 xlMostSyncVolume	171
10.5.17 xlMostSyncGetVolumeStatus	172
10.5.18 xlMostSyncMute	172
10.5.19 xlMostSyncGetMuteStatus	173
10.5.20 xlMostGetRxLight	174
10.5.21 xlMostSetTxLight	174
10.5.22 xlMostGetTxLight	175
10.5.23 xlMostSetLightPower	175
10.5.24 xlMostGetLockStatus	176
10.5.25 xlMostGenerateLightError	176
10.5.26 xlMostGenerateLockError	177
10.5.27 xlMostCtrlRxBuffer	178
10.5.28 xlMostCtrlConfigureBusload	179
10.5.29 xlMostCtrlGenerateBusload	179
10.5.30 xlMostAsyncConfigureBusload	180

10.5.31 xlMostAsyncGenerateBusload	181
10.5.32 xlMostReceive	181
10.5.33 xlMostTwinklePowerLed	182
10.5.34 Streaming	183
10.5.34.1 General Information	183
10.5.34.2 Frame Format	184
10.5.35 xlMostStreamOpen	185
10.5.36 xlMostStreamClose	186
10.5.37 xlMostStreamStart	186
10.5.38 xlMostStreamStop	187
10.5.39 xlMostStreamBufferAllocate	187
10.5.40 xlMostStreamBufferDeallocateAll	188
10.5.41 xlMostStreamBufferSetNext	189
10.5.42 xlMostStreamClearBuffers	189
10.5.43 xlMostStreamGetInfo	190
10.6 Structs	192
10.6.1 s_xl_most_async_busload_configuration	192
10.6.2 s_xl_most_ctrl_busload_configuration	192
10.6.3 XL_MOST_STREAM_OPEN	193
10.7 Events	195
10.7.1 s_xl_event_most	195
10.7.2 s_xl_most_tag_data	196
10.7.3 XL_MOST_START	197
10.7.4 XL_MOST_STOP	198
10.7.5 XL_MOST_EVENT_SOURCE_EV	198
10.7.6 XL_MOST_ALLBYPASS_EV	198
10.7.7 XL_MOST_TIMING_MODE_EV	198
10.7.8 XL_MOST_TIMING_MODE_SPDIF_EV	199
10.7.9 XL_MOST_FREQUENCY_EV	199
10.7.10 XL_MOST_REGISTER_BYTES	200
10.7.11 XL_MOST_REGISTER_BITS_EV	200
10.7.12 XL_MOST_SPECIAL_REGISTER_EV	200
10.7.13 XL_MOST_CTRL_SPY_EV	202
10.7.14 XL_MOST_CTRL_MSG_EV	203
10.7.15 XL_MOST_CTRL_TX	205
10.7.16 XL_MOST_ASYNC_MSG_EV	205
10.7.17 XL_MOST_ASYNC_TX_EV	205
10.7.18 XL_MOST_SYNC_ALLOC_EV	206
10.7.19 XL_MOST_SYNC_VOLUME_STATUS_EV	206
10.7.20 XL_MOST_RX_LIGHT_EV	207
10.7.21 XL_MOST_TX_LIGHT_EV	207
10.7.22 XL_MOST_LOCK_STATUS_EV	207

10.7.23 XL_MOST_ERROR_EV	208
10.7.24 XL_MOST_RX_BUFFER_EV	208
10.7.25 XL_MOST_CTRL_SYNC_AUDIO_EV	209
10.7.26 XL_MOST_CTRL_SYNC_AUDIO_EX	209
10.7.27 XL_MOST_SYNC_MUTES_STATUS_EV	210
10.7.28 XL_MOST_LIGHT_POWER_EV	210
10.7.29 XL_MOST_GEN_LIGHT_ERROR_EV	210
10.7.30 XL_MOST_GEN_LOCK_ERROR_EV	211
10.7.31 XL_MOST_CTRL_BUSLOAD_EV	212
10.7.32 XL_MOST_ASYNC_BUSLOAD_EV	212
10.7.33 XL_MOST_STREAM_BUFFER	212
10.7.34 XL_MOST_STREAM_STATE_EV	213
10.7.35 XL_MOST_SYNC_TX_UNDERFLOW_EV	214
10.7.36 XL_MOST_SYNC_RX_OVERFLOW_EV	214
10.8 Application Examples	215
10.8.1 xIMOSTView	215
10.8.1.1 General Information	215
10.8.1.2 Classes	216
10.8.1.3 Functions	216

11 MOST 150 Commands 219

11.1 Introduction	220
11.2 Flowchart	221
11.3 MOST150 Analysis Library and Node Functions	223
11.4 Functions	226
11.4.1 xIMost150SwitchEventSources	226
11.4.2 xIMost150SetDeviceMode	227
11.4.3 xIMost150GetDeviceMode	228
11.4.4 xIMost150SetSPDIFMode	229
11.4.5 xIMost150GetSPDIFMode	229
11.4.6 xIMost150SetSpecialNodeInfo	230
11.4.7 xIMost150GetSpecialNodeInfo	230
11.4.8 xIMost150SetFrequency	231
11.4.9 xIMost150GetFrequency	232
11.4.10 xIMost150GetSystemLockFlag	233
11.4.11 xIMost150GetShutdownFlag	233
11.4.12 xIMost150Shutdown	234
11.4.13 xIMost150Startup	234
11.4.14 xIMost150SetSSOResult	235
11.4.15 xIMost150GetSSOResult	235
11.4.16 xIMost150CtrlTransmit	236

11.4.17 xlMost150AsyncTransmit	237
11.4.18 xlMost150EthernetTransmit	237
11.4.19 xlMost150SyncGetAllocTable	238
11.4.20 xlMost150CtrlSyncAudio	238
11.4.21 xlMost150SyncSetVolume	239
11.4.22 xlMost150SyncGetVolume	240
11.4.23 xlMost150SyncSetMute	240
11.4.24 xlMost150SyncGetMute	241
11.4.25 xlMost150GetRxLightLockStatus	241
11.4.26 xlMost150SetTxLight	242
11.4.27 xlMost150GetTxLight	243
11.4.28 xlMost150SetTxLightPower	243
11.4.29 xlMost150GenerateLightError	244
11.4.30 xlMost150GenerateLockError	245
11.4.31 xlMost150CtrlConfigureBusload	246
11.4.32 xlMost150CtrlGenerateBusload	246
11.4.33 xlMost150AsyncConfigureBusload	247
11.4.34 xlMost150AsyncGenerateBusload	248
11.4.35 xlMost150ConfigureRxBuffer	248
11.4.36 xlMost150GenerateBypassStress	249
11.4.37 xlMost150SetECLLine	250
11.4.38 xlMost150SetECLTermination	251
11.4.39 xlMost150GetECLInfo	251
11.4.40 xlMost150ECLConfigureSeq	252
11.4.41 xlMost150ECLGenerateSeq	253
11.4.42 xlMost150SetECLGlitchFilter	254
11.4.43 Streaming	255
11.4.43.1 General Information	255
11.4.43.2 Layout of Streaming Data	256
11.4.44 xlMost150StreamOpen	257
11.4.45 xlMost150StreamClose	257
11.4.46 xlMost150StreamStart	258
11.4.47 xlMost150StreamStop	259
11.4.48 xlMost150StreamTransmitData	260
11.4.49 xlMost150StreamClearTxFifo	260
11.4.50 xlMost150StreamInitRxFifo	261
11.4.51 xlMost150StreamReceiveData	262
11.4.52 xlMost150StreamGetInfo	262
11.4.53 xlMost150Receive	264
11.4.54 xlMost150TwinklePowerLed	264
11.5 Structs	265
11.5.1 XLmost150AsyncBusloadConfig	265

11.5.2 XLmost150AsyncTxMsg	265
11.5.3 XLmost150CtrlBusloadConfig	266
11.5.4 XLmost150CtrlTxMsg	267
11.5.5 XLmost150EthernetTxMsg	267
11.5.6 XLmost150SetSpecialNodeInfo	268
11.5.7 XLmost150StreamInfo	269
11.5.8 XLmost150StreamOpen	269
11.5.9 XLmost150SyncAudioParameter	270
11.6 Events	272
11.6.1 XLmost150event	272
11.6.2 XLmost150AsyncBusloadConfig	273
11.6.3 XLmost150AsyncTxMsg	274
11.6.4 XLmost150EthernetTxMsg	275
11.6.5 XL_START	275
11.6.6 XL_STOP	275
11.6.7 XL_MOST150_EVENT_SOURCE_EV	276
11.6.8 XL_MOST150_DEVICE_MODE_EV	276
11.6.9 XL_MOST150_SPDIF_MODE_EV	277
11.6.10 XL_MOST150_FREQUENCY_EV	277
11.6.11 XL_MOST150_SPECIAL_NODE_INFO_EV	277
11.6.12 XL_MOST150_CTRL_SPY_EV	279
11.6.13 XL_MOST150_CTRL_RX_EV	281
11.6.14 XL_MOST150_CTRL_TX_ACK_EV	282
11.6.15 XL_MOST150_ASYNC_SPY_EV	283
11.6.16 XL_MOST150_ASYNC_RX_EV	285
11.6.17 XL_MOST150_ASYNC_TX_ACK_EV	285
11.6.18 XL_MOST150_CL_INFO	286
11.6.19 XL_MOST150_SYNC_ALLOC_INFO_EV	286
11.6.20 XL_MOST150_TX_LIGHT_EV	287
11.6.21 XL_MOST150_RXLIGHT_LOCKSTATUS_EV	287
11.6.22 XL_MOST150_ERROR_EV	287
11.6.23 XL_MOST150_CTRL_SYNC_AUDIO_EV	288
11.6.24 XL_MOST150_SYNC_VOLUME_STATUS_EV	289
11.6.25 XL_MOST150_SYNC_MUTE_STATUS_EV	290
11.6.26 XL_MOST150_LIGHT_POWER_EV	290
11.6.27 XL_MOST150_GEN_LIGHT_ERROR_EV	291
11.6.28 XL_MOST150_GEN_LOCK_ERROR_EV	291
11.6.29 XL_MOST150_CONFIGURE_RX_BUFFER_EV	291
11.6.30 XL_MOST150_CTRL_BUSLOAD_EV	292
11.6.31 XL_MOST150_ASYNC_BUSLOAD_EV	293
11.6.32 XL_MOST150_ETHERNET_SPY_EV	293

11.6.33 XL_MOST150_ETHERNET_RX_EV	295
11.6.34 XL_MOST150_ETHERNET_TX_ACK_EV	295
11.6.35 XL_MOST150_SYSTEMLOCK_FLAG_EV	296
11.6.36 XL_MOST150_SHUTDOWN_FLAG_EV	296
11.6.37 XL_MOST150_NW_STARTUP_EV	297
11.6.38 XL_MOST150_NW_SHUTDOWN_EV	297
11.6.39 XL_MOST150_ECL_EV	297
11.6.40 XL_MOST150_ECL_TERMINATION_EV	298
11.6.41 XL_MOST150_ECL_SEQUENCE_EV	298
11.6.42 XL_MOST150_ECL_GLITCH_FILTER_EV	298
11.6.43 XL_MOST150_STREAM_STATE_EV	299
11.6.44 XL_MOST150_STREAM_TX_BUFFER_EV	300
11.6.45 XL_MOST150_STREAM_TX_LABEL_EV	301
11.6.46 XL_MOST150_STREAM_TX_UNDERFLOW_EV	302
11.6.47 XL_MOST150_STREAM_RX_BUFFER_EV	303
11.6.48 XL_MOST150_GEN_BYPASS_STRESS_EV	304
11.6.49 XL_MOST150_SSO_RESULT_EV	304
11.7 Application Examples	306
11.7.1 xIMOST150View	306
11.7.1.1 General Information	306
11.7.1.2 Classes	306
11.7.1.3 Functions	307

12 FlexRay Commands 310

12.1 Introduction	311
12.2 Flowchart	312
12.3 Free Libray and Advanced Library	313
12.4 FlexRay Basics	314
12.4.1 Introduction	314
12.4.2 Data Transmission Requirements	314
12.4.3 FlexRay Communication Architecture	315
12.4.4 Deterministic and Dynamic	317
12.4.5 CRC-Protected Data Transmission	319
12.5 Functions	320
12.5.1 xlFrSetConfiguration	320
12.5.2 xlFrGetChannelConfiguration	320
12.5.3 xlFrSetMode	321
12.5.4 xlFrInitStartupAndSync	321
12.5.5 xlFrSetupSymbolWindow	322
12.5.6 xlFrActivateSpy	322
12.5.7 xlSetTimerBaseNotify	323

12.5.8 xlFrReceive	323
12.5.9 xlFrTransmit	324
12.5.10 xlFrSetTransceiverMode	324
12.5.11 xlFrSendSymbolWindow	325
12.5.12 xlFrSetAcceptanceFilter	326
12.6 Structs	327
12.6.1 XLfrClusterConfig	327
12.6.2 XLfrChannelConfig	332
12.6.3 XLfrMode	332
12.6.4 XLfrAcceptanceFilter	333
12.7 Events	335
12.7.1 XLfrEvent	335
12.7.2 XL_FR_START_CYCLE_EV	336
12.7.3 XL_FR_RX_FRAME_EV	337
12.7.4 XL_FR_TX_FRAME_EV	339
12.7.5 XL_FR_TXACK_FRAME	340
12.7.6 XL_FR_INVALID_FRAME	340
12.7.7 XL_FR_WAKEUP_EV	340
12.7.8 XL_FR_SYMBOL_WINDOW_EV	341
12.7.9 XL_FR_ERROR_EV	342
12.7.10 XL_FR_ERROR_POC_MODE_EV	342
12.7.11 XL_FR_ERROR_SYNC_FRAMES_BELOWMIN	343
12.7.12 XL_FR_ERROR_SYNC_FRAMES_EV	343
12.7.13 XL_FR_ERROR_CLOCK_CORR_FAILURE_EV	343
12.7.14 XL_FR_ERROR_NIT_FAILURE_EV	344
12.7.15 XL_FR_ERROR_CC_ERROR_EV	344
12.7.16 XL_FR_STATUS_EV	345
12.7.17 XL_FR_NM_VECTOR_EV	346
12.7.18 XL_FR_SPY_FRAME_EV	347
12.7.19 XL_FR_SPY_SYMBOL_EV	347
12.7.20 XL_APPLICATION_NOTIFICATION_EV	348
12.8 Application Examples	349
12.8.1 xlFlexDemo	349
12.8.1.1 General Information	349
12.8.1.2 Classes	349
12.8.1.3 Functions	349
12.8.1.4 Events	350
12.8.2 xlFlexDemoCmdLine	351
12.8.2.1 General Information	351
12.8.2.2 Functions	351
12.8.3 Fibex2CSharpReaderDemo	352
12.8.3.1 General Information	352

12.8.3.2 Classes	352
13 Ethernet Commands	353
13.1 Introduction	354
13.2 Flowchart	355
13.3 Functions	356
13.3.1 xlEthSetConfig	356
13.3.2 xlEthGetConfig	356
13.3.3 xlEthSetBypass	357
13.3.4 xlEthTransmit	359
13.3.5 xlEthReceive	359
13.3.6 xlEthTwinkleStatusLed	360
13.4 Structs	361
13.4.1 XLdriverConfig	361
13.4.2 T_XL_ETH_CONFIG	363
13.4.2.1 Valid Configuration Combinations	364
13.5 Events	366
13.5.1 T_XL_ETH_FRAME	366
13.5.2 T_XL_ETH_EVENT	366
13.5.3 T_XL_ETH_DATAFRAME_RX	367
13.5.4 T_XL_ETH_DATAFRAME_RX_ERROR	368
13.5.5 T_XL_ETH_DATAFRAME_TX_EVENT	369
13.5.6 T_XL_ETH_DATAFRAME_TXACK	370
13.5.7 T_XL_ETH_DATAFRAME_TXACK_OTHERAPP	370
13.5.8 T_XL_ETH_DATAFRAME_TXACK_SW	371
13.5.9 T_XL_ETH_DATAFRAME_TX_ERROR	371
13.5.10 T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP	371
13.5.11 T_XL_ETH_DATAFRAME_TX_ERR_SW	372
13.5.12 T_XL_ETH_CONFIG_RESULT	372
13.5.13 T_XL_ETH_LOSTEVENT	372
13.5.14 T_XL_ETH_CHANNEL_STATUS	373
13.5.15 T_XL_ETH_DATAFRAME_TX	374
13.6 Application Examples	376
13.6.1 xlEthDemo	376
13.6.1.1 General Information	376
13.6.1.2 Keyboard Commands	376
13.6.1.3 Command Line Interface	377
13.6.2 xlEthBypassDemo	378
14 ARINC 429 Commands	379

14.1 Introduction	380
14.2 Flowchart	381
14.3 Functions	382
14.3.1 xIA429SetChannelParams	382
14.3.2 xIA429Transmit	383
14.3.3 xIA429Receive	385
14.4 Structs	387
14.4.1 XL_A429_PARAMS	387
14.4.2 XL_A429_MSG_TX	392
14.5 Events	395
14.5.1 XLa429Event	395
14.5.2 XL_A429_EV_TX_OK	396
14.5.3 XL_A429_EV_TAG_TX_ERR	397
14.5.4 XL_A429_EV_TAG_RX_OK	398
14.5.5 XL_A429_EV_TAG_RX_ERR	399
14.5.6 XL_A429_EV_BUS_STATISTIC	401
14.6 Application Examples	403
14.6.1 xIA429Control	403
14.6.1.1 General Information	403
15 .NET Wrapper	404
15.1 Overview	405
15.2 XLDriver - Accessing Driver	406
15.3 XLClass - Storing Data/Parameters	407
15.4 XLDefine - Using Predefined Values	408
15.5 Including the Wrapper in a New .NET Project	409
15.6 Application Examples	411
15.6.1 xICANdemo .NET	411
15.6.2 xICANDemo .NET	412
15.6.3 xILINdemo .NET	413
15.6.4 xILINdemo Single .NET	414
15.6.5 xIDAIOexample .NET	415
15.6.5.1 General Information	415
15.6.5.2 Setup	415
15.6.5.3 Keyboard commands	416
15.6.5.4 Output Examples	417
15.6.6 xIOPiggyExample .NET	419
15.6.6.1 General Information	419
15.6.6.2 Setup	419
15.6.7 xIEthernetDemo .NET	421

15.6.8 4.8 xlFRdemo .NET	422
16 Error Codes	423
16.1 XL Status Error Codes	424
16.2 MOST150 Error Codes	426
16.3 Ethernet Error Codes	427

1 Introduction

In this chapter you find the following information:

1.1 About this User Manual	19
1.1.1 Certification	19
1.1.2 Warranty	20
1.1.3 Registered Trademarks	20

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
Microsoft	Legally protected proper names and side notes.
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 Certification

Certified Quality Management System

Vector Informatik GmbH has ISO 9001:2008 certification. The ISO standard is a globally recognized standard.

1.1.2 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.

1.1.3 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
are trademarks of the Microsoft Corporation.

2 Overview

In this chapter you find the following information:

2.1 General Information	22
2.2 Principles of the XL Driver Library	23
2.3 Driver Files and Examples	27
2.4 System Requirements	28
2.5 Additional Information	29

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, MOST, 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.

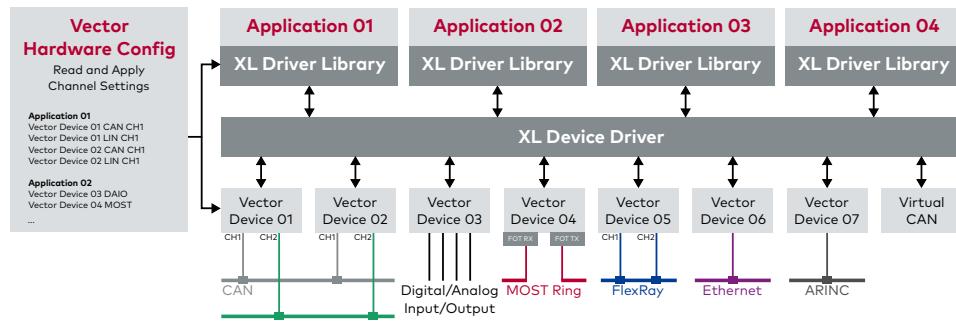


Figure 1: Example of applications using the XL Driver Library to access Vector devices

Vector Hardware Config

The **Vector Hardware Config** tool is required to set up the hardware settings like physical channel assignment etc. 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.

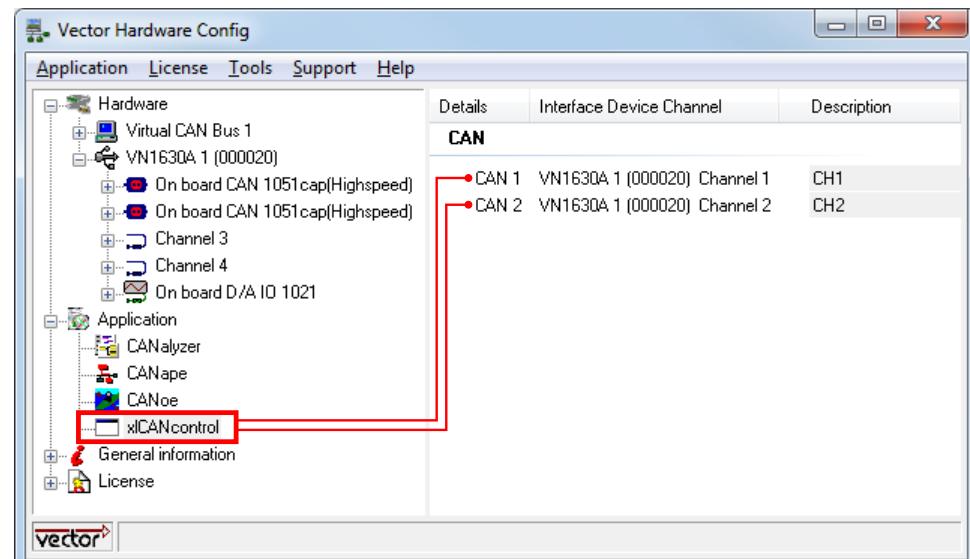


Figure 2: Example of hardware settings - `xICANcontrol` accesses VN1630A (CH1/CH2)



Reference

Please refer to the user manual of your Vector device for detailed information on the hardware installation and the **Vector Hardware Configuration** tool.

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 Configuration** tool, the following sections provide background information on indexing of hardware channels which is required in almost each function call.

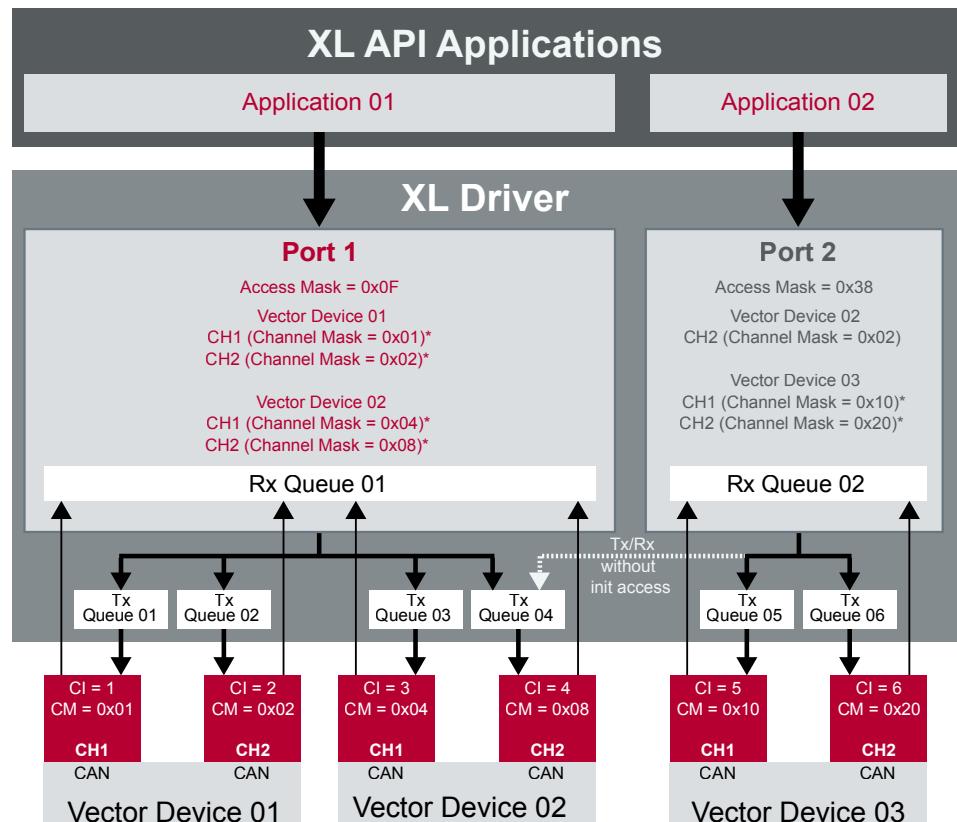


Figure 3: Principle structure of CAN applications

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 37).

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	0b00000001	0x001
	2	0b00000010	0x002
02	3	0b01000000	0x040
	4	0b10000000	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 7/8/10
vxlapi64.dll	64 bit DLL for Windows 7/8/10
vxlapi.h	C header for C/C++ based applications
vxlapi_.NET.dll	Wrapper for .NET bases applications (requires vxlapi64.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:

- > CANcardXL/XLe
- > CANboardXL Family
- > CANcaseXL/XL log
- > VN0600 Interface Family
- > VN1600 Interface Family
- > VN2600 Interface Family
- > VN5600 Interface Family
- > VN7000 Interface Family
- > VN8800 Interface Family
- > VN8900 Interface Family
- > VX0300 Interface Family

Supported operating systems

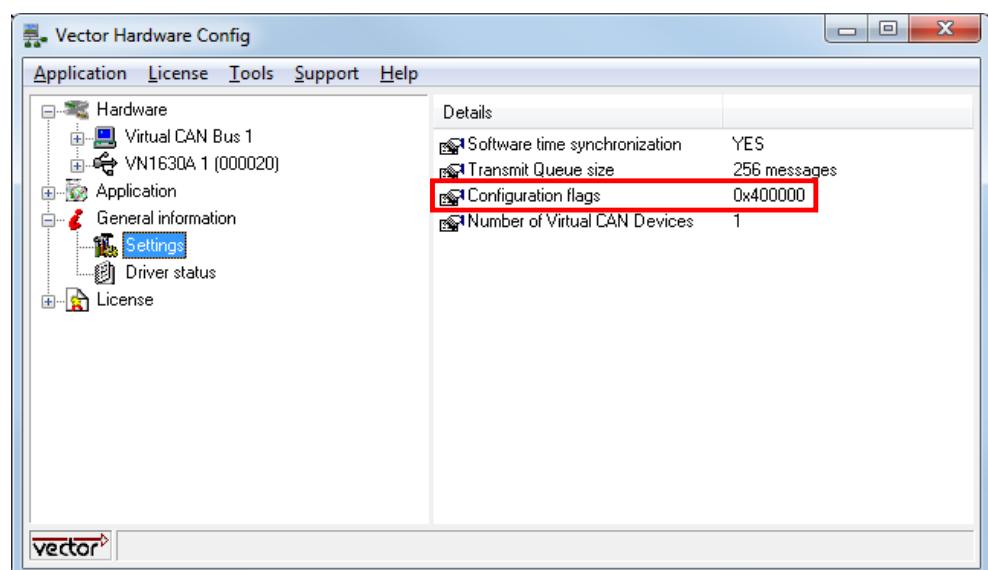
The **XL Driver Library** is compatible with the following operating systems:

- > Windows 7 (32 bit / 64 bit)
- > Windows 8 (32 bit / 64 bit)
- > Windows 10 (64 bit)

2.5 Additional Information

Debug prints

The **XL Driver Library** supports debug prints which can be enabled in the **Vector Hardware Configuration** tool. In section **General information**, select **Settings** and double-click on **Configuration flags**. Enter the required flag (see table below). To activate the flags, restart the PC.



Flags	Supported Bus Type
0x400000	CAN, LIN, DAIO
0x2000	MOST
0x010000	FlexRay



Reference

The debug prints can be viewed with the freeware tool **DebugView** (download from Microsoft website: <https://technet.microsoft.com/de-de/sysinternals/bb896647%28en-us%29.aspx>).

3 Common Commands

In this chapter you find the following information:

3.1 Introduction	31
3.2 Functions	32
3.3 Structs	50
3.4 Events	59

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 Configuration** tool as well as to open or close ports (see section [Principles of the XL Driver Library](#) on page 23).



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 Functions

3.2.1 xlOpenDriver

Syntax	<pre>XLstatus xlOpenDriver(void)</pre>
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 423).

3.2.2 xlCloseDriver

Syntax	<pre>XLstatus xlCloseDriver(void)</pre>
Description	This function closes the driver.
Return value	Returns an error code (see section Error Codes on page 423).

3.2.3 xlGetAppConfig

Syntax	<pre>XLstatus xlGetAppConfig(char *appName, unsigned int appChannel, unsigned int *pHwType, unsigned int *pHwIndex, unsigned int *pHwChannel, unsigned int busType)</pre>
Description	Retrieves the hardware settings for an application which are configured in the Vector Hardware Configuration tool. The information can then be used to get the required channel mask (see section xlGetChannelMask on page 36). 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.

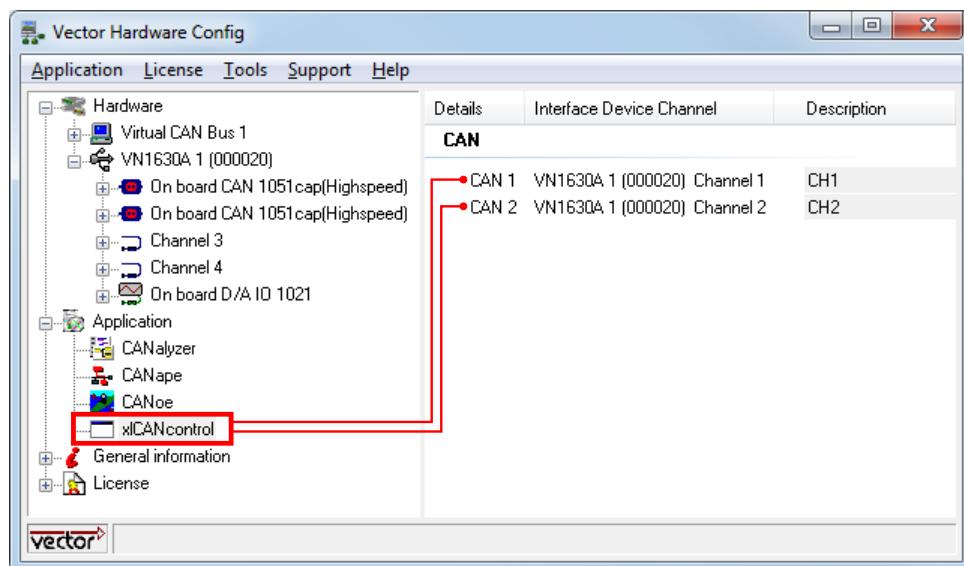


Figure 4: Example of hardware settings - xICANcontrol accesses VN1630A (CH1/CH2)

Input parameters

> **appName**

Name of the application to be read (e. g. "xICANcontrol").

Application names are listed in the **Vector Hardware Configuration** tool.

> **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 Config** tool.

> **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_MOST
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_HWTTYPE_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 423).

3.2.4 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 Config** tool 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 Configuration** tool.

> **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_HWTTYPE_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 423).

3.2.5 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.

Input parameters> **XLdriverConfig**

Points to the information structure that is returned by the driver (see section [XLdriverConfig on page 50](#)).

Return value

Returns an error code (see section [Error Codes on page 423](#)).

3.2.6 xlGetRemoteDriverConfig

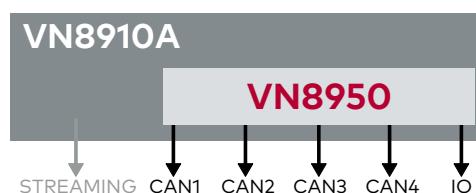
Syntax

```
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.

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 423](#)).

**Note**

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

3.2.7 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_HWTYPED_CANCARDXL
- XL_HWTYPED_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.

3.2.8 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 Configuration** tool via `xlGetApIConfig()`.

Input parameters**> hwType**

Required to distinguish the different hardware types, e. g.

- 1
- XL_HWTYPED_CANCARDXL
- XL_HWTYPED_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_HWTYPED_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);
```

3.2.9 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**.

Input parameters

- > **userName**
The name of the application that is listed in the **Vector Hardware Configuration** tool.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.
- > **rxQueueSize**
CAN, LIN, DAIO
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`.
 - 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).
 - MOST, 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.
`XL_INTERFACE_VERSION_V4` for MOST,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_FLEXRAY`
`XL_BUS_TYPE_AFDX`
`XL_BUS_TYPE_MOST`
`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 423).


Note

For LIN (`busType = XL_BUS_TYPE_LIN`), **init access** is needed (see section [Introduction](#) on page 99). If the LIN channel gets no **init access** the function returns `XL_ERR_INVALID_ACCESS`.

3.2.10 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 423).

3.2.11 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

**Note**

For XL Interface Family (excluding CANcardXLe): Timer events will be dropped if the Rx fifo level is above a specific level. If the application timing is based on Rx events, all Rx events should be used (not only timer events).

Input parameters> **portHandle**

The port handle retrieved by `xIOpenPort()`.

> **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 423).

3.2.12 `xISetTimerRateAndChannel`

Syntax

```
XLstatus xISetTimerRateAndChannel (
    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

> **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

**Note**

Timer events will only be generated if no other event occurs during the timer interval. Timer events might be dropped if other events occur.

Input parameters> **portHandle**

The port handle retrieved by `xIOpenPort()`.

> **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 423).
---------------------	--

3.2.13 xlResetClock

Syntax

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

Description

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

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

Return value

Returns an error code (see section [Error Codes](#) on page 423).

3.2.14 xlSetNotification

Syntax

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

Description

The function returns the notification handle. It notifies when messages are available in the receive queue. The handle is closed when unloading the library.

The parameter `queueLevel` specifies the number of messages that triggers the event. Note that the event is triggered only once when the `queueLevel` is reached. An application should read all available messages by [xlReceive\(\)](#) to be sure to re-enable the event.

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **queueLevel**

Queue level that triggers this event. For LIN, this is fixed to '1'.

Output parameters

> **handle**

Pointer to a WIN32 event handle.

Return value

Returns an error code (see section [Error Codes](#) on page 423).



Example

Setting up the notification for a CAN application

See example in [xlReceive\(\)](#).

3.2.15 xlFlushReceiveQueue

Syntax

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

Description

This function flushes the port's receive queue.

Input parameters	> portHandle The port handle retrieved by <code>xlOpenPort()</code> .
Return value	Returns an error code (see section Error Codes on page 423).

3.2.16 `xlGetReceiveQueueLevel`

Syntax	<pre>XLstatus xlGetReceiveQueueLevel (XLportHandle portHandle, int *level)</pre>
Input parameters	> portHandle The port handle retrieved by <code>xlOpenPort()</code> .
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 37).
Return value	Returns an error code (see section Error Codes on page 423).

3.2.17 `xlActivateChannel`

Syntax	<pre>XLstatus xlActivateChannel(XLportHandle portHandle, XLaccess accessMask, unsigned int busType, unsigned int flags)</pre>
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	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section xlGetChannelMask on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23. > busType Bus type that has also been used for <code>xlOpenPort()</code>. > flags Additional flags for activating the channels: <p><code>XL_ACTIVATE_RESET_CLOCK</code> Resets the internal clock after activating the channel.</p> <p><code>XL_ACTIVATE_NONE</code></p>
Return value	Returns an error code (see section Error Codes on page 423).



Example

Channel Activation

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

3.2.18 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
- > 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 59). 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 423)

**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", xlGetString(&xlEvent));
            }
        }
    }
}

```

3.2.19 xlGetString

Syntax

```
XLstringType xlGetString (XLevent *ev)
```

Description

Returns the textual description of the given event.

Supported bus types and events:

- > CAN
- > LIN
- > partly DAIO
- > common events (e. g. TIMER events)

Input parameters> **ev**

Points to the event (see section XLevent on page 59).

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
- > **00000000000000**
D0 to D7 are set to 0.
- > **TX tid=CC**
Tx flag, message was transmitted successfully by the CAN controller.

3.2.20 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 423)

Return value

Error code as plain text string.

3.2.21 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()`.

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 423).

3.2.22 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.

Input parameters

- > **portHandle**
 The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
 The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

Output parameters > **pChannelTime**
 64 bit PC-based card time.

Return value Returns an error code (see section [Error Codes](#) on page 423).

3.2.23 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.

Input parameters

- > **portHandle**
 The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
 The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

Return value Returns an error code (see section [Error Codes](#) on page 423).

3.2.24 xlPopupHwConfig

Syntax	<pre>XLstatus xlPopupHwConfig (char *callSign, unsigned int waitForFinish)</pre>
Description	Call this function to pop up the Vector Hardware Config tool.
Input parameters	<ul style="list-style-type: none"> > callSign Reserved type. > 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 423).

3.2.25 xlDeactivateChannel

Syntax	<pre>XLstatus xlDeactivateChannel (XlportHandle portHandle, XLaccess accessMask)</pre>
Description	The selected channels go off the bus. The channels are deactivated if there is no further port that activates the channels.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by xlOpenPort(). > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section xlGetChannelMask on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23.
Return value	Returns an error code (see section Error Codes on page 423).

3.2.26 xlGetLicenseInfo

Syntax	<pre>XLstatus xlGetLicenseInfo (XLaccess channelMask, XLlicenseInfo *pLicInfoArray, unsigned int licInfoArraySize)</pre>
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	<ul style="list-style-type: none"> > 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 57).
Return value	Returns an error code (see section Error Codes on page 423).

3.2.27 xlSetGlobalTimeSync

Syntax	<pre>XLstatus xlSetGlobalTimeSync (unsigned long newValue, unsigned long *previousValue);</pre>
Description	Reads/sets the software synchronization setting in the Vector Hardware Config tool. 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	<p>> newValue</p> <ul style="list-style-type: none"> XL_SET_TIMESYNC_NO_CHANGE Use this value to read the current setting which is stored in <code>previousValue</code>. XL_SET_TIMESYNC_ON Enables the software synchronization in the Vector Hardware Config tool. XL_SET_TIMESYNC_OFF Disables the software synchronization in the Vector Hardware Config tool.
Output parameters	> previousValue Buffer which stores the previous value.
Return value	Returns an error code (see section Error Codes on page 423).

3.2.28 xlGetKeymanBoxes

Syntax	<pre>XLstatus xlGetKeymanBoxes(unsigned int* boxCount);</pre>
Description	Returns the connected Keyman license dongles.
Output parameters	> boxCount Number of connected Keyman license dongles.
Return value	Returns an error code (see section Error Codes on page 423).

3.2.29 xlGetKeymanInfo

Syntax	<pre>XLstatus xlGetKeymanInfo (unsigned int boxIndex, unsigned int* boxMask, unsigned int* boxSerial, XLuint64* licInfo);</pre>
--------	--

Description	Returns the serial number and license info (license bits) of a selected Keyman license dongle.
Input parameters	<ul style="list-style-type: none"> > boxIndex Index of the Keyman license dongle (zero based).
Output parameters	<ul style="list-style-type: none"> > 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 423).

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 SerialNumber:
            %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 Structs

3.3.1 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 50). `XL_CONFIG_MAX_CHANNELS=64`.

3.3.2 XLchannelConfig

Syntax

```
typedef struct s_xl_channel_config {
    char          name [XL_MAX_LENGTH + 1];
    unsigned char 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 int   reserved[8];
} XLchannelConfig;
```

Description

This structure is used in [XLdriverConfig](#) (see section [XLdriverConfig](#) on page 50).

Parameters

- > **name**
The channel's name.
- > **hwType**
Contains the hardware types (see vxlapi.h),
e. g. CANcardXL: XL_HWTTYPE_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 ($1 \ll \text{channelIndex}$).
- > **channelCapabilities**
XL_CHANNEL_FLAG_TIME_SYNC_RUNNING
XL_CHANNEL_FLAG_TIME_SYNC_POSSIBLE
XL_CHANNEL_FLAG_TIME_SYNC_ENABLED
XL_CHANNEL_FLAG_NO_HWSYNC_SUPPORT
XL_CHANNEL_FLAG_TX_OFF_MODE_ENABLED
XL_CHANNEL_FLAG_32MHZ_CAN_CLOCK
XL_CHANNEL_FLAG_FR_SPY_ONLY_SUPPORT
XL_CHANNEL_FLAG_FR_IL_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.

The channel (hardware) supports the bus types:

```
XL_BUS_COMPATIBLE_CAN  
XL_BUS_COMPATIBLE_LIN  
XL_BUS_COMPATIBLE_FLEXRAY  
XL_BUS_COMPATIBLE_MOST  
XL_BUS_COMPATIBLE_DAIO  
XL_BUS_COMPATIBLE_J1708  
XL_BUS_COMPATIBLE_ETHERNET  
XL_BUS_COMPATIBLE_A429
```

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_MOST  
XL_BUS_ACTIVE_CAP_DAIO  
XL_BUS_ACTIVE_CAP_J1708  
XL_BUS_ACTIVE_CAP_ETHERNET  
XL_BUS_ACTIVE_CAP_A429
```

> 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 53).

> 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.

3.3.3 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 activeSpeedGrade;
            unsigned int compatibleSpeedGrade;
            unsigned int inicFwVersion;
        } most;
        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 bit rate (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

> **arbitrationBitRate**

CAN bus timing for nominal/arbitration bit rate.

> **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 bit rate.
- > **tseg1Dbr**
Bus timing value tseg1.
- > **tseg2Dbr**
Bus timing value tseg1.
- > **dataBitRate**
Data bit rate.
- > **canOpMode**
CAN 2.0: XL_BUS_PARAMS_CANOPMODE_CAN20
CAN FD: XL_BUS_PARAMS_CANOPMODE_CANFD

MOST

- > **activeSpeedGrade**
- > **compatibleSpeedGrade**
- > **inicFwVersion**

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_ASYNCNOMOUS
- > **baudrate**
FlexRay baud rate in kBaud.

Ethernet

- > **macAddr**
The MAC address starting with MSB.
- > **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_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**

3.3.4 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 Events

3.4.1 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

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.

3.4.2 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_daio_data    daioData;
    struct s_xl_transceiver transceiver;
};
```

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.
- > **daioData**
Structure for all DAIO data.
- > **transceiver**
Structure for all TRANSCEIVER events.

3.4.3 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 59).

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.

3.4.4 XL Transceiver

Syntax

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

Tag

XL_TRANSCEIVER (see section XEvent on page 59).

Parameters**> event_reason**

Reason for occurred event.

> is_present

Always valid transceiver.

3.4.5 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 XEvent on page 59).

4 CAN Commands

In this chapter you find the following information:

4.1 Introduction	63
4.2 Flowchart	64
4.3 Functions	65
4.4 Structs	76
4.5 Events	77
4.6 Application Examples	80

4.1 Introduction

Description

The **XL Driver Library** enables the development of CAN applications for supported Vector devices (see section [System Requirements](#) on page 28). Multiple CAN applications can use a common physical CAN channel at the same time.

Depending on the channel property **init access** (see page 25), 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.

4.2 Flowchart

Calling sequence

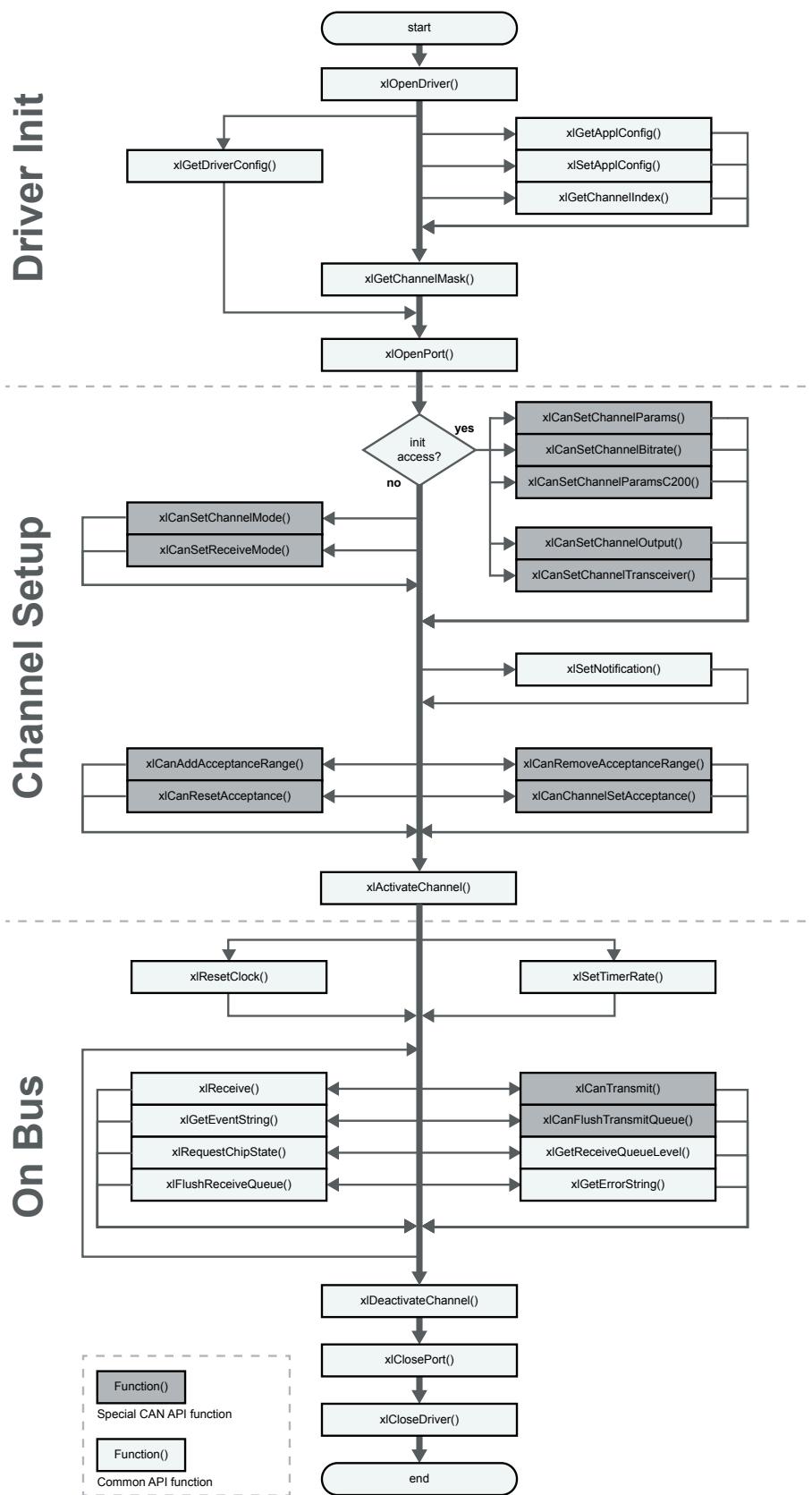


Figure 5: Function calls for CAN applications

4.3 Functions

4.3.1 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **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 423).

4.3.2 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 [xIOpenPort](#) on page 37) for the specified channels, and the channels must be deactivated.

Input parameters> **portHandle**

The port handle retrieved by [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).

4.3.3 xICanSetReceiveMode**Syntax**

```
XLstatus xICanSetReceiveMode (
    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 [xIOpenPort\(\)](#).

> **ErrorFrame**

Suppresses error frames.

> **ChipState**

Suppresses chipstate events.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

4.3.4 xICanSetChannelTransceiver**Syntax**

```
XLstatus xICanSetChannelTransceiver(
    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 [xIOpenPort](#) on page 37) to the channels.

Input parameters**> portHandle**

The port handle retrieved by [xIOpenPort\(\)](#).

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> 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 `vxlapic.h`.

> **lineMode**

Lowspeed (252/1053/1054)

`XL_TRANSCEIVER_LINEMODE_SLEEP`
Puts CANcab into sleep mode.

`XL_TRANSCEIVER_LINEMODE_NORMAL`
Enables normal operation.

Hightspeed (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`
One wire mode on CAN high.

`XL_TRANSCEIVER_LINEMODE_TT_CAN_L`
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 423).

4.3.5 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 37), and the selected channels must be deactivated.

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > pChipParams Pointer to an array of chip parameters (see section <code>XLchipParams</code> on page 76).
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

4.3.6 xlCanSetChannelParamsC200

Syntax	<pre>XLstatus xlCanSetChannelParamsC200 (XLportHandle portHandle, XLaccess accessMask, unsigned char btr0, unsigned char btr1)</pre>
Description	This function initializes the channels defined by <code>accessMask</code> with the given parameters. In order to call this function, the port must have init access (see section <code>xlOpenPort</code> on page 37), and the selected channels must be deactivated.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > 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 <code>Error Codes</code> on page 423).

4.3.7 xlCanSetChannelBitrate

Syntax	<pre>XLstatus xlCanSetChannelBitrate (XLportHandle portHandle, XLaccess accessMask, unsigned long bitrate)</pre>
Description	This function provides a simple way to specify the bit rate. The sample point is about 69 % (SJW=1, samples=1).

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > bitrate Bit rate in BPS. May be in the range 15000 ... 1000000.
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

4.3.8 `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.
`Accept if ((id ^ code) & mask) == 0`



Note

By default, all IDs are accepted after `xlOpenPort()`.

Input parameters

<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > 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: <code>XL_CAN_STD</code> <code>XL_CAN_EXT</code>
Return value

Returns an error code (see section `Error Codes` on page 423).

**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	0xFFFF	0xFFFF	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);
```

**Example****Set acceptance filter for several IDs (formula)**

```
code = id(1)
mask = 0xFFFF
loop over id(1) ... id(n)
mask = (! (id(n) & mask) xor (code&mask)) & mask
```

	Binary	General rule
ID = 6 (0x006)	0110	-
ID = 4 (0x004)	0100	-
Mask	1101	Compare the IDs at each bit position. If they are different, mask at this bit position must be '0'.
Code	0110	Take one ID (it does not matter which one).

4.3.9 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.

**Note**

By default, all **standard IDs** are accepted after `xlOpenPort()`. To receive only a specific ID range, the acceptance filter must be removed before.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **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 423).

**Example****Receiving IDs between 10...17 and 22...33**

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

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

4.3.10 `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.

**Note**

By default, all **standard IDs** are accepted after `xlOpenPort()`. This function is for **standard IDs** only.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

- > **first_id**
First ID to remove.
- > **last_id**
Last ID to remove.

Return value	Returns an error code (see section Error Codes on page 423).
--------------	--



Example

Removing range between 10...13 and 27...30

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

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

4.3.11 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **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 423).

**Example****Opening filter for all messages with extended IDs**

```
XLstatus xlStatus = xlCanResetAcceptance(XLportHandle,
                                         XLchannelMask,
                                         XL_CAN_EXT);
```

4.3.12 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

Return value

Returns an error code (see section **Error Codes** on page 423).

4.3.13 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> 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 423)

**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);
```

4.3.14 xlCanFlushTransmitQueue

Syntax

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

Description

The function flushes the transmit queues of the selected channels.

Input parameters

> **portHandle**

The port handle retrieved by xlOpenPort().

Return value

Returns an error code (see section Error Codes on page 423).

4.4 Structs

4.4.1 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 bit rate. (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

$$\text{Baudrate} = f/(2 \cdot \text{presc} \cdot (1 + \text{tseg1} + \text{tseg2}))$$

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

4.5 Events

4.5.1 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 59.

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.

4.5.2 XL Chip State**Syntax**

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

Description

This event occurs after calling `xICanRequestChipState()`.

Tag

XL_CHIP_STATE (see section XLevent on page 59).

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.

4.6 Application Examples

4.6.1 xICANdemo

4.6.1.1 General Information

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

```
xICANdemo <Baudrate> <ApplicationName> <Identifier>
```

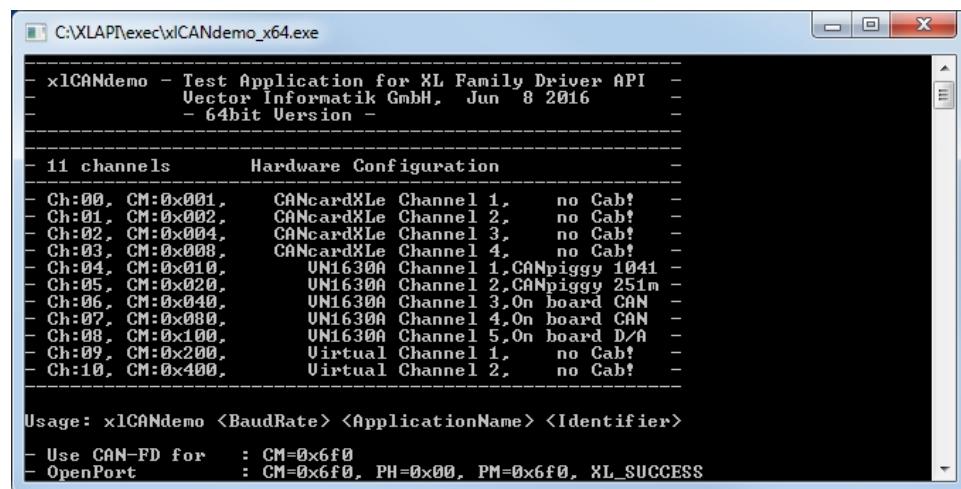


Figure 6: Running xICANdemo

4.6.1.2 Keyboard Commands

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

Key	Command
<t>	Transmit a message
	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)
<l>	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

Key	Command
<ESC>	Exit

4.6.1.3 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.

4.6.2 xICANcontrol

4.6.2.1 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 Config**.

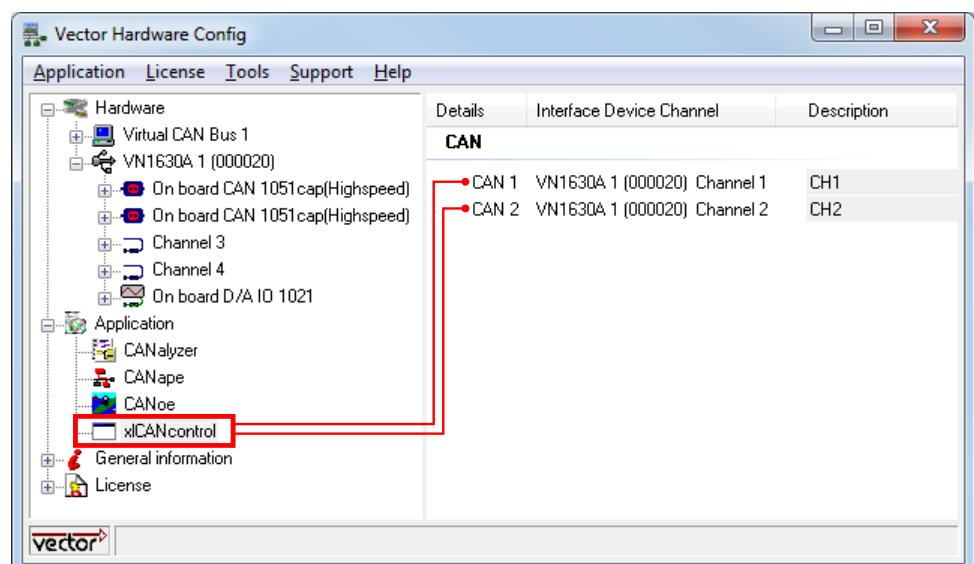


Figure 7: Example of hardware settings - xICANcontrol accesses VN1630A (CH1/CH2)



Note

If you want to use other CAN channels, close the application and change the assignments in the **Vector Hardware Config** tool. 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.

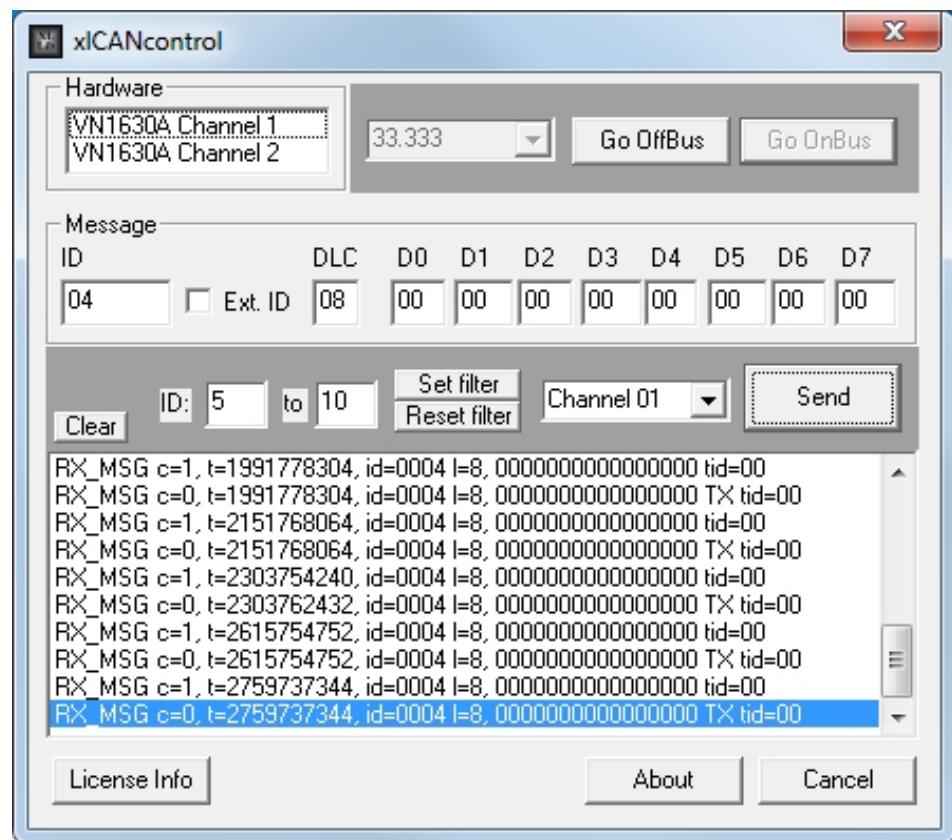


Figure 8: xICANcontrol accessing VN1630A (CH1/CH2)

4.6.2.2 Classes

Description

The example has the following class structure:

- > **CaboutDlg**
About box.
- > **CXLICANcontrolApp**
Main MFC class → xICANcontrol.cpp
- > **CXLICANcontrolIDlg**
The 'main' dialog box → xICANcontrol1Dlg.cpp
- > **CCANFunctions**
Contains all functions for the LIN access → xICANFunctions.cpp

4.6.2.3 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 `xICANtransmit()`.
- > **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 Config** tool with `xIGetApplConfig()`. If there is no application registered, the application searches for available CAN channels and assigns them in the **Vector Hardware Config** tool with `xISetApplConfig()`. The function fails if there are no valid channels found.
- > **canInit**
Opens one port with both channels (see section `xIOpenPort` on page 37).
- > **canCreateRxThread**
In order to readout the driver message queue, the application uses a thread (RxThread). An event is created and set up with `xISetNotification()` to notify the thread.

5 CAN FD Commands

In this chapter you find the following information:

5.1 Introduction	86
5.2 Flowchart	87
5.3 Functions	88
5.4 Structs	90
5.5 Events	91

5.1 Introduction

Description

The **XL Driver Library** enables the development of CAN FD applications for supported Vector devices (see section [System Requirements](#) on page 28). Multiple CAN applications can use a common physical CAN FD channel at the same time.

Depending on the channel property **init access** (see page 25), 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.

5.2 Flowchart

Calling sequence

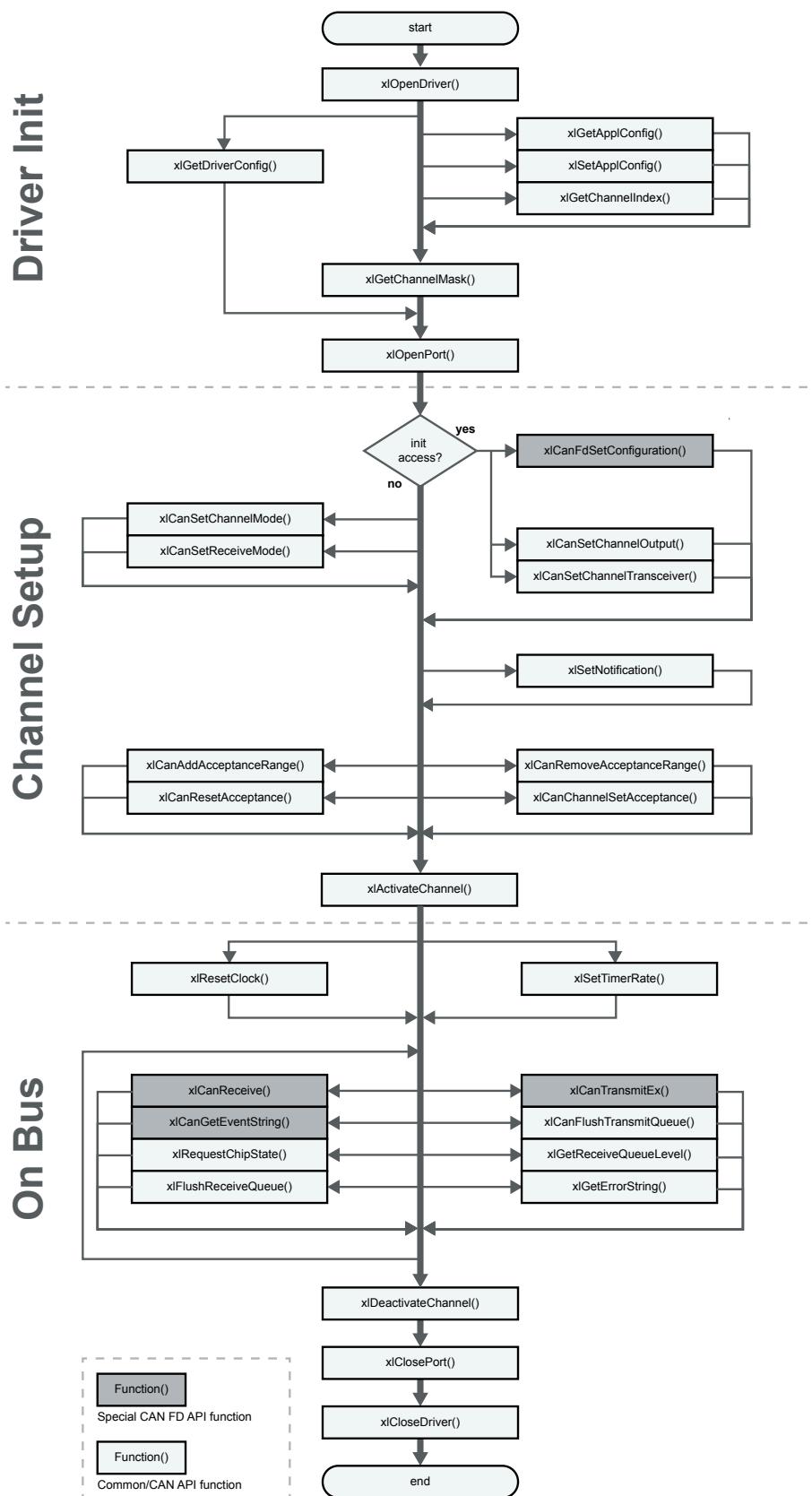


Figure 9: Function calls for CAN FD applications

5.3 Functions

5.3.1 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 37) for the specified channels.

Input parameters

- > **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **pCanFdConf**

Points to the CAN FD configuration structure to set up a CAN FD channel (see section [XLcanFdConf](#) on page 90).

Return value

Returns an error code (see section [Error Codes](#) on page 423).

5.3.2 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **msgCnt**
Amount of messages to be transmitted by the user.
- > **pMsgCntSent**
Amount of messages which were transmitted.
- > **pXICanTxEvt**
Points to a user buffer with messages to be transmitted (see section [XLcanTxEvent](#) on page 91).
At least the buffer must have the size of `msgCnt`.

Return value	Returns <code>XL_SUCCESS</code> 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, <code>XL_ERR_QUEUE_IS_FULL</code> is returned (see section Error Codes on page 423)
---------------------	---

5.3.3 xlCanReceive

Syntax	<pre>XLstatus xlCanReceive (XLportHandle portHandle, XLcanRxEvent *pXlCanRxEvt)</pre>
Description	The function receives the CAN FD messages on the selected port.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by xlOpenPort().
Input/output parameters	<ul style="list-style-type: none"> > pXLCanRxEvt Pointer to the application allocated receive event buffer (see section XLcanRxEvent on page 92).
Return value	<code>XL_ERR_QUEUE_IS_EMPTY</code> : No event is available (see section Error Codes on page 423)

5.3.4 xlCanGetEventString

Syntax	<pre>XLstringType xlCanGetEventString (XLcanRxEvent *pEv)</pre>
Description	This function returns a string based on the passed CAN Rx event data.
Input parameters	<ul style="list-style-type: none"> > pEv Points the CAN Rx event buffer to be parsed (see section XLcanRxEvent on page 92).
Return value	Returns an error code (see section Error Codes on page 423).

5.4 Structs

5.4.1 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 int reserved[2];
} XLcanFdConf;
```

Parameters

- > **arbitrationBitRate**
Arbitration CAN bus timing for nominal / arbitration bit rate.
- > **sjwAbr**
Arbitration CAN bus timing value (sample jump width).
- > **tseg1Abr**
Arbitration CAN bus timing tseg1.
- > **tseg2Abr**
Arbitration CAN bus timing tseg2.
- > **dataBitRate**
CAN bus timing for data bit rate. Set `dataBitRate = arbitrationBitrate` for transmitting CAN 2.0 frames.
- > **sjwDbr**
CAN bus timing value (sample jump width).
- > **tseg1Dbr**
CAN bus timing for data tseg1.
- > **tseg2Dbr**
CAN bus timing for data tseg2.
- > **reserved**
Reserved for future use. Set to 0.

5.5 Events

5.5.1 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.
- > **transId**
Internal use.
- > **channelIndex**
Channel index of the hardware (see section [xIGetChannelIndex](#) on page 36).
- > **reserved**.
Internal use.
- > **tagData**
Tag Data (see section [XL_CAN_TX_MSG](#) on page 91).

5.5.2 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;
```

Parameters

- > **canId**
CAN ID (11 or 29 bits).
- > **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.

> **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.

5.5.3 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         timeStamp;

    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	<ul style="list-style-type: none"> > 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_STATISTIC XL_CAN_EV_TAG_CHIP_STATE > channelIndex Channel index of the hardware (see section xIGetChannelIndex on page 36). > reserved Internal use. > userHandle Internal use. > flagsChip Queue overflow (upper 8bit), XL_CAN_QUEUE_OVERFLOW. > reserved0 Internal use. > reserved1 Internal use. > timeStamp Timestamp which is synchronized by the driver. > tagData Tag Data. See the following sections for further details.
------------	---

5.5.4 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;
```

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.

5.5.5 XL_CAN_EV_ERROR

Syntax

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

Parameters	<ul style="list-style-type: none"> > errorCode
	<ul style="list-style-type: none"> 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
	<ul style="list-style-type: none"> > reserved <p>Internal use.</p>

5.5.6 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;
```

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.

5.5.7 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;

```

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.

5.5.8 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;

```

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.

6 LIN Commands

In this chapter you find the following information:

6.1 Introduction	99
6.2 Flowchart	100
6.3 LIN Basics	101
6.4 Functions	102
6.5 Structs	108
6.6 Events	109
6.7 Application Examples	112

6.1 Introduction

Description

The **XL Driver Library** enables the development of LIN applications for supported Vector devices (see section System Requirements on page 28). A LIN application always requires **init access**(see section `xlOpenPort` on page 37)multiple LIN applications cannot use a common physical LIN channel at the same time.

Depending on the channel property **init access** (see page 25), 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.

6.2 Flowchart

Calling sequence

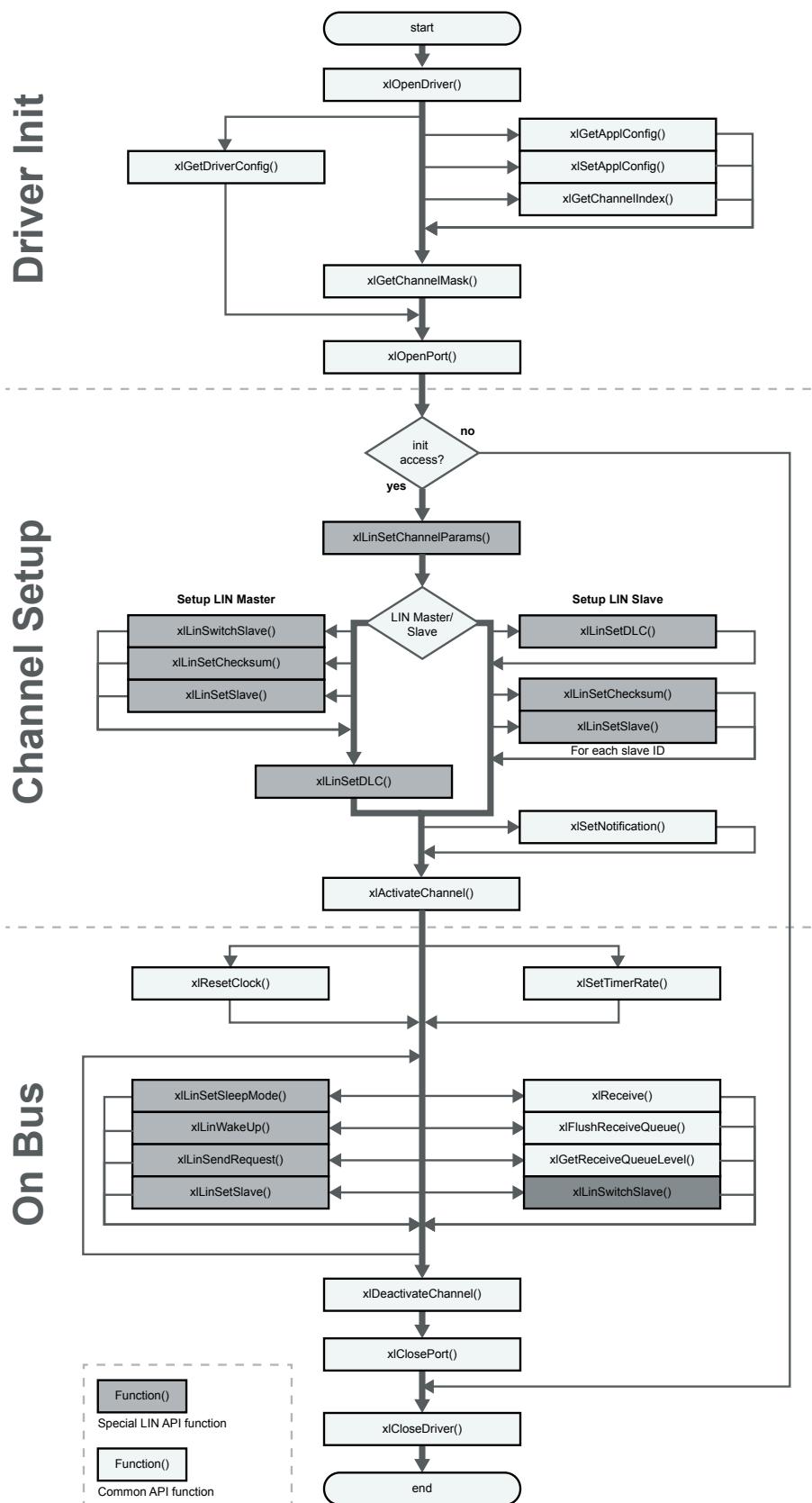


Figure 10: Function calls for LIN applications

6.3 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.

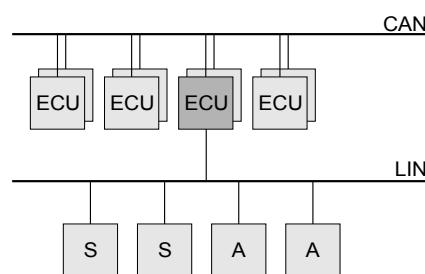
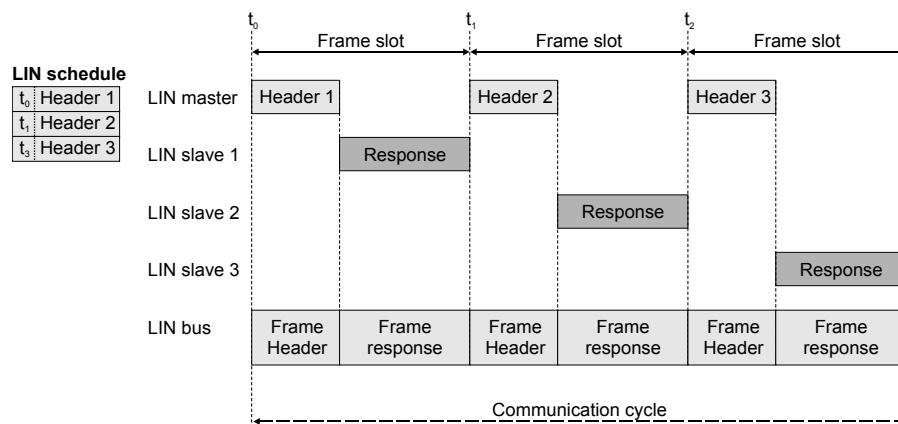


Figure 11: CAN and LIN

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 `xILinSetDLC()`). A response can contain up to eight data bytes (defined for each slave, see `xILinSetSlave()`).

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 `xILinSetChannelParams()` with Master flag. Furthermore, the XL API can be simultaneously used for LIN slaves, which must be configured separately via `xILinSetChannelParams()` (Slave flag), `xILinSetDLC`, `xILinSetChecksum()` and `xILinSetSlave`. See the LIN flowchart and the provided LIN examples for further details.

6.4 Functions

6.4.1 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 `xIOpenPort()`.

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

- > **statPar**

Defines the mode of the LIN channel and the baud rate (see section `XLLinStatPar` on page 108).

Return value

Returns an error code (see section `Error Codes` on page 423).

6.4.2 xlLinSetDLC

Syntax

```
XLstatus xlLinSetDLC(
    XLportHandle portHandle,
    XLaccess accessMask,
    unsigned char DLC[60]
)
```

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 `xIOpenPort()`.

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

- > **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 423).
---------------------	--

**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);
```

6.4.3 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).

**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);
```

6.4.4 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 [xiOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xiGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).

**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);
```

6.4.5 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 [xiOpenPort\(\)](#).

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the [Vector Hardware Configuration](#) tool if there is a prepared application setup (see section [xiGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> 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 423).

6.4.6 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **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 423).

6.4.7 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

Return value

Returns an error code (see section `Error Codes` on page 423).

6.4.8 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> flags

`XL_LIN_SET_SILENT`

Sets hardware into sleep mode (transmits no 'Sleep-Mode' frame).

`XL_LIN_SET_WAKEUPID`

Transmits the indicated LIN ID at wake up and set hardware into sleep mode. It is only possible on a LIN master.

> linID

Defines the LIN ID that is transmitted at wake-up.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

6.5 Structs

6.5.1 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**

Set 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.

> **LINVersion**

`XL_LIN_VERSION_1_3`

Use LIN 1.3 protocol

`XL_LIN_VERSION_2_0`

Use LIN 2.0 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);
```

6.6 Events

6.6.1 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 109).
- > **linNoAns**
Structure for the LIN message that gets no answer (see section [LIN No Answer](#) on page 110).
- > **linWakeUp**
Structure for the wake up events (see section [LIN Wake Up](#) on page 110).
- > **linSleep**
Structure for the sleep events (see section [LIN Sleep](#) on page 111).
- > **linCRCinfo**
Structure for the CRC info events (see section [LIN CRC Info](#) on page 111).

6.6.2 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 59).

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.
- > **data**
`XL_LIN_MSGFLAG_CRCERROR`
LIN CRC error.
- > **crc**
Content of the message.
- > **crc**
Checksum.

6.6.3 XL LIN Error Message

Tag	<code>XL_LIN_ERRMSG</code> (see section XLevent on page 59).
-----	--

6.6.4 XL LIN Sync Error

Description	Notifies an error in analyzing the sync field.
Tag	<code>XL_LIN_SYNC_ERR</code> (see section XLevent on page 59).

6.6.5 LIN No Answer

Syntax	<pre>struct s_lin_NoAns { unsigned char id; }</pre>
Description	If a LIN master request gets no slave response a <code>linNoAns</code> event is received.
Tag	<code>XL_LIN_NOANS</code> (see section XLevent on page 59).
Parameters	<ul style="list-style-type: none"> > id The LIN ID on which was the master request.

6.6.6 LIN Wake Up

Syntax	<pre>struct s_xl_lin_wake_up { unsigned char flag; unsigned char unused[3]; unsigned int startOffs; unsigned int width; };</pre>
Description	When a channel wakes up (comes out of the sleep mode) a <code>linWakeUp</code> event is received.
Tag	<code>XL_LIN_WAKEUP</code> (see section XLevent on page 59).
Parameters	<ul style="list-style-type: none"> > flag If the wake-up signal comes from the internal hardware, the flag is set to <code>XL_LIN_WAKUP_INTERNAL</code> 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
```

6.6.7 LIN Sleep

Syntax

```
struct s_lin_sleep {
    unsigned char flag;
}
```

Description

For this event, there can be different reasons:

- > After `xIActivatechannel()` a linSleep event is received (only for a LIN application).
- > After `xILinWakeUp()` (e. g. an internal wake-up).
- > After receiving a LIN message the master goes back into sleep mode.

Tag

`XL_LIN_SLEEP` (see section [XLevent](#) on page 59).

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).

6.6.8 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 `xILinSetChecksum()` 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 59).

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.

6.7 Application Examples

6.7.1 xLINExample

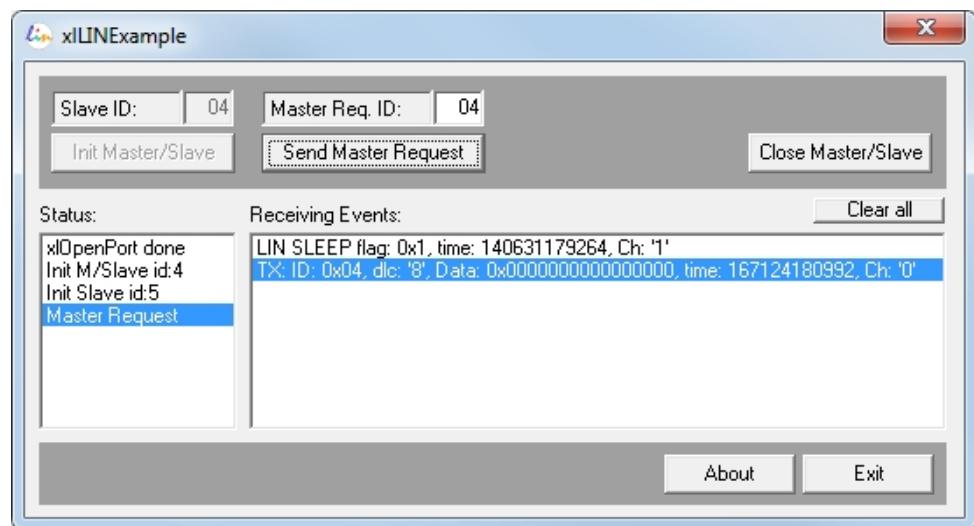
6.7.1.1 General Information

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 Configuration** tool. 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.



6.7.1.2 Classes

Description

The example has the following class structure:

- > **CaboutDlg**
About box. → `AboutDlg.cpp`
- > **CLINExampleApp**
Main MFC class → `xLINExample.cpp`
- > **CLINExampleDlg**
The 'main' dialog box → `xLINExampleDlg.cpp`
- > **CLINFunctions**
Contains all functions for the LIN access → `xLINFunctions.cpp`

6.7.1.3 Functions

- | | |
|--------------------|--|
| Description | <ul style="list-style-type: none">> LINGetDevice
In order to get the channel mask, use <code>xIGetChannelMask()</code> to read all hardware parameters. <code>xIGetApplConfig()</code> checks whether the application has already been assigned. If not, a new entry with <code>xISetApplConfig()</code> is created.> LINInit
<code>LINInit</code> 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 <code>xIOpenPort()</code> call, a Rx thread is created. Use <code>xILinSetChannelParams()</code> in order to initialize the channels (like master/slave and the baud rate). It is also recommended to set up the LIN dlc with <code>xILinSetDLC()</code>.> linInitMaster
In order to use the LIN bus, it is necessary to define the specific DLC for each LIN ID. → <code>xILinSetDLC()</code>. This must be done only for a LIN master and before you go 'onBus'.> linInitSlave
Use <code>xILinSetSlave()</code> to set up slave. Before you go 'onBus' it is needed to define the LIN slave ID that cannot be changed after <code>xIActivateChannel()</code>. 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 <code>xILinSendRequest()</code>.> LINCclose
When all is done, the port is closed with <code>xIClosePort()</code>. |
|--------------------|--|

7 D/A IO Commands (IOcab)

In this chapter you find the following information:

7.1 Introduction	115
7.2 Flowchart	116
7.3 Functions	117
7.4 Events	124
7.5 Application Examples	126

7.1 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 25), 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.

7.2 Flowchart

Calling sequence

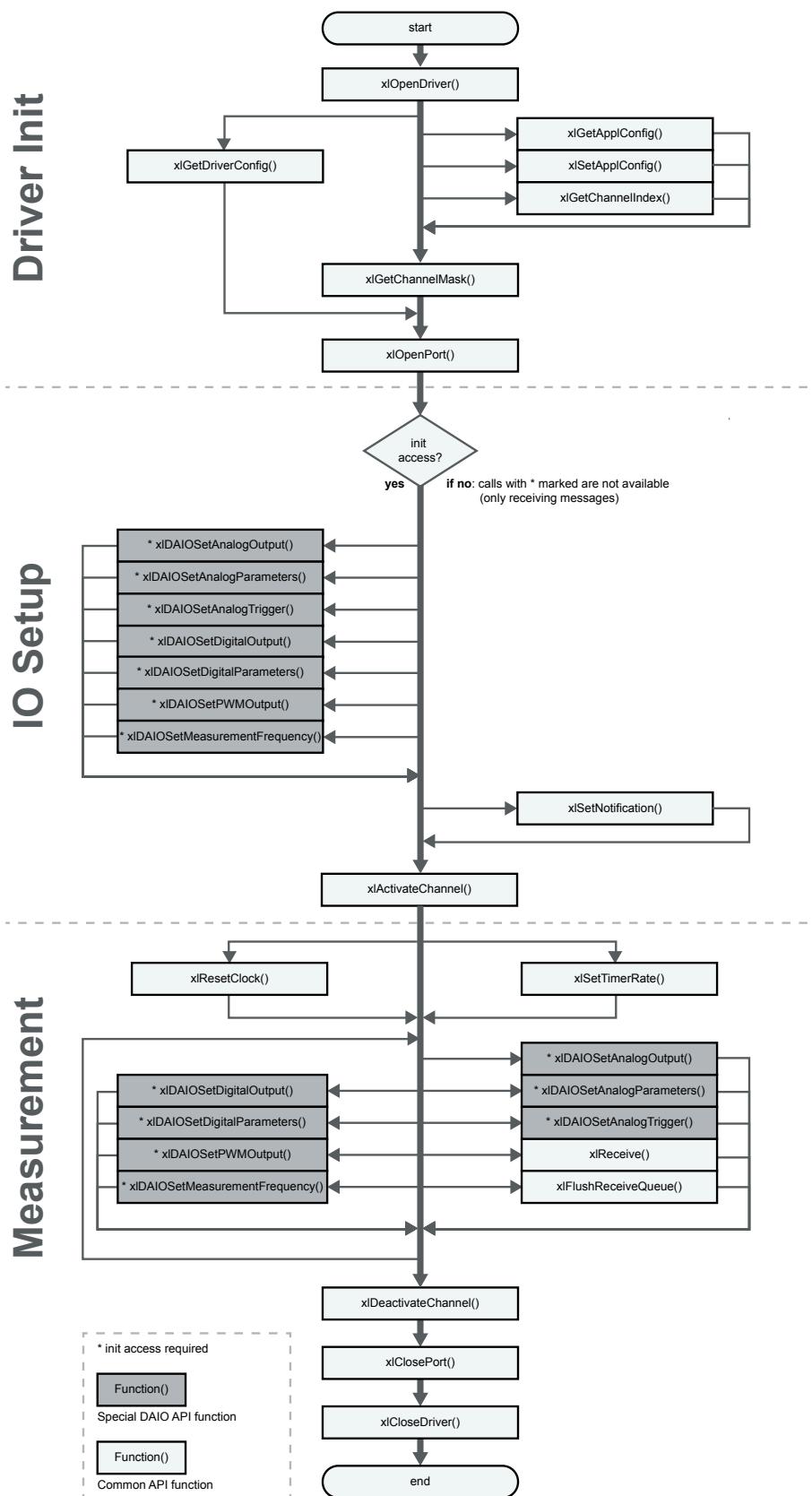


Figure 12: Function calls for DAIO applications

7.3 Functions

7.3.1 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **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 423).

**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)
```

7.3.2 xIDAOSetAnalogOutput

Syntax

```
XLstatus xIDAOSetAnalogOutput (
    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 `xIOpenPort()`.

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xIGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> 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 423).
--------------	--

7.3.3 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. Typically, the access mask can be directly retrieved from the [Vector Hardware Configuration](#) tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).
--

7.3.4 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: 0b00010000
- > DAIO5: 0b00100000
- > DAIO6: 0b01000000
- > DAIO7: 0b10000000

Input parameters

- > **portHandle**
The port handle retrieved by `xIOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xIGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **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.

**Caution!**

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 423).

7.3.5 xIDAOSetDigitalOutput

Syntax

```
XLstatus xIDAOSetDigitalOutput (
    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 `xIOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **outputMask**

Switches to be changed:

- DAIO0/DAIO1: 0b0001
- DAIO2/DAIO3: 0b0010
- DAIO4/DAIO5: 0b0100
- DAIO6/DAIO7: 0b1000

> **valuePattern**

Mask specifying the switch state for digital output.

- DAIO0/DAIO1: 0b000x
- DAIO2/DAIO3: 0b00x0
- DAIO4/DAIO5: 0b0x00
- DAIO6/DAIO7: 0bx000

x = 0 (switch opened) or 1 (switch closed)

Return value

Returns an error code (see section [Error Codes](#) on page 423).

**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)
```

7.3.6 xIDAIOSetPWMOutput

Syntax

```
XLstatus xIDAIOSetPWMOutput (
    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 [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).

**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
```

7.3.7 xIDAIOSetMeasurementFrequency

Syntax

```
XLstatus xIDAIOSetMeasurementFrequency (
    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 [xIDAIOResponseMeasurement](#) on page 122.

Input parameters> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **measurementInterval**

Measurement frequency in ms.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

7.3.8 xIDAIOResponseMeasurement

Syntax

```
XLstatus xIDAIOResponseMeasurement (
    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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

Return value

Returns an error code (see section `Error Codes` on page 423).

7.4 Events

7.4.1 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 59).

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 subtract 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.

7.5 Application Examples

7.5.1 xIDAOexample

7.5.1.1 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.

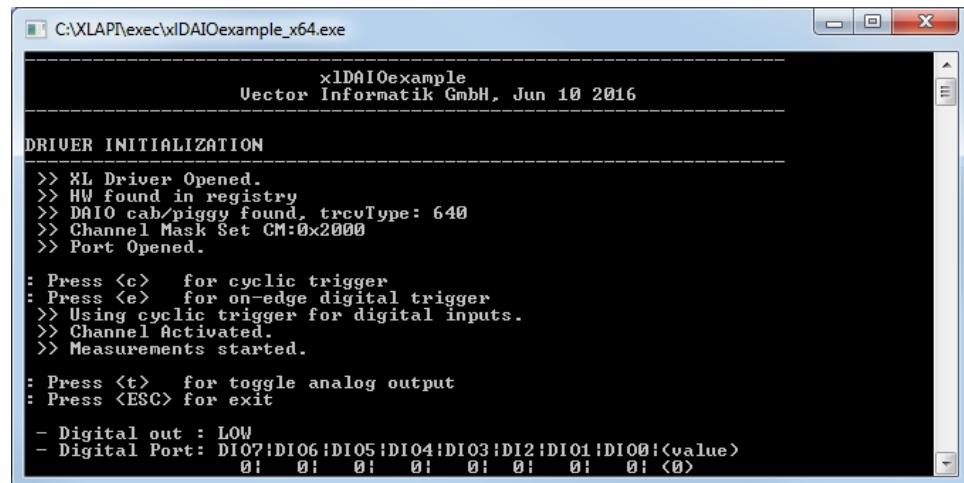


Figure 13: Running xIDAOexample

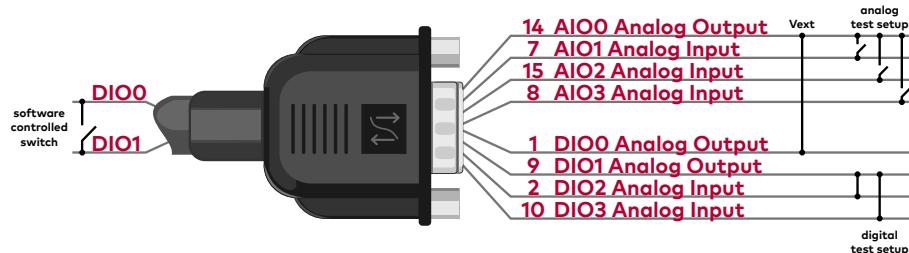
7.5.1.2 Setup

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 `xIDAIOSetDigitalOutput()`. If the switch is closed, the applied voltage at DIO0 can be measured at DIO1.

7.5.1.3 Keyboard commands

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

Key	Command
<ENTER>	Toggle digital output.
<x>	Closes application.

7.5.1.4 Output Examples

**Example**

```
AIO0:          4032mV
AIO1:          0mV
AIO2:          0mV
AIO3:          0mV
Switch selected: DIO0/DIO1
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 to 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/DIO1
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 to 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.)

7.5.1.5 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.

7.5.2 xIDAIdemo

7.5.2.1 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.

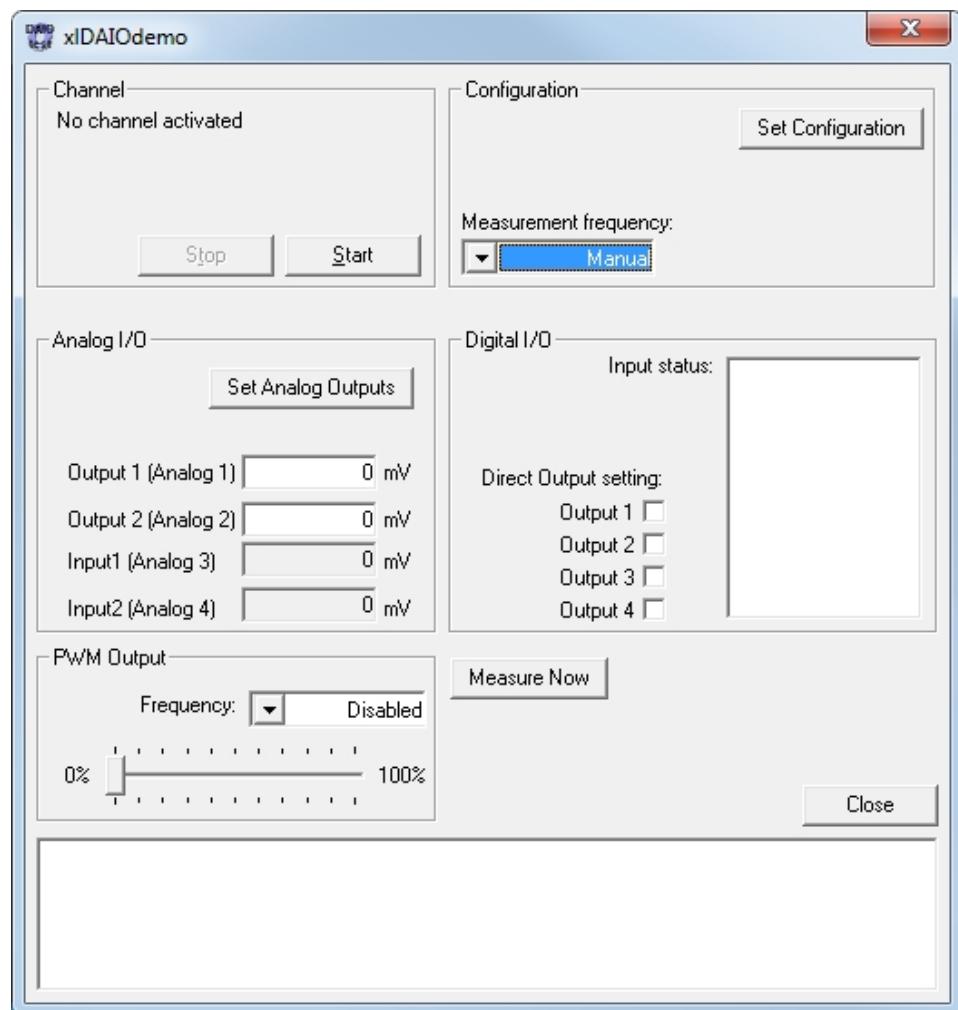


Figure 14: xIDAIDemo

7.5.2.2 Classes

Description

The example has the following class structure:

- > **CXIDAIDemoApp**
Main MFC class → xIDAIDemo.cpp
- > **CXIDAIDemoDig**
Handles the window dialog messages and control the IOcab → xIDAIDemoDlg.cpp
- > **ReceiveThread**
Thread to handle the DAIO events.

8 D/A IO Commands (IOpiggy)

In this chapter you find the following information:

8.1 Introduction	131
8.2 Flowchart	132
8.3 Functions	133
8.4 Structs	137
8.5 Events	142

8.1 Introduction

Description

The **XL Driver Library** enables the development of DAIO applications for the Vector IOpiggy 8642.

Depending on the channel property **init access** (see page 25), 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.

8.2 Flowchart

Calling sequence

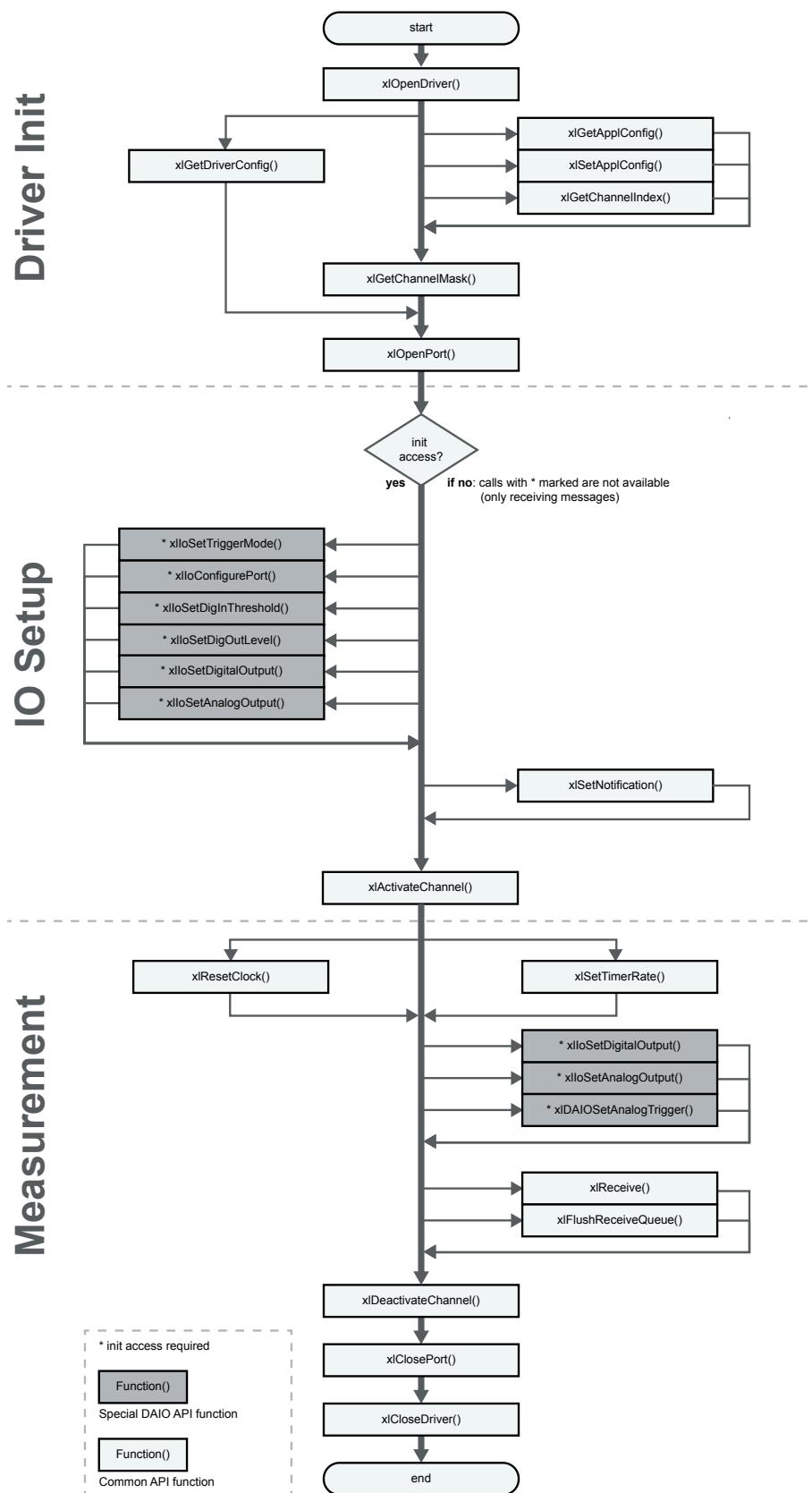


Figure 15: Function calls for DAIO (IOpiggy) applications

8.3 Functions

8.3.1 xlIoSetTriggerMode (IOpiggy)

Syntax

```
XLstatus xlIoSetTriggerMode (
    XLportHandle      portHandle,
    XLaccess          accessMask,
    XLdaioTriggerMode* pxDaioTriggerMode)
```

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 section [Introduction](#) on page 145).

Input parameters

- > **portHandle**

The port handle retrieved by [xiOpenPort\(\)](#).

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xiGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **pxlDaioTriggerMode**

Use this structure to define the trigger type (see section [XLdaioTriggerMode](#) on page 137).

Return value

Returns an error code (see section [Error Codes](#) on page 423).

8.3.2 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 [xiOpenPort\(\)](#).

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xiGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **pxlDaioSetPort**

Port configuration (see section [XLdaioDigitalParams](#) on page 138).

Return value

Returns an error code (see section [Error Codes](#) on page 423).

8.3.3 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **level**

10 bit value that defines the voltage level (mV) for the input threshold.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

8.3.4 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **level**

`XL_DAIO_DO_LEVEL_0V`
`XL_DAIO_DO_LEVEL_5V`
`XL_DAIO_DO_LEVEL_12V`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

8.3.5 xlIoSetDigitalOutput

Syntax	<pre>XLstatus xlIoSetDigitalOutput (XLportHandle portHandle, XLaccess accessMask, XLdaioDigitalParams* pxlDaioDigitalParams)</pre>
Description	Configures the digital output.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by xlOpenPort(). > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section xlGetChannelMask on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23. > pxlDaioDigitalParams Use this structure to set the value of the digital out pin (see section XLdaioDigitalParams (IOpiggy) on page 140).
Return value	Returns an error code (see section Error Codes on page 423).

8.3.6 xlIoSetAnalogOutput

Syntax	<pre>XLstatus xlIoSetAnalogOutput (XLportHandle portHandle, XLaccess accessMask, XLdaioAnalogParams* pxlDaioAnalogParams)</pre>
Description	Configures the analog output.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by xlOpenPort(). > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section xlGetChannelMask on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23. > pxlDaioAnalogParams Use this structure to set the value of the analog out pin (see section XLdaioAnalogParams on page 140).
Return value	Returns an error code (see section Error Codes on page 423).

8.3.7 xlIoStartSampling

Syntax	<pre>XLstatus xlIoStartSampling (XLportHandle portHandle, XLaccess accessMask, unsigned int portTypeMask)</pre>
--------	--

Description	This command requests DAIO measurement data and is independent of the defined trigger mode.
Input parameters	<ul style="list-style-type: none">> portHandle The port handle retrieved by <code>xlOpenPort()</code>.> accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23.> portTypeMask <code>XL_DAIO_PORT_TYPE_MASK_ANALOG</code> <code>XL_DAIO_PORT_TYPE_MASK_DIGITAL</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

8.4 Structs

8.4.1 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 use with `XL_DAIO_TRIGGER_TYPE_CYCLIC`.

Cyclic trigger time in μs (1000...1048575).

The specified cycle time guarantees the minimum interval in which events will be fired. 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`

> **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);

```

8.4.2 XLdaioDigitalParams

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...D7):

```
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_PUSH_PULL  
XL_DAIO_PORT_DIGITAL_IN
```

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.



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, &confDaioPortsDig);

```

8.4.3 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

8.4.4 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).

8.5 Events

8.5.1 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 147](#).

> **IOpiggy**

[xlioSetTriggerMode \(IOpiggy\) on page 133](#).

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 151](#) and section [XL IO Analog Data on page 150](#).



Example

Checking digital port D0

```
if (ev.daioData.digital.digitalInputData &
    XL_DAIO_PORT_MASK_DIGITAL_D0) {...}
```

8.5.2 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.
- > **measuredAnalogData0**
Third analog port that is defined as an input.
This value is 0 for differential input.
- > **measuredAnalogData0**
Fourth analog port that is defined as an input.
This value is 0 for differential input.

8.5.3 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.

9 D/A IO Commands (VN1600)

In this chapter you find the following information:

9.1 Introduction	145
9.2 Flowchart	146
9.3 Functions	147
9.4 Structs	149
9.5 Events	150

9.1 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 25), 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.

9.2 Flowchart

Calling sequence

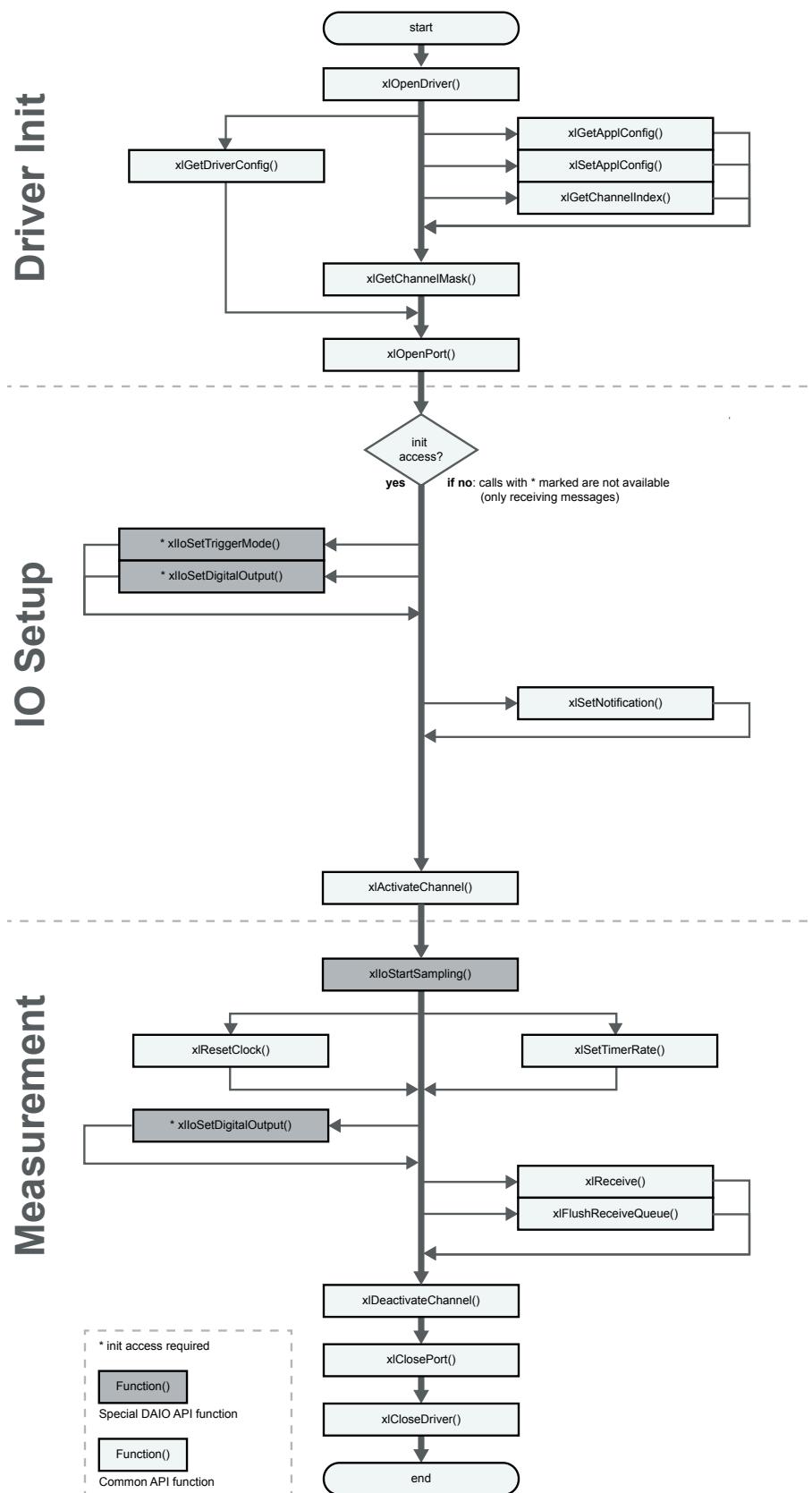


Figure 16: Function calls for DAIO (VN1600) applications

9.3 Functions

9.3.1 xlIoSetTriggerMode (VN1600)

Syntax

```
XLstatus xlIoSetTriggerMode (
    XLportHandle      portHandle,
    XLaccess          accessMask,
    XLdaioTriggerMode* pxDaioTriggerMode)
```

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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **pxlDaioTriggerMode**

Use this structure to define the trigger type (see section [XLdaioTriggerMode](#) on page 137).

Note: Currently only `XL_DAIO_TRIGGER_TYPE_CYCLIC` is supported.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

9.3.2 xlIoSetDigitalOutput

Syntax

```
XLstatus xlIoSetDigitalOutput (
    XLportHandle      portHandle,
    XLaccess          accessMask,
    XLdaioDigitalParams* pxDaioDigitalParams)
```

Description

Configures the digital output.

Input parameters

- > **portHandle**

The port handle retrieved by xlOpenPort().

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **pxlDaioDigitalParams**

Use this structure to set the value of the digital out pin (see section [XLdaioDigitalParams \(VN1600\)](#) on page 149).

Return value

Returns an error code (see section [Error Codes](#) on page 423).

9.4 Structs

9.4.1 XLdaioDigitalParams (VN1600)

Syntax

```
typedef struct xl_daio_digital_params{  
    unsigned int portMask;  
    unsigned int valueMask;  
} XLdaioDigitalParams;
```

Parameters

> **portMask**

Only XL_DAIO_PORT_MASK_DIGITAL_D0 is available.

> **valueMask**

Specifies the port value:

ON/HIGH: 1

OFF/LOW: 0

9.5 Events

9.5.1 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 147](#).

> **IOpiggy**

`xlioSetTriggerMode (IOpiggy)` on page 133.

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 151 and section [XL IO Analog Data](#) on page 150.



Example

Checking digital port D0

```
if (ev.daioData.digital.digitalInputData &
    XL_DAIO_PORT_MASK_DIGITAL_D0) {...}
```

9.5.2 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.
- > **measuredAnalogData0**
Third analog port that is defined as an input.
This value is 0 for differential input.
- > **measuredAnalogData0**
Fourth analog port that is defined as an input.
This value is 0 for differential input.

9.5.3 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.

10 MOST Commands

In this chapter you find the following information:

10.1 Introduction	153
10.2 Flowchart	154
10.3 MOST Analysis Library and Node Functions	156
10.4 Specific OS8104 Registers	159
10.5 Functions	160
10.6 Structs	192
10.7 Events	195
10.8 Application Examples	215

10.1 Introduction

Description

The **XL Driver Library** enables the development of MOST applications for supported Vector devices (see section System Requirements on page 28). A MOST application always requires **init access**(see section `xlOpenPort` on page 37)multiple MOST applications cannot use a common physical MOST channel at the same time.

Depending on the channel property **init access** (see page 25), the application's main features are as follows:

With init access

- > channel parameters can be changed/configured
- > MOST frames can be transmitted on the channel
- > MOST frames 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.

Generally, the VN2600 interface family can be parametrized without activating the channel. However, it is recommended to activate the channel before, otherwise the responding events are not recognized. To address the event to the corresponding function call, a user handle within the event is available. If the `userHandle` is non zero the event is a response to a function call, otherwise it is a message or state change event. The `userHandle` can be set up on function call and returns on the responding event.

Reset of VN2600 interface family

When the VN2610/VN2640 interface is plugged in, the following default values are set for a MOST node:

frequency	44.1 kHz
Node address	0xFFFF
Group address	0x300
Alternate packet address	0xFFF

10.2 Flowchart

Calling sequence

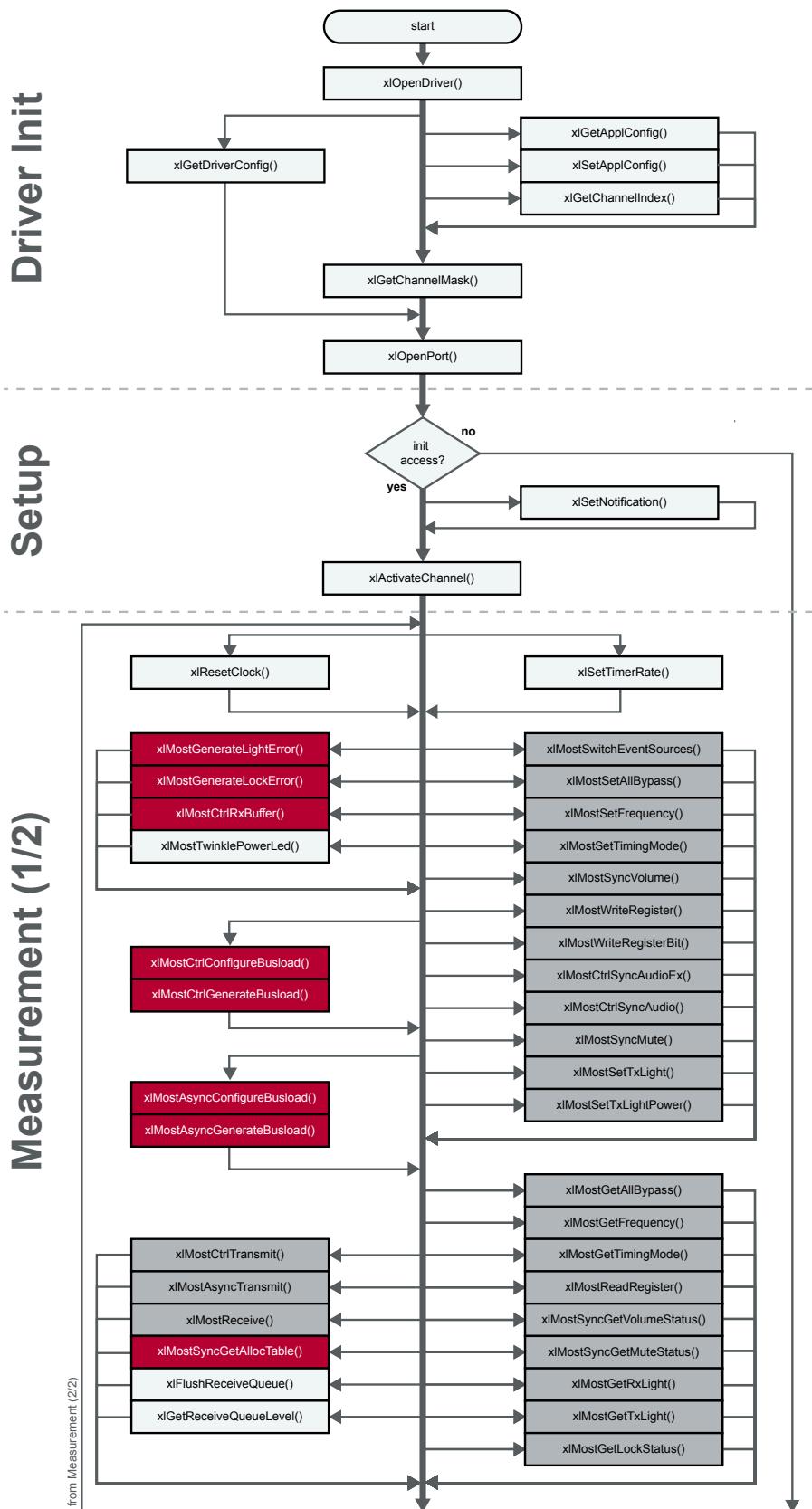


Figure 17: Function calls for MOST applications (1/2)

Calling sequence

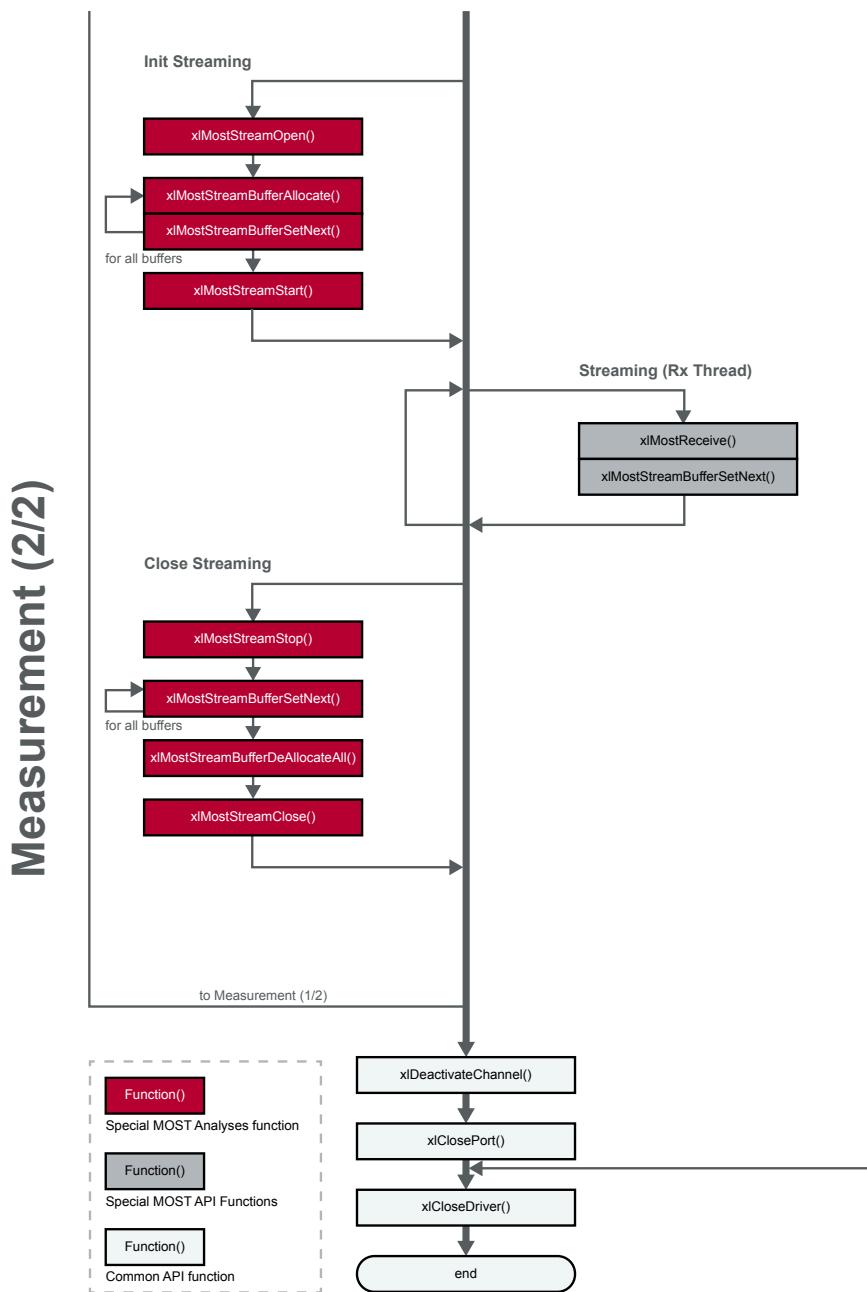


Figure 18: Function calls for MOST applications (2/2)

10.3 MOST Analysis Library and Node Functions

Administration/
configuration

MOST Command	Node Function	MOST Analysis Lib
xIMostSwitchEventSources	Limited*	X

Node
Configuration

MOST Command	Node Function	MOST Analysis Lib
xIMostSetAllBypass	X	X
xIMostGetAllBypass	X	X
xIMostSetTimingMode	X	X
xIMostGetTimingMode	X	X
xIMostSetFrequency	X	X
xIMostGetFrequency	-	X
xIMostWriteRegister	X	X
xIMostReadRegister	X	X
xIMostWriteRegisterBit	X	X

Messages

MOST Command	Node Function	MOST Analysis Lib
xIMostCtrlTransmit	X	X
xIMostAsyncTransmit	X	X

Channel allocation

MOST Command	Node Function	MOST Analysis Lib
xIMostSyncGetAllocTable	-	X

Synchronous
channel I/O

MOST Command	Node Function	MOST Analysis Lib
xIMostCtrlSyncAudio	X	X
xIMostSyncVolume	X	X
xIMostSyncGetVolumeStatus	X	X
xIMostSyncMute	X	X
xIMostSyncGetMuteStatus	X	X

Optical interface

MOST Command	Node Function	MOST Analysis Lib
xIMostGetRxLight	X	X
xIMostSetTxLight	X	X
xIMostGetTxLight	X	X
xIMostSetTxLightPower	X	X
xIMostGetLockStatus	X	X

General tester

MOST Command	Node Function	MOST Analysis Lib
xIMostGenerateLightError	-	X
xIMostGenerateLockError	-	X
xIMostCtrlRxBuffer	X	X
xIMostAsyncConfigureBusload	-	X

General tester

MOST Command	Node Function	MOST Analysis Lib
xIMostAsyncGenerateBusload	-	X
xIMostCtrlConfigureBusload	-	X
xIMostCtrlGenerateBusload	-	X

Streaming
MOST commands

MOST Command	Node Function	MOST Analysis Lib
xIMostStreamOpen	-	X
xIMostStreamClose	-	X
xIMostStreamStart	-	X
xIMostStreamStop	-	X
xIMostStreamBufferAllocate	-	X
xIMostStreamBufferDeallocateAll	-	X
xIMostStreamBufferSetNext	-	X
xIMostStreamGetInfo	-	X
xIMostStreamBufferClearAll	-	X

General
MOST commands

MOST Command	Node Function	MOST Analysis Lib
xIGenerateSyncPulse	-	X
xIMostReceive	Limited*	X
xIMostTwinklePowerLed	X	X

Possible Rx events
(for xIMostReceive)

MOST Command	Node Function	MOST Analysis Lib
XL_MOST_START	X	X
XL_MOST_STOP	X	X
XL_MOST_EVENTSOURCES	Limited*	X
XL_TIMER	X	X
XL_SYNC_PULSE	-	X
XL_MOST_ALLBYPASS	X	X
XL_MOST_TIMINGMODE	X	X
XL_MOST_FREQUENCY	X	X
XL_MOST_REGISTER_BYTES	X	X
XL_MOST_REGISTER_BITS	X	X
XL_MOST_SPECIAL_REGISTER	X	X
XL_MOST_CTRL_RX_SPY	-	X
XL_MOST_CTRL_RX_OS8104	X	X
XL_MOST_CTRL_TX	X	X
XL_MOST_ASYNC_MSG	X	X
XL_MOST_ASYNC_TX	X	X
XL_MOST_SYNC_VOLUME_STATUS	X	X
XL_MOST_RXLIGHT	X	X
XL_MOST_TXLIGHT	X	X
XL_MOST_LOCKSTATUS	X	X
XL_MOST_ERROR	X	X
XL_MOST_CTRL_RXBUFFER	X	X
XL_MOST_CTRL_SYNC_AUDIO	X	X

Possible Rx events
(for xlMostReceive)

MOST Command	Node Function	MOST Analysis Lib
XL_MOST_SYNC_MUTE_STATUS	X	X
XL_MOST_GENLIGHTERROR	-	X
XL_MOST_GENLOCKERROR	-	X
XL_MOST_TXLIGHT_POWER	X	X
XL_MOST_CTRL_BUSLOAD	-	X
XL_MOST_ASYNC_BUSLOAD	-	X
XL_MOST_XL_MOST_STREAM_STATE	-	X
XL_MOST_STREAM_BUFFER	-	X

* No control spy events, no asynchronous spy events, no allocation table events, no synchronous events.

10.4 Specific OS8104 Registers

Map	Reg	XL API Def	Description	Byte	Acc
0x8A	bNAH bNAL	XL_MOST_bNAH XL_MOST_bNAL	Logical Node address high byte/low byte.	2	r/w
0x89	bGA	XL_MOST_bGA	Group address.	1	r/w
0xE8	bAPAH bAPAL		Alternate Packet Address High/Low byte. This value cannot be the same as NAH, NAL.	2	r/w
0x87	bNPR		Node Position Register. Reports physical position of a node, relative to the Network timingmaster.	1	r
0x90	bMPR	XL_MOST_bMPR	Maximum Position Register. Reports total number of active nodes in the Network.	1	r
0x8F	bNDR	XL_MOST_bNDR	Node Delay Register. Reports source data delay between timing-master and local node.	1	r
0x91	bMDR	XL_MOST_bMDR	Maximum Delay Register. Reports total synchronous data delay in the Network.	1	r
0x96	bSBC	XL_MOST_bSBC	Synchronous Bandwidth Control. Controls the number of bytes used for synchronous data transfer vs. the number of bytes used for asynchronous packet data transfer.	1	r/w
0xBE	bXTIM	XL_MOST_bXTIM	Transmit Retry Time Register	1	r/w
0xBF	bXRTY	XL_MOST_bXRTY	Transmit Retry Register. Retry time = <Time Unit> × bXTIM The time units are approximately: 421 µs at Fs = 38 kHz 363 µs at Fs = 44.1 kHz 333 µs at Fs = 48 kHz	1	r/w

10.5 Functions

10.5.1 xlMostSwitchEventSources

Syntax

```
XLstatus xlMostSwitchEventSources (
    XLportHandle    portHandle,
    XLaccess        accessMask,
    XLuserHandle    userHandle,
    unsigned short  sourceMask)
```

Description

Switches the different MOST events (like asynchronous or control frames) depending on the license on/off. Events from closed channels are not transmitted to the PC.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **sourceMask**

This flag describes the switched events (event will be passed when bit is set).

Free Library:

`XL_MOST_SOURCE_ASYNC_RX`

Switch on the `XL_MOST_ASYNC_MSG` events.

`XL_MOST_SOURCE_ASYNC_TX`

Switch on the `XL_MOST_ASYNC_TX` events.

`XL_MOST_SOURCE_CTRL_OS8104A`

Switch on the `XL_MOST_CTRL_RX_OS8104` events.

`XL_MOST_SOURCE_ASYNC_RX_FIFO_OVER`

Switch on the `XL_MOST_ERROR` events with

`errorCode XL_MOST_ASYNC_TYPE_QUEUE_OVERFLOW`.

MOST Analysis Library:

`XL_MOST_SOURCE_ASYNC_RX`

Switch on the `XL_MOST_ASYNC_MSG` events.

`XL_MOST_SOURCE_ASYNC_TX`

Switch on the `XL_MOST_ASYNC_TX` events.

`XL_MOST_SOURCE_CTRL_OS8104A`

Switch on the `XL_MOST_CTRL_RX_OS8104` events.

`XL_MOST_SOURCE_ASYNC_RX_FIFO_OVER`

Switch on the `XL_MOST_ERROR` events with

`errorCode XL_MOST_ASYNC_TYPE_QUEUE_OVERFLOW`.

`XL_MOST_SOURCE_CTRL_SPY`

Switch on the `XL_MOST_CTRL_RX_SPY` events.

`XL_MOST_SOURCE_ASYNC_SPY`

Switch on the `XL_MOST_ASYNC_MSG` events with `flagsChip XL_MOST_SPY`

`XL_MOST_SOURCE_SYNCLINE`

Switch on the `XL_SYNC_PULSE` events.

Return event

`XL_MOST_EVENTSOURCES`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

10.5.2 xIMostSetAllBypass

Syntax

```
XLstatus xlMostSetAllBypass(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned char bypassMode)
```

Description

Opens/closes the bypass functionality.

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23. > userHandle The handle is created by the application and is used for the event assignment. > bypassMode XL_MOST_MODE_DEACTIVATE Bypass deactivated. XL_MOST_MODE_ACTIVATE Bypass activated.
Return event	<code>XL_MOST_ALLBYPASS</code>
Return value	Returns an error code (see section Error Codes on page 423).

10.5.3 `xlMostGetAllBypass`

Syntax	<pre>XLstatus xlMostGetAllBypass (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle)</pre>
Description	Gets the bypass mode.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23. > userHandle The handle is created by the application and is used for the event assignment.
Return event	<code>XL_MOST_ALLBYPASS</code>
Return value	Returns an error code (see section Error Codes on page 423).

10.5.4 `xlMostSetTimingMode`

Syntax	<pre>XLstatus xlMostSetTimingMode (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle,</pre>
---------------	--

```
unsigned char timingMode)
```

Description	Sets the timing mode between master/slave.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > timingMode Describes the timing mode. SPDIF timing modes are only available for VN2610/VN2640.
	<pre>XL_MOST_TIMING_SLAVE XL_MOST_TIMING_MASTER XL_MOST_TIMING_SLAVE_SPDIF_MASTER XL_MOST_TIMING_SLAVE_SPDIF_SLAVE XL_MOST_TIMING_MASTER_SPDIF_MASTER XL_MOST_TIMING_MASTER_SPDIF_SLAVE XL_MOST_TIMING_MASTER_FROM_SPDIF_SLAVE</pre>
Return event	<code>XL_MOST_TIMINGMODE, XL_MOST_TIMINGMODE_SPDIF</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

10.5.5 `xlMostGetTimingMode`

Syntax	<pre>XLstatus xlMostGetTimingMode(XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle)</pre>
Description	Gets the timing mode (timing master/ timing slave).
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment.
Return event	<code>XL_MOST_TIMINGMODE, XL_MOST_TIMINGMODE_SPDIF</code>

Return value	Returns an error code (see section Error Codes on page 423).
---------------------	--

10.5.6 xlMostSetFrequency

Syntax

```
XLstatus xlMostSetFrequency(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned short frequency)
```

Description

Sets the frame rate of the MOST network for a timing master. The setting will be active when:

- > bypass is opened
- > from slave to master mode is switched or
- > measurement is started

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **frequency**

Frame rate in kHz.

XL_MOST_FREQUENCY_44100
44.1 kHz

XL_MOST_FREQUENCY_48000
48 kHz

Return event

XL_MOST_FREQUENCY

Return value

Returns an error code (see section [Error Codes](#) on page 423).

10.5.7 xlMostGetFrequency



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
XLstatus xlMostGetFrequency(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle)
```

Description

Acquires the frame rate of the MOST network (timing slave) or returns the frame rate

of the timing master.

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment.
Return event	<code>XL_MOST_FREQUENCY</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

10.5.8 `xIMostWriteRegister`

Syntax	<pre>XLstatus xlMostWriteRegister(XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle, unsigned short adr, unsigned char numBytes, unsigned char data[16])</pre>
Description	Writes up to 16 register values of a hardware chip and returns a write confirmation. Refer also to <code>xIMostWriteSpecialRegister()</code> .
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > adr Register address (see section <code>Specific OS8104 Registers</code> on page 159). > numBytes Number of bytes. > data[16] Register values.
Return event	<code>XL_MOST_REGISTER_BYTES</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

**Example****Group setup to address 0x0300**

```
data[0] = 0x00;
xlStatus = xlMostWriteRegister(m_XLportHandle[nChan],
                                m_xlChannelMask[nChan],
                                0,
                                XL_MOST_bGA,
                                1,
                                data);
```

10.5.9 xlMostReadRegister

Syntax

```
XLstatus xlMostReadRegister(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned short adr,
    unsigned char numBytes)
```

Description

Reads up to 16 register values of a hardware chip (OS8104).

Input parameters**> portHandle**

The port handle retrieved by `xiOpenPort()`.

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xiGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> userHandle

The handle is created by the application and is used for the event assignment.

> adr

Register address(see section `Specific OS8104 Registers` on page 159).

> numBytes

Number of bytes.

Return event

`XL_MOST_REGISTER_BYTES`

Return value

Returns an error code (see section `Error Codes` on page 423).

10.5.10 xlMostWriteRegisterBit

Syntax

```
XLstatus xlMostWriteRegisterBit(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned short adr,
    unsigned char mask,
    unsigned char value)
```

Description

Writes single bits of a register byte, e. g. to change the Source Data Control Register or to mute Source Data Outputs.

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > adr Register address (see section <code>Specific OS8104 Registers</code> on page 159). > mask Bit mask. > Value Register value.
Return event	<code>XL_MOST_REGISTER_BITS</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

10.5.11 `xlMostCtrlTransmit`

Syntax

```
XLstatus xlMostCtrlTransmit(
    XLportHandle    portHandle,
    XLaccess        accessMask,
    XLuserHandle    userHandle,
    XLmostCtrlMsg   *pCtrlMsg)
```

Description

Transmits a message over the control channel. The transmit confirmation is reported as `XL_MOST_CTRL_MSG` when the MOST chip displays the receiving or not-receiving.



Note

The transmit confirmation does not need contain the same data bytes as in the sent request (see system properties: `RemoteRead`, `RemoteWrite`, `Alloc`, `Dealloc`, `GetSource`).

The Tx confirmation should return the data bytes as well as the handle in order to prepare the multi-use of the driver dll by more than one application.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **pCtrlMsg**
See section [XL_MOST_CTRL_MSG_EV](#) on page 203 (structure `s_xl_most_ctrl_msg`).

Return event `XL_MOST_CTRL_TX`

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.12 xlMostAsyncTransmit

Syntax

```
XLstatus xlMostAsyncTransmit(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    XLmostAsyncMsg *pAsyncMsg)
```

Description

Transmits a message over the asynchronous channel and returns the point of time of transmission as confirmation. The transmit confirmation in case of asynchronous messages means that the message was sent to the bus, but not that the data has been correctly received.

In the first step, the confirmation with all data bytes is created in the firmware and is handed over to the application.

Input parameters

- > **portHandle**
The port handle retrieved by [xIOpenPort\(\)](#).
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **pAsyncMsg**
See section [XL_MOST_ASYNC_TX_EV](#) on page 205 (structure (`s_xl_most_async_tx`)).

Return event `XL_MOST_ASYNC_MSG`

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.13 xlMostSyncGetAllocTable



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
XLstatus xlMostSyncGetAllocTable(
```

```
XLportHandle portHandle,
XLaccess accessMask,
XLuserHandle userHandle)
```

Description Requests allocation table for synchronous channels.
OS8104: Register 0x380...0x3BB.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.

Return event `XL_MOST_SYNC_ALLOCABLE`

Return value Returns an error code (see section `Error Codes` on page 423).

10.5.14 `xlMostCtrlSyncAudio`

Syntax

```
XLstatus xlMostCtrlSyncAudio(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int channel[4],
    unsigned int device,
    unsigned int mode)
```

Description Defines the channels for synchronous input/output. The channel routing is done after this function call, therefore the firmware programs the routing engine according to OS8104.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **channel**
Contains the channel numbers for the synchronous data (LMSB, LLSB, RMSB, RLSB).
- > **device**
`XL_MOST_DEVICE_CASE_LINE_IN`
`XL_MOST_DEVICE_CASE_LINE_OUT`

> **mode**

Line in

1 (on): reprogramming the routing engine (RE), that the AD converted values are assigned to the according MOST channels (uncared for the allocation).

0 (off): programming RE in that way the switch on state is set for the port (no data is send to the ring by the port)

Line out

1 (on): reprogramming RE, that the DA converted values are assigned to the according MOST channels (uncared for the allocation); Insertion of channel number at the fitting places in the RE. If not inserted yet, the control registers bSDC1...bSDC3 are set.

0 (off): programming RE in that way the switch on state is set for the out port (mute value inserted in fitting place in the RE. Reset of control registers if necessary).

Return event XL_MOST_CTRL_SYNC_AUDIO**Return value** Returns an error code (see section [Error Codes](#) on page 423).

10.5.15 xlMostCtrlSyncAudioEx

Syntax

```
XLstatus xlMostCtrlSyncAudioEx (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int channel[16],
    unsigned int device,
    unsigned int mode)
```

Description

Defines the channels for synchronous input/output including SPDIF. Whereas the SPDIF functionality is only available on the VN2610/VN2640. The channel routing is done after this function call, therefore the firmware programs the routing engine according to OS8104.

Input parameters> **portHandle**The port handle retrieved by [xlOpenPort\(\)](#).> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the [Vector Hardware Configuration](#) tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **channel**

Contains the channel numbers for the synchronous data (LMSB, LLSB, RMSB, RLSB).

> **device**

`XL_MOST_DEVICE_CASE_LINE_IN`
Selects as device line in.

`XL_MOST_DEVICE_CASE_LINE_OUT`
Selects as device line out.

`XL_MOST_DEVICE_SPDIF_IN`
Selects as device SPDIF in (only VN2610).

`XL_MOST_DEVICE_SPDIF_OUT`
Selects as device SPDIF out (only VN2610).

`XL_MOST_DEVICE_SPDIF_IN_OUT_SYNC`
Synchronizes the SPDIF in/out (only VN2610).

> **mode**

Line in

1 (on): reprogramming RE, that the AD converted values are assigned to the according MOST channels (uncared for the allocation).

0 (off): programming RE in that way the switch on state set for the port (no data is send to the ring by the port).

Line out

1 (on): reprogramming RE, that the DA converted values are assigned to the according MOST channels (uncared for the allocation); Insertion of channel number at the fitting places in the RE. If not inserted yet, the control registers bSDC1...bSDC3 are set.

0 (off): programming RE in that way the switch on state is set for the out port (mute value inserted in fitting place in the RE. Reset of control registers if necessary).

`XL_MOST_SPDIF_LOCK_OFF`

Switches off the SPDIF synchronization.

`XL_MOST_SPDIF_LOCK_ON`

Switches on the SPDIF synchronization.

Return event `XL_MOST_CTRL_SYNC_AUDIO_EX`

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.16 xlMostSyncVolume

Syntax

```
XLstatus xlMostSyncVolume(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int device,
    unsigned char volume)
```

Description

Defines the input gain of the device (line in / line out). 100% means maximum level, 0% minimum level (no level). The function does not work for SPDF.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **device**

XL_MOST_DEVICE_CASE_LINE_IN
XL_MOST_DEVICE_CASE_LINE_OUT

> **volume**

Value range 0...255 (means 0%...100%).

Return event XL_MOST_SYNCVOLUMESTATUS

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.17 xlMostSyncGetVolumeStatus

Syntax

```
XLstatus xlMostSyncGetVolumeStatus (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int device)
```

Description Requests the state of line in/out ports. The function does not work for SPDIF.

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **device**

XL_MOST_DEVICE_CASE_LINE_IN
XL_MOST_DEVICE_CASE_LINE_OUT

Return event XL_MOST_SYNCVOLUMESTATUS

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.18 xlMostSyncMute

Syntax

```
XLstatus xlMostSyncMute(
    XLportHandle portHandle,
```

```
XLaccess accessMask,
XLuserHandle userHandle,
unsigned int device,
unsigned char mute)
```

Description	Mute/unmutes a port. The function does not work for SPDIF.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > device <code>XL_MOST_DEVICE_CASE_LINE_IN</code> <code>XL_MOST_DEVICE_CASE_LINE_OUT</code> > mute <code>XL_MOST_NO_MUTE</code> Port not muted. <code>XL_MOST_MUTE</code> Port is muted.
Return event	<code>XL_MOST_SYNC_MUTE_STATUS</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

10.5.19 `xlMostSyncGetMuteStatus`

Syntax	<pre>XLstatus xlMostSyncGetMuteStatus (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle, unsigned int device)</pre>
Description	Requests mute state.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment.

> **device**
 XL_MOST_DEVICE_CASE_LINE_IN
 XL_MOST_DEVICE_CASE_LINE_OUT

Return event XL_MOST_SYNC_MUTE_STATUS

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.20 xlMostGetRxLight

Syntax

```
XLstatus xlMostGetRxLight (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle)
```

Description Requests light state at FOR. Forces [XL_MOST_RXLIGHT](#) event.

Input parameters

- > **portHandle**
 The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
 The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
 The handle is created by the application and is used for the event assignment.

Return event XL_MOST_RXLIGHT

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.21 xlMostSetTxLight

Syntax

```
XLstatus xlMostSetTxLight (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned char txLight)
```

Description Sets light status at FOT.

Input parameters

- > **portHandle**
 The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
 The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
 The handle is created by the application and is used for the event assignment.

- > **txLight**
 XL_MOST_LIGHT_OFF
 XL_MOST_LIGHT_FORCE_ON
 XL_MOST_LIGHT_MODULATED

Return event XL_MOST_TXLIGHT

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.22 xlMostGetTxLight

Syntax

```
XLstatus xlMostGetTxLight (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLuserHandle userHandle,
  unsigned char txlight)
```

Description Requests light status at FOT. Forces XL_MOST_TXLIGHT event.

Input parameters

- > **portHandle**
 The port handle retrieved by [xIOpenPort\(\)](#).
- > **accessMask**
 The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
 The handle is created by the application and is used for the event assignment.
- > **txLight**
 XL_MOST_LIGHT_OFF
 XL_MOST_LIGHT_FORCE_ON
 XL_MOST_LIGHT_MODULATED

Return event XL_MOST_TXLIGHT

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.23 xlMostSetLightPower

Syntax

```
XLstatus xlMostSetLightPower (
  XLportHandle portHandle,
  XLaccess accessMask,
  XLuserHandle userHandle,
  unsigned char attenuation)
```

Description Sets the attenuation of the modulated light at FOT.

Input parameters

- > **portHandle**
 The port handle retrieved by [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **attenuation**

`XL_MOST_LIGHT_FULL`
Full power.

`XL_MOST_LIGHT_3DB`
Decreased power.

Return event `XL_MOST_TXLIGHT_POWER`

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.24 `xlMostGetLockStatus`

Syntax

```
XLstatus xlMostGetLockStatus (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle)
```

Description Requests lock status of PLL (LOK bit of clock manager register 2 of OS8104). Forces an `XL_MOST_LOCKSTATUS` event.

Input parameters

> **porthandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

Return event `XL_MOST_LOCKSTATUS`

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.25 `xlMostGenerateLightError`



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
XLstatus xlMostGenerateLightError (
```

```
XLportHandle    portHandle,
XLaccess        accessMask,
XLuserHandle    userHandle,
unsigned long   lightofftime,
unsigned long   lightontime,
unsigned short  repeat)
```

Description Starts/stops the generation of light-off/on changes. Point of time of start and stop are signaled to the application by [XL_MOST_GENLIGHTERROR](#) events.

Input parameters

- > **portHandle**
The port handle retrieved by [xIOpenPort\(\)](#).
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **lightofftime**
Time of unmodulated light emission.
- > **lightontime**
Time of modulated light emission.
- > **repeat**
 - 0
Stop.
 - >0
Start.

Return event [XL_MOST_GENLIGHTERROR](#)

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.26 [xIMostGenerateLockError](#)

Syntax

```
XLstatus xIMostGenerateLockError(
    XLportHandle    portHandle,
    XLaccess        accessMask,
    XLuserHandle    userHandle,
    unsigned long   unmodtime,
    unsigned long   modtime,
    unsigned short  repeat)
```

Description Starts/stops the generation of light unmodulated/modulated changes. Point of time of start and stop are signaled to the application by [XL_MOST_GENLOCKERROR](#) events.

Input parameters

- > **portHandle**
The port handle retrieved by [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **unmodtime**

Time of unmodulated light emission.

> **Modtime**

Time of modulated light emission.

> **repeat**

0

Stop generation.

>0

Number of changes.

0xFFFF

Generation of continual changes.

Return event

XL_MOST_GENLOCKERROR

Return value

Returns an error code (see section [Error Codes](#) on page 423).

10.5.27 xIMostCtrlRxBuffer

Syntax

```
XLstatus xIMostCtrlRxBuffer (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned short bufferMode)
```

Description

Defines the event Rx event handling within the internal message queues. Per default bufferMode is on.

Input parameters

> **portHandle**

The port handle retrieved by [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

- > **bufferMode**
- 0
Off.
- 1
On, every message will be received and the buffer will be freed.
- 2
Empty once, simulated full Rx buffer.

Return event XL_MOST_CTRLRXBUFFER

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.28 xlMostCtrlConfigureBusload



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
XLstatus xlMostCtrlConfigureBusload(
    XLportHandle           portHandle,
    XLaccess                accessMask,
    XLuserHandle             userHandle,
    XLmostCtrlBusloadConfiguration *pCtrlBusloadConfiguration)
```

Description Prepares and configures busload generation with MOST control frames.

Input parameters

- > **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **userHandle**

The handle is created by the application and is used for the event assignment.

- > **pCtrlBusloadConfiguration**

Pointer to a structure containing the control message used for busload generation and configuration, its storage has to be supplied by the caller (see section [s_xl_most_ctrl_busload_configuration](#) on page 192).

Return event None.

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.29 xlMostCtrlGenerateBusload



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
XLstatus xlMostCtrlGenerateBusload(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned long numberCtrlFrames)
```

Description

Starts busload generation with MOST control frames.

Input parameters**> portHandle**

The port handle retrieved by `xlOpenPort()`.

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> userHandle

The handle is created by the application and is used for the event assignment.

> numberCtrlFrames

Number of busload control messages (0xFFFFFFFF indicates infinite number of messages).

Return event

`XL_MOST_CTRL_BUSLOAD`

Return value

Returns an error code (see section **Error Codes** on page 423).

10.5.30 `xlMostAsyncConfigureBusload`

**Note**

This feature is available in the MOST Analysis Library only.

Syntax

```
XLstatus xlMostAsyncConfigureBusload(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    XLmostCtrlBusloadConfiguration *pAsyncBusloadConfiguration)
```

Description

Prepares and configures busload generation of MOST asynchronous frames.

Input parameters**> portHandle**

The port handle retrieved by `xlOpenPort()`.

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> userHandle

The handle is created by the application and is used for the event assignment.

> **pAsyncBusloadConfiguration**

Pointer to a structure containing the asynchronous message used for busload generation and configuration, its storage has to be supplied by the caller (see section [s_xl_most_ctrl_busload_configuration](#) on page 192).

Return event None.

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.31 xlMostAsyncGenerateBusload


Note

This feature is available in the MOST Analysis Library only.

Syntax

```
XLstatus xlMostAsyncGenerateBusload(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned long numberCtrlFrames)
```

Description Starts busload generation with MOST asynchronous frames.

Input parameters
> **portHandle**

The port handle retrieved by [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **numberCtrlFrames**

Number of busload asynchronous messages (0xFFFFFFFF indicates infinite number of messages).

Return event XL_MOST_ASYNC_BUSLOAD

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.32 xlMostReceive

Syntax

```
XLstatus xlMostReceive (
    XLportHandle portHandle,
    XLmostevent *pEventBuffer)
```

Description

Reads one event from the MOST receive queue. An overrun of the receive queue can be determined by the message flag [XL_MOST_QUEUE_OVERFLOW](#) in [XLmostEvent.flagsChip](#).

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > pEventBuffer Pointer to the event buffer. Buffer size: <code>XL_MOST_EVENTBUFFER_SIZE</code>.
Return event	If the queue is empty: <code>XL_ERR_QUEUE_IS_EMPTY</code> . If the buffer within the application is too small, the function returns <code>XL_ERR_BUFFER_TOO_SMALL</code> . In this case the event contains the first 32 byte of the event header.
Return value	Returns an error code (see section Error Codes on page 423).

10.5.33 xlMostTwinklePowerLed

Syntax	<pre>XLstatus xlMostTwinklePowerLed (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle)</pre>
Description	The MOST device power LED will twinkle three times.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section xlGetChannelMask on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23. > userHandle The handle is created by the application and is used for the event assignment.
Return event	None.
Return value	Returns an error code (see section Error Codes on page 423).

10.5.34 Streaming

10.5.34.1 General Information

**Note**

This feature is available in the MOST Analysis Library only.

Streaming functions

The streaming functions of the XL MOST API can be used for transmission of data from or to synchronous MOST channels. Minimum requirements are a VN2610/VN2640 interface and USB2.0.

The streaming interface is asynchronous, i. e. the application must handle the streaming state which is reported by an `XL_MOST_STREAM_STATE` event.

**Step by Step Procedure**

1. With `xIMostStreamOpen()`, a stream-handle is opened. This one is valid only if the return value is `XL_SUCCESS`.
2. If the event `XL_MOST_STREAM_STATE` (`streamState = XL_MOST_STREAM_STATE_OPEN`) is received, the buffer(s) must be allocated with `xIMostStreamBufferAllocate()`. The return value `XL_SUCCESS` reports that the buffer has been successfully allocated (pointer `pBuffer` is valid).

**Note**

Up to ten buffers can be allocated, each with a maximum size of 4 MB. The buffer size depends on the latency setting and the options (see section `xIMostStreamOpen` on page 185). The higher the latency, the bigger each buffer will be. On Rx streaming, the buffers are MOST frame aligned.

At least two buffers should be allocated to assure a continuously data stream. It is recommended to allocate the maximum count of buffers.



3. After the buffer has been allocated, data can be stored there for Tx streaming. The buffers are given to the driver by `xIMostStreamBufferSetNext()`.
4. The stream is started with `xIMostStreamStart()`. The successful start is acknowledged with an `XL_MOST_STREAM_STATE` event (`streamState = XL_MOST_STREAM_STATE_STARTED`).
5. A processed buffer (Tx: buffer empty, Rx: buffer full) is reported by an `XL_MOST_STREAM_BUFFER` event. In case of Tx, the buffer can be refilled again. In case of Rx, the data can be written into a file. Afterwards, the buffer is given back to the driver again by `xIMostStreamBufferSetNext()`. This is repeated cyclically until the stream is stopped with `xIMostStreamStop()`.
6. A stream is stopped by `xIMostStreamStop()`. This is acknowledged with an `XL_MOST_STREAM_STATE` event (stopped). In case of Rx, the last (maybe incomplete) buffer will be reported to the application by the event `XL_MOST_STREAM_BUFFER`.
7. In order to close the stream, all buffers must be deallocated with `xIMostStreamBufferDeAllocateAll()`.
8. The stream is closed with `xIMostStreamClose()` afterwards. The stream handle is invalid at this point and cannot be used for further function calls. The closing is acknowledged with an `XL_MOST_STREAM_STATE` event (`streamState = XL_MOST_STREAM_STATE_STOPPED`).

Clear buffer

It is possible to clear all buffers of a certain stream with `xIMostStreamClearBuffers()`. This transmits '0' to the MOST ring, which can be used for muting the streams. The function call is reported to the application with the event `XL_MOST_STREAM_BUFFER`.



Note

The buffers are allocated by the driver. A parallel access of application and driver must be avoided. This means that the application may access the buffer only if the buffer was successfully allocated by `xIMostStreamBufferAllocate()` and acknowledged by the event `XL_MOST_STREAM_BUFFER`.

The application may not access the buffer after `xIMostStreamBufferSetNext()` has been called.



Note

If the application reports a filled buffer to the driver by `xIMostStreamBufferSetNext()` too late, a buffer underflow can occur. This is reported by the event `XL_MOST_SYNC_TX_UNDERFLOW` and causes routing '0' to the MOST ring.



Note

If the application reports an empty buffer to the driver by `xIMostStreamBufferSetNext()` too late, a buffer overflow can occur. This is reported by the event `XL_MOST_SYNC_RX_OVERFLOW` and incoming data from the MOST ring is lost.

10.5.34.2 Frame Format

Tx

The format of the Tx streaming data is in raw format. This means that every byte of the buffer is fed into the MOST controller in the given order. Please note that the order on the ring is also affected by the routing table of the MOST controller.

Rx raw

The format of the Rx streaming data can be in raw format. This means that every programmed byte from the MOST controller is appended to succeeding bytes. The recorded frames are in raw format when a stream with options = 0x000000001 is opened.

Rx with header

The format of the Rx streaming data can also be delivered with additional format (header).

The recorded frames contain additional data when a stream with options = 0x000000001 is opened.

In this case it has the following format:

Width	Description
64 bit	Start of frame time stamp from hardware clock; unsynchronized; in 20 ns; LSB first
2, 4, 6... 60 bytes	MOST frame data; the number of bytes depends on parameter numChannels of MostSyncStrmOpen; for odd values of numChannels a fill byte (0xFB) is inserted
8 bit	Reserved
4 bit	SBC (mask: 0b11110000)
1 bit	Light status (mask: 0b000001000)
1 bit	Lock status (mask: 0b000000100)
1 bit	Overflow flag (mask: 0b000000010)
1 bit	Underflow flag (mask: 0b000000001)

10.5.35 xlMostStreamOpen

Syntax

```
XLstatus xlMostStreamOpen(
    XLportHandle      portHandle,
    XLaccessMask     accessMask,
    XLuserHandle     userHandle,
    XLmostStreamOpen* pStreamOpen)
```

Description

Defines an input or output stream for synchronous MOST data. Only USB 2.0 is supported. USB 1.x returns an error.

Input parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **pStreamOpen**

Points to the `XLmostStreamOpen` structure which contains the streaming parameters.

Return event

`XL_MOST_STREAM_STATE` (state = open).

Return value	Returns an error code (see section Error Codes on page 423).
--------------	--

10.5.36 xlMostStreamClose

Syntax	<pre>XLstatus xlMostStreamClose(XLportHandle portHandle, XLaccessMask accessMask, XLuserHandle userHandle, unsigned int streamHandle)</pre>
Description	Closes the stream. If any buffer was allocated before by calling xlMostStreamBufferAllocate() , it has to be released before closing the stream by calling xlMostStreamBufferDeallocateAll() .
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by xIOpenPort(). > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section xlGetChannelMask on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23. > userHandle The handle is created by the application and is used for the event assignment. > streamHandle Handle to the data stream.
Return event	<code>XL_MOST_STREAM_STATE</code> (state = closed).
Return value	Returns an error code (see section Error Codes on page 423).

10.5.37 xlMostStreamStart

Syntax	<pre>XLstatus xlMostStreamStart(XLportHandle portHandle, XLaccessMask accessMask, XLuserHandle userHandle, unsigned int streamHandle, unsigned char syncChannels[MOST_ALLOC_TABLE_SIZE])</pre>
Description	Starts the transmission of data from or to the buffer. The application will be informed by an <code>XL_MOST_STREAM_BUFFER</code> event if the buffer is ready. Before starting the stream, some buffers have to be allocated by calling xlMostStreamBufferAllocate() .
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by xIOpenPort(). > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section xlGetChannelMask on page 36). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 23.

- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **streamHandle**
Handle to the data stream.

Return event XL_MOST_STREAM_STATE (state = started).

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.38 xlMostStreamStop

Syntax

```
XLstatus xlMostStreamStop(
    XLportHandle portHandle,
    XLaccessMask accessMask,
    XLuserHandle userHandle,
    unsigned int streamHandle)
```

Description The data transmission to the buffer is stopped.

Input parameters

- > **portHandle**
The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **streamHandle**
Handle to the data stream.

Return event XL_MOST_STREAM_STATE (state = stopped).

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.39 xlMostStreamBufferAllocate

Syntax

```
XLstatus xlMostStreamBufferAllocate(
    XLportHandle    portHandle,
    XLaccessMask   accessMask,
    XLuserHandle   userHandle,
    unsigned int   streamHandle,
    unsigned char** ppBuffer,
    unsigned int*   pBufferSize)
```

Description Reserves a buffer. The application reads and writes synchronous data from or to this buffer. This command has to be called after [xlMostStreamOpen\(\)](#) and before [xlMostStreamStart\(\)](#).

Input parameters

- > **portHandle**
The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Handle to the data stream.

Output parameters> **ppBuffer**

Pointer to the reserved buffer.

> **pBufferSize**

Size of the buffer. This value depends on the parameter latency (see [xlMostStreamOpen\(\)](#)).

Return event

`XL_ERR_NO_RESOURCES`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

10.5.40 `xlMostStreamBufferDeallocateAll`

Syntax

```
XLstatus xlMostStreamBufferDeallocateAll(
    XLportHandle    portHandle,
    XLaccessMask   accessMask,
    XLuserHandle   userHandle,
    unsigned int    streamHandle,
    unsigned char*  pBuffer)
```

Description

Releases any allocated buffer. Must be called before closing the stream with [xlMostStreamClose\(\)](#).

Input parameters> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Handle to the data stream.

> **pBuffer**

Pointer to the reserved buffer.

Return event

None.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

10.5.41 xlMostStreamBufferSetNext

Syntax

```
XLstatus xlMostStreamBufferSetNext(
    XLportHandle portHandle,
    XLaccessMask accessMask,
    XLuserHandle userHandle,
    unsigned int streamHandle,
    unsigned char* pBuffer,
    unsigned int filledBytes)
```

Description

This command informs the driver which buffer has to be handled next. The application may not access the buffer as long as the driver has not released it with the event `XL_MOST_STREAM_BUFFER` or if the command `xlMostStreamBufferAllocate()` fails.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **streamHandle**
Handle to the data stream.
- > **pBuffer**
Pointer to the reserved buffer.
- > **filledBytes**
Count of valid bytes in `pBuffer`.

Return event

None.

Return value

Returns an error code (see section `Error Codes` on page 423).

10.5.42 xlMostStreamClearBuffers

Syntax

```
XLstatus xlMostStreamClearBuffers (
    XLportHandle portHandle,
    XLaccessMask accessMask,
    XLuserHandle userHandle,
    unsigned int streamHandle)
```

Description

This command is available for Tx streaming only. The sizes of the buffers in the queue are set to 0 bytes. This may be used for "muting" (sending "0" on the synchronous channels).

Input parameters

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

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Handle to the data stream.

Return event None.

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.5.43 xlMostStreamGetInfo

Syntax

```
XLstatus xlMostStreamGetInfo(
    XLportHandle portHandle,
    XLaccessMask accessMask,
    XLuserHandle userHandle,
    unsigned int streamHandle,
    unsigned int* pNumSyncChannels,
    unsigned int* pDirection,
    unsigned int* pOptions,
    unsigned int* pLatency,
    unsigned int* pStreamState,
    unsigned char syncChannels[MOST_ALLOC_TABLE_SIZE])
```

Description This command gets information about a stream handle (synchronous access).

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Handle to the data stream.

> **pNumSyncChannels**

Destination buffer for width of stream.

> **pDirection**

Destination buffer for direction of stream.

> **pOptions**

Destination buffer for stream options.

> **pLatency**

Destination buffer for latency settings.

> **pStreamState**

Destination buffer for the state of the stream.

> **synChannels**

Destination buffer for channel information. Valid after `xIMostStreamStart()`.

Return event None.

Return value Returns an error code (see section [Error Codes](#) on page 423).

10.6 Structs

10.6.1 s_xl_most_async_busload_configuration

Syntax

```
typedef struct s_xl_most_async_busload_configuration {
    unsigned int      transmissionRate;
    unsigned int      counterType;
    unsigned int      counterPosition;
    XL_MOST_ASYNC_TX_EV busloadAsyncMsg;
}
```

Parameters

> **transmissionRate**

The transmission rate for stressing in frames/sec.

> **counterType**

Specifies a counter within the asynchronous frame:

XL_MOST_BUSLOAD_COUNTER_TYPE_NONE
 XL_MOST_BUSLOAD_COUNTER_TYPE_1_BYTE
 XL_MOST_BUSLOAD_COUNTER_TYPE_2_BYTE
 XL_MOST_BUSLOAD_COUNTER_TYPE_3_BYTE
 XL_MOST_BUSLOAD_COUNTER_TYPE_4_BYTE

> **counterPosition**

Describes the position of the counter within the asynchronous frame (Byte 0...1013).

Note: The counter position depends on the counter type:

- In case of a one byte counter, the position can be in the range 0..1013
- In case of a two byte counter, the position can only be in the range 1..1013
- In case of a three byte counter, the position can only be in the range 2..1013
- In case of a four byte counter, the position can only be in the range 3..1013

> **busloadAsyncMsg**

See section [XL_MOST_ASYNC_TX_EV](#) on page 205

10.6.2 s_xl_most_ctrl_busload_configuration

Syntax

```
typedef struct s_xl_most_ctrl_busload_configuration {
    unsigned int      transmissionRate;
    unsigned int      counterType;
    unsigned int      counterPosition;
    XL_MOST_CTRL_MSG_EV busloadCtrlMsg;
}
```

Parameters

> **transmissionRate**

The transmission rate for stressing in frames/sec.

> **counterType**

XL_MOST_BUSLOAD_COUNTER_TYPE_NONE
 XL_MOST_BUSLOAD_COUNTER_TYPE_1_BYTE
 XL_MOST_BUSLOAD_COUNTER_TYPE_2_BYTE
 XL_MOST_BUSLOAD_COUNTER_TYPE_3_BYTE
 XL_MOST_BUSLOAD_COUNTER_TYPE_4_BYTE

> **counterPosition**

Describes the position within the control frame (byte 0...16).

Note: The counter position depends on the counter type:

- In case of a one byte counter, the position can be in the range 0..16
- In case of a two byte counter, the position can only be in the range 1..16
- In case of a three byte counter, the position can only be in the range 2..16
- In case of a four byte counter, the position can only be in the range 3..16

> **busloadCtrlMsg**

Only the following parameters have to be set:

ctrlPrio

Transmission priority.

Can be 0x0 (for lowest priority) to 0xF (for highest priority).

ctrlType

```
XL_MOST_CTRL_TYPE_NORMAL
XL_MOST_CTRL_TYPE_REMOTE_READ
XL_MOST_CTRL_TYPE_REMOTE_WRITE
XL_MOST_CTRL_TYPE_RESOURCE_ALLOCATE
XL_MOST_CTRL_TYPE_RESOURCE_DEALLOCATE
XL_MOST_CTRL_TYPE_GET_SOURCE
```

targetAddress

Destination address.

ctrlData

Control data.

10.6.3 XL_MOST_STREAM_OPEN

Syntax

```
typedef struct s_xl_most_stream_open {
    unsigned int* pStreamHandle,
    unsigned int numSyncChannels,
    unsigned int direction,
    unsigned int options,
    unsigned int latency
} XL_MOST_STREAM_OPEN
```

Parameters

> **pStreamHandle**

Returns the handle for further operations on data stream.

> **numSyncChannels**

Count of synchronous channels (1...60).

> **direction**

XL_MOST_STREAM_RX_DATA RX streaming, MOST → PC
 XL_MOST_STREAM_TX_DATA TX streaming, PC → MOST

> **options**

With this parameter, further options can be set:

Adds time stamp and status information to the recorded data (only in Rx direction).

XL_MOST_STREAM_ADD_FRAME_HEADER

> **latency**

This parameter influences the buffer size for the streaming data (see `xlMostStreamBufferAllocate()`) and accordingly the notification of the application and CPU load respectively. There are five latency levels defined:

`XL_MOST_STREAM_LATENCY_VERY_LOW`

Very low notification cycles, very high CPU load.

`XL_MOST_STREAM_LATENCY_LOW`

`XL_MOST_STREAM_LATENCY_MEDIUM`

`XL_MOST_STREAM_LATENCY_HIGH`

`XL_MOST_STREAM_LATENCY_VERY_HIGH`

Very high notification cycles, very low CPU load.

10.7 Events

10.7.1 s_xl_event_most

Syntax

```
struct s_xl_event_most {
    unsigned int          size;
    XLeventTagMost        tag;
    unsigned short        channelIndex;
    unsigned int          userHandle;
    unsigned short        flagsChip;
    unsigned short        reserved;
    XLuint64              timeStamp;
    XLuint64              timeStamp_sync;
    union s_xl_tag_data tagData;
}
```

Parameters

- > **size**
Overall size of the event (in bytes).
The maximum size is defined in `XL_MOST_EVENT_MAX_SIZE`.
- > **tag**
Specifies the event
- > **channelIndex**
Channel of the received event.
- > **userHandle**
Enables the assignment of requests and results, e. g. while sending messages or read/write of registers.
- > **flagsChip**
The lower 8 bits specify the event source:
`XL_MOST_VN2600`
`XL_MOST_OS8104A`
`XL_MOST_OS8104B`
`XL_MOST_SPY`

The upper 8 bits specifies the flags:
`XL_MOST_QUEUE_OVERFLOW`
`XL_COMMAND_FAILED`
`XL_MOST_INTERNAL_OVERFLOW`
`XL_MOST_MEASUREMENT_NOT_ACTIVE`
`XL_MOST_QUEUE_OVERFLOW_ASYNC`
`XL_MOST_QUEUE_OVERFLOW_CTRL`
`XL_MOST_QUEUE_OVERFLOW_DRV`
- > **reserved**
For future use.
- > **timeStamp**
64 bit hardware time stamp with 1 ns resolution and 8 µs granularity.
- > **timestamp_sync**
64 bit driver synchronized time stamp with 1 ns resolution and 8 µs granularity.
- > **tagData**
Event data, depending on the size.

10.7.2 s_xl_most_tag_data

Syntax

```
union s_xl_most_tag_data {
    XL_MOST_CTRL_SPY_EV           mostCtrlSpy;
    XL_MOST_CTRL_MSG_EV          mostCtrlMsg;
    XL_MOST_ASYNC_MSG_EV         mostAsyncMsg;
    XL_MOST_ASYNC_TX_EV          mostAsyncTx;
    XL_MOST_SPECIAL_REGISTER_EV   mostSpecialRegister;
    XL_MOST_EVENT_SOURCE_EV      mostEventSource;
    XL_MOST_ALL_BYPASS_EV        mostAllBypass;
    XL_MOST_TIMING_MODE_EV       mostTimingMode;
    XL_MOST_TIMING_MODE_SPDIF_EV mostTimingModeSpdif;
    XL_MOST_FREQUENCY_EV         mostFrequency;
    XL_MOST_REGISTER_BYTES_EV    mostRegisterBytes;
    XL_MOST_REGISTER_BITS_EV     mostRegisterBits;
    XL_MOST_SYNC_ALLOC_EV        mostSyncAlloc;
    XL_MOST_CTRL_SYNC_AUDIO_EV   mostCtrlSyncAudio;
    XL_MOST_CTRL_SYNC_AUDIO_EX_EV mostCtrlSyncAudioEx;
    XL_MOST_SYNC_VOLUME_STATUS_EV mostSyncVolumeStatus;
    XL_MOST_SYNC_MUTES_STATUS_EV mostSyncMutesStatus;
    XL_MOST_RX_LIGHT_EV          mostRxLight;
    XL_MOST_TX_LIGHT_EV          mostTxLight;
    XL_MOST_LIGHT_POWER_EV       mostLightPower;
    XL_MOST_LOCK_STATUS_EV       mostLockStatus;
    XL_MOST_GEN_LIGHT_ERROR_EV   mostGenLightError;
    XL_MOST_GEN_LOCK_ERROR_EV   mostGenLockError;
    XL_MOST_RX_BUFFER_EV        mostRxBuffer;
    XL_MOST_ERROR_EV             mostError;
    XL_MOST_SYNC_PULSE_EV        mostSyncPulse;
    XL_MOST_CTRL_BUSLOAD_EV     mostCtrlBusload;
    XL_MOST_ASYNC_BUSLOAD_EV    mostAsyncBusload;
}
```

Parameters

- > **mostCtrlSpy**
See section [XL_MOST_CTRL_SPY_EV](#) on page 202.
- > **mostCtrlMsg**
See section [XL_MOST_CTRL_MSG_EV](#) on page 203.
- > **mostAsyncMsg**
See section [XL_MOST_ASYNC_MSG_EV](#) on page 205.
- > **mostAsyncTx**
See section [XL_MOST_ASYNC_TX_EV](#) on page 205.
- > **mostSpecialRegister**
See section [XL_MOST_SPECIAL_REGISTER_EV](#) on page 200.
- > **mostEventSource**
See section [XL_MOST_EVENT_SOURCE_EV](#) on page 198.
- > **mostAllBypass**
See section [XL_MOST_ALLBYPASS_EV](#) on page 198.
- > **mostTimingMode**
See section [XL_MOST_TIMING_MODE_EV](#) on page 198.
- > **mostTimingModeSpdif**
See section [XL_MOST_TIMING_MODE_SPDIF_EV](#) on page 199.
- > **mostFrequency**
See section [XL_MOST_FREQUENCY_EV](#) on page 199.
- > **mostRegisterBytes**
See section [XL_MOST_REGISTER_BYTES](#) on page 200.

- > **mostRegisterBits**
See section [XL_MOST_REGISTER_BITS_EV](#) on page 200.
- > **mostSyncAlloc**
See (see section [XL_MOST_SYNC_ALLOC_EV](#) on page 206).
- > **mostCtrlSyncAudio**
See section [XL_MOST_CTRL_SYNC_AUDIO_EV](#) on page 209.
- > **mostCtrlSyncAudioEx**
See section [XL_MOST_CTRL_SYNC_AUDIO_EX](#) on page 209.
- > **mostSyncVolumeStatus**
See section [XL_MOST_SYNC_VOLUME_STATUS_EV](#) on page 206.
- > **mostSyncMutesStatus**
See section [XL_MOST_SYNC_MUTES_STATUS_EV](#) on page 210.
- > **mostRxLight**
See section [XL_MOST_RX_LIGHT_EV](#) on page 207.
- > **mostTxLight**
See section [XL_MOST_TX_LIGHT_EV](#) on page 207.
- > **mostLightPower**
See section [XL_MOST_LIGHT_POWER_EV](#) on page 210.
- > **mostLockStatus**
See section [XL_MOST_LOCK_STATUS_EV](#) on page 207.
- > **mostGenLightError**
See section [XL_MOST_GEN_LIGHT_ERROR_EV](#) on page 210.
- > **mostGenLockError**
See section [XL_MOST_GEN_LOCK_ERROR_EV](#) on page 211.
- > **mostRxBuffer**
See section [XL_MOST_RX_BUFFER_EV](#) on page 208.
- > **mostError**
See section [XL_MOST_ERROR_EV](#) on page 208.
- > **mostSyncPulse**
See section [XL Sync Pulse](#) on page 60.
- > **mostCtrlBusload**
See section [XL_MOST_CTRL_BUSLOAD_EV](#) on page 212.
- > **mostAsyncBusload**
See section [XL_MOST_ASYNC_BUSLOAD_EV](#) on page 212.

10.7.3 XL_MOST_START

Description This event is returned after `xlActivateChannel()` call and contains the time stamp counter at measuring start without event data.

Tag `XL_MOST_START`

10.7.4 XL_MOST_STOP

Description This event is returned after `xlDeactivateChannel()` call without event data.

Tag XL_MOST_STOP

10.7.5 XL_MOST_EVENT_SOURCE_EV



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_event_source {
    unsigned int mask;
    unsigned int state;
} XL_MOST_EVENT_SOURCE_EV;
```

Description This event is returned after `xlMostSwitchEventSources()`.

Parameters

> **mask**

See `xlMostSwitchEventSources()`.

> **State**

See `xlMostSwitchEventSources()`.

Tag

XL_MOST_EVENT_SOURCE

10.7.6 XL_MOST_ALLBYPASS_EV

Syntax

```
typedef struct s_xl_most_all_bypass {
    unsigned int bypassState;
}
```

Description

Reports state of the **AllBypass** bits (see `xlMostSetAllBypass()`, `xlMostGetAllBypass()`).

Parameters

> **bypassState**

Shows the bypass state:

XL_MOST_MODE_DEACTIVATE
Bypass open.

XL_MOST_MODE_ACTIVATE
Bypass.

Tag

XL_MOST_ALLBYPASS

10.7.7 XL_MOST_TIMING_MODE_EV

Syntax

```
typedef struct s_xl_most_timing_mode {
    unsigned int timingmode;
} XL_MOST_TIMING_MODE_EV;
```

Description	Reports state of master/slave bits (see <code>xIMostSetTimingMode()</code> , <code>xIMostGetTimingMode()</code>).
Parameters	<ul style="list-style-type: none"> > timingmode <ul style="list-style-type: none"> <code>XL_MOST_TIMING_SLAVE</code> <code>XL_MOST_TIMING_MASTER</code>
Tag	<code>XL_MOST_TIMINGMODE</code>

10.7.8 XL_MOST_TIMING_MODE_SPDIF_EV

Syntax

```
typedef struct s_xl_most_timing_mode_spdif {
    unsigned int timingmode;
} XL_MOST_TIMING_MODE_SPDIF_EV;
```

Description

Reports state of master/slave SPDIF bits (see `xIMostSetTimingMode()`, `xIMostGetTimingMode()`).

Parameters

- > **timingmode**

- `XL_MOST_TIMING_SLAVE`
- `XL_MOST_TIMING_MASTER`
- `XL_MOST_TIMING_SLAVE_SPDIF_MASTER`
- `XL_MOST_TIMING_SLAVE_SPDIF_SLAVE`
- `XL_MOST_TIMING_MASTER_SPDIF_MASTER`
- `XL_MOST_TIMING_MASTER_SPDIF_SLAVE`
- `XL_MOST_TIMING_MASTER_FROM_SPDIF_SLAVE`

Tag

`XL_MOST_TIMINGMODE_SPDIF`

10.7.9 XL_MOST_FREQUENCY_EV

Syntax

```
typedef struct s_xl_most_frequency {
    unsigned int frequency;
} XL_MOST_FREQUENCY_EV;
```

Description

Reports frame rate of the MOST network.

Parameters

- > **frequency**

`XL_MOST_FREQUENCY_44100`
Bus frequency is 44.1 kHz.

`XL_MOST_FREQUENCY_48000`
Bus frequency is 48 kHz.

`XL_MOST_FREQUENCY_ERROR`
Error while getting the frequency.

Tag

`XL_MOST_FREQUENCY`

10.7.10 XL_MOST_REGISTER_BYTES

Syntax

```
typedef struct s_xl_most_register_bytes {
    unsigned int number;
    unsigned int address;
    unsigned char value[16];
} XL_MOST_REGISTER_BYTES_EV;
```

Description

This event is returned after a read or write request (see [xIMostReadRegister\(\)](#) and [xIMostWriteRegister\(\)](#)).

Parameters

- > **number**
Number of bytes (max 16).
- > **address**
Start address of the data.
- > **value**
Requested data.

Tag

XL_MOST_REGISTER_BYTES

10.7.11 XL_MOST_REGISTER_BITS_EV

Syntax

```
typedef struct s_xl_most_register_bits {
    unsigned int address;
    unsigned int value;
    unsigned int mask;
} XL_MOST_REGISTER_BITS_EV;
```

Description

This event is returned after a write request (see section [xIMostWriteRegisterBit](#) on page 166).

Parameters

- > **address**
Address for the requested register.
- > **value**
Values for the with mask specified bits.
- > **mask**
Mask for the identified values.

Tag

XL_MOST_REGISTER_BITS

10.7.12 XL_MOST_SPECIAL_REGISTER_EV

Syntax

```
struct s_xl_most_special_register{
    unsigned int changeMask;
    unsigned int lockStatus;
    unsigned char register_bNAH;
    unsigned char register_bNAL;
    unsigned char register_bGA;
    unsigned char register_bAPAH;
    unsigned char register_bAPAL;
    unsigned char register_bNPR;
    unsigned char register_bMPR;
    unsigned char register_bNDR;
    unsigned char register_bMDR;
```

```

    unsigned char register_bSBC;
    unsigned char register_bXTIM;
    unsigned char register_bXRTY;
} XL_MOST_SPECIAL_REGISTER_EV;

```

Description

This event reports spontaneously changes of specific register values. This event should also occur when the registers are overwritten by `xlMostWriteRegister()` or `xlMostWriteRegisterBit()`.

Parameters**> changeMask**

Mask for the register changes.

- `XL_MOST_NA_CHANGED`
- `XL_MOST_GA_CHANGED`
- `XL_MOST_APACHE_CHANGED`
- `XL_MOST_NPR_CHANGED`
- `XL_MOST_MPR_CHANGED`
- `XL_MOST_NDR_CHANGED`
- `XL_MOST_MDR_CHANGED`
- `XL_MOST_SBC_CHANGED`
- `XL_MOST_XTIM_CHANGED`
- `XL_MOST_XRTY_CHANGED`

> lockStatus

- `XL_MOST_UNLOCK`
- `XL_MOST_LOCK`

> register_bNAH

Node address high byte (see section [Specific OS8104 Registers](#) on page 159).

> register_bNAL

Node address low byte (see section [Specific OS8104 Registers](#) on page 159).

> register_bGA

Group address (see section [Specific OS8104 Registers](#) on page 159).

> register_bAPAH

Alternate packet address high byte (see section [Specific OS8104 Registers](#) on page 159).

> register_bAPAL

Alternate packet address low byte (see section [Specific OS8104 Registers](#) on page 159).

> register_bNPR

Node position register (see section [Specific OS8104 Registers](#) on page 159).

Maximum position register (see section [Specific OS8104 Registers](#) on page 159).

Node delay register (see section [Specific OS8104 Registers](#) on page 159).

> register_bMDR

Maximum delay register (see section [Specific OS8104 Registers](#) on page 159).

> register_bSBC

Synchronous bandwidth control (see section [Specific OS8104 Registers](#) on page 159).

> register_bXTIM

Transmit retry time register (see section [Specific OS8104 Registers](#) on page 159).

> register_bXRTY

Transmit retry register (see section [Specific OS8104 Registers](#) on page 159).

Tag

XL_MOST_SPECIAL_REGISTER

10.7.13 XL_MOST_CTRL_SPY_EV

**Note**

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_ctrl_spy {  
    unsigned int arbitration;  
    unsigned short targetAddress;  
    unsigned short sourceAddress;  
    unsigned char ctrlType;  
    unsigned char ctrlData[17];  
    unsigned short crc;  
    unsigned short txStatus;  
    unsigned short ctrlRes;  
    unsigned int spyRxStatus;  
} XL_MOST_CTRL_SPY_EV;
```

Description

This event shows a received control message from the spy (userHandle=0).

Parameters

- > **arbitration**
NULL.
- > **targetAddress**
Received target address.
- > **sourceAddress**
Received source address.
- > **ctrlType**
XL_MOST_CTRL_TYPE_NORMAL
XL_MOST_CTRL_TYPE_REMOTE_READ
XL_MOST_CTRL_TYPE_REMOTE_WRITE
XL_MOST_CTRL_TYPE_RESOURCE_ALLOCATE
XL_MOST_CTRL_TYPE_RESOURCE_DEALLOCATE
XL_MOST_CTRL_TYPE_GET_SOURCE
- > **ctrlData**
Data of the control frame.
- > **crc**
CRC of the control frame.
- > **txStatus**
Tx status of the received control frame.
- > **ctrlRes**
For future use.

> **spyRxStatus**

XL_MOST_SPY_RX_STATUS_NO_LIGHT

After the first preamble, the light disappeared; At least once, maybe more times.
An undefined part of the message is invalid.

XL_MOST_SPY_RX_STATUS_NO_LOCK

After the first preamble, a loss of lock has been detected; At least once, maybe more times. An undefined part of the message can be invalid

XL_MOST_SPY_RX_STATUS_BIPHASE_ERROR

After the first preamble, a biphasic coding error has been detected; At least once, maybe more times. An undefined part of the message can be invalid.

XL_MOST_SPY_RX_STATUS_MESSAGE_LENGTH_ERROR

This message consisted of more or less preambles than allowed (MOST specification). The stored message was cut or filled with undefined data.

XL_MOST_SPY_RX_STATUS_PARITY_ERROR

In one or more of all 16 frames a parity error has been detected.

This could have caused a wrong control message but needs not to.

XL_MOST_SPY_RX_STATUS_FRAME_LENGTH_ERROR

After the first preamble, a frame longer than allowed (MOST specification, 512 Bit) has been detected. This could result in an erroneous message.

XL_MOST_SPY_RX_STATUS_PREAMBLE_TYPE_ERROR

After the first preamble, an unknown preamble type has been detected.

This could result in an erroneous message.

XL_MOST_SPY_RX_STATUS_CRC_ERROR

The CRC check of the message detected an error.

Tag

XL_MOST_CTRL_RX_SPY

10.7.14 XL_MOST_CTRL_MSG_EV**Syntax**

```
typedef struct s_xl_most_ctrl_msg {
    unsigned char ctrlPrio;
    unsigned char ctrlType;
    unsigned short targetAddress;
    unsigned short sourceAddress;
    unsigned char ctrlData[17];
    unsigned char direction;
    unsigned int status;
} XL_MOST_CTRL_MSG_EV;
```

Description

This event reports the receiving of a control message of the node (`userHandle = 0`). Transmits a control message or is transmission confirmation.

Parameters> **ctrlPrio**

Transmission priority. Can be `0x0` (for lowest priority) to `0xF` (for highest priority).

> ctrlType

XL_MOST_CTRL_TYPE_NORMAL
XL_MOST_CTRL_TYPE_REMOTE_READ
XL_MOST_CTRL_TYPE_REMOTE_WRITE
XL_MOST_CTRL_TYPE_RESOURCE_ALLOCATE
XL_MOST_CTRL_TYPE_RESOURCE_DEALLOCATE
XL_MOST_CTRL_TYPE_GET_SOURCE

> targetAddress

Own address on receiving.

> sourceAddress

Unused for transmitting.

> ctrlData

Control data.

> direction

XL_MOST_DIRECTION_RX
XL_MOST_DIRECTION_TX (also on Tx acknowledge)

> status

Only relevant on transmitting:

Low byte

Transmit Status Register (see OS8104 datasheet, 13.2.3 bXTS):

0x00

Transmission failed. No response from target node.

0x10

Transmission successful.

0x11

Transmission successful, message type not supported by receiving node.

0x20

Transmission failed: Bad CRC.

0x21

Transmission failed. Node's receive buffer was full.

0x30

Groupcast/broadcast transmission partly failed (one node acknowledged 0x10, other node acknowledged 0x20).

0x31

Groupcast/broadcast transmission partly failed (one node acknowledged 0x11, other node acknowledged 0x20).

Flags

XL_MOST_TX_WHILE_UNLOCKED

The slave is unlocked. The message is not send.

XL_MOST_TX_TIMEOUT

Error while transmitting to the OS8104 or switched off os8104 events (see section [xlMostSwitchEventSources](#) on page 160).

Tag XL_MOST_CTRL_RX_OS8104

10.7.15 XL_MOST_CTRL_TX

Description See section [XL_MOST_CTRL_MSG_EV](#) on page 203.

Tag XL_MOST_CTRL_TX

10.7.16 XL_MOST_ASYNC_MSG_EV

Syntax

```
typedef struct s_xl_most_async_msg {
    unsigned int    status;
    unsigned int    crc;
    unsigned char   arbitration;
    unsigned char   length;
    unsigned short  targetAddress;
    unsigned short  sourceAddress;
    unsigned char   asyncData[1018];
} XL_MOST_ASYNC_MSG_EV;
```

Description The event is fired on node Rx and spy messages.

Parameters

- > **status**
XL_MOST_ASYNC_NO_ERROR
XL_MOST_ASYNC_SBC_ERROR
XL_MOST_ASYNC_NEXT_STARTS_TO_EARLY
XL_MOST_ASYNC_TO_LONG
- > **Crc**
Not used.
- > **arbitration**
Value is calculated by the bus controller in the following way:
(node position * 2) + 1.
- > **length**
Databytes + 2 Byte in quadlets (4 Bytes).
- > **targetAddress**
Unused.
- > **sourceAddress**
Unused.
- > **asyncData**
Unused.

Tag XL_MOST_ASYNC_MSG

10.7.17 XL_MOST_ASYNC_TX_EV

Syntax

```
typedef struct s_xl_most_async_tx{
    unsigned char   arbitration;
    unsigned char   length;
    unsigned short  targetAddress;
```

```
    unsigned short sourceAddress;
    unsigned char  asyncData[1014];
} XL_MOST_ASYNC_TX_EV;
```

Description The event is fired as a transmit acknowledge (`userHandle != 0`; refer to [xIMostASyncTransmit\(\)](#)).

Parameters

- > **arbitration**
Value is calculated by the bus controller in the following way:
(node position * 2) + 1.
- > **length**
Databytes + 2 Byte in quadlets (4 Bytes).
- > **targetAddress**
Logical target address.
- > **sourceAddress**
Logical Source address.
- > **asyncData**
Data bytes (depending on length).

Tag XL_MOST_ASYNC_TX

10.7.18 XL_MOST_SYNC_ALLOC_EV

Syntax

```
typedef struct s_xl_most_sync_alloc {
    unsigned char allocTable[MOST_ALLOC_TABLE_SIZE];
} XL_MOST_SYNC_ALLOC_EV;
```

Description The event responses on changes within the allocation table for the synchronous channels. It is also the answer for [xIMostSyncGetAllocTable\(\)](#) (`userHandle != 0`).

Parameters

- > **allocTable**
Only the first 60 bytes contains the alloc table.
Byte 63 MPR .
Byte 64 MDR.

Tag XL_MOST_SYNC_ALLOCABLE

10.7.19 XL_MOST_SYNC_VOLUME_STATUS_EV

Syntax

```
typedef struct s_xl_most_sync_volume_status {
    unsigned int device;
    unsigned int volume;
} XL_MOST_SYNC_VOLUME_STATUS_EV;
```

Description Reports the volume level for the line in and line out ports.

Parameters

- > **device**
Describes the device address:
`XL_MOST_DEVICE_CASE_LINE_IN`
`XL_MOST_DEVICE_CASE_LINE_OUT`

- > **volume**
Volume level from 0...255 (0...100%).

Tag XL_MOST_SYNC_VOLUME_STATUS

10.7.20 XL_MOST_RX_LIGHT_EV

Syntax

```
typedef struct s_xl_most_rx_light {
    unsigned int light;
} XL_MOST_RX_LIGHT_EV;
```

Description This event reports changes on the FOT (`userHandle = 0`) or answers to an `xIMostGetRxLight()` request (`userHandle != 0`).

Parameters

- > **light**
`XL_MOST_LIGHT_OFF`
FOT light is off.

`XL_MOST_LIGHT_FORCE_ON`
FOT light is on.

`XL_MOST_LIGHT_MODULATED`
FOT light is modulated.

Tag XL_MOST_RX_LIGHT

10.7.21 XL_MOST_TX_LIGHT_EV

Syntax

```
typedef struct s_xl_most_tx_light {
    unsigned int light;
} XL_MOST_TX_LIGHT_EV;
```

Description The event reports changes on the FOT (`userHandle = 0`) or answers to `xIMostSetTxLight()` and `xIMostGetTxLight()` (`userHandle != 0`) requests.

Parameters

- > **light**
`XL_MOST_LIGHT_OFF`
FOT light is off.

`XL_MOST_LIGHT_FORCE_ON`
FOT light is on.

`XL_MOST_LIGHT_MODULATED`
FOT light is modulated.

Tag XL_MOST_TX_LIGHT

10.7.22 XL_MOST_LOCK_STATUS_EV

Syntax

```
typedef struct s_xl_most_lock_status {
    unsigned int lockStatus;
} XL_MOST_LOCK_STATUS_EV;
```

Description This event reports changes on the lock status of the PLL (`userHandle = 0`) or reports an answer to `xlMostGetLockStatus()` (`userHandle != 0`).

Parameters

- > **lockStatus**

`XL_MOST_UNLOCK`
Ring unlocked.

`XL_MOST_LOCK`
Ring locked.

Tag `XL_MOST_LOCKSTATUS`

10.7.23 XL_MOST_ERROR_EV

Syntax

```
typedef struct s_xl_most_error {
    unsigned int errorCode;
    unsigned int parameter[3];
} XL_MOST_ERROR_EV;
```

Description This event reports an error.

Parameters

- > **errorCode**

`XL_MOST_ERROR_UNKNOWN_COMMAND`
Unknown function call.

`XL_MOST_CTRL_TYPE_QUEUE_OVERFLOW`
Overflow of the internal Tx queue for control frames.

`XL_MOST_ASYNC_TYPE_QUEUE_OVERFLOW`
Overflow of the internal Tx queue for asynchronous frames.

`XL_MOST_SYNC_PULSE_ERROR`
Internal sync pulse error.

`XL_MOST_FPGA_TS_FIFO_OVERFLOW`
Internal overflow.

`XL_MOST_ASYNC_RX_OVERFLOW_ERROR`
Lost received asynchronous frames.

`XL_MOST_SPY_OVERFLOW_ERROR`
Lost received ctrl frames (from spy).

- > **parameter**

Reserved for future use.

Tag `XL_MOST_ERROR`

10.7.24 XL_MOST_RX_BUFFER_EV

Syntax

```
typedef struct s_xl_most_rx_buffer {
    unsigned int mode;
} XL_MOST_RX_BUFFER_EV;
```

Description This event confirms the `xIMostCtrlRxBuffer()` call.

Parameters

- > **mode**

0

Off.

1

Simulation of full Rx buffer on.

Tag

`XL_MOST_CTRL_RXBUFFER`

10.7.25 XL_MOST_CTRL_SYNC_AUDIO_EV

Syntax

```
typedef struct s_xl_most_ctrl_sync_audio {
    unsigned int channelMask[4];
    unsigned int device;
    unsigned int mode;
} XL_MOST_CTRL_SYNC_AUDIO_EV;
```

Description

The event is the response on an `xIMostCtrlSyncAudio()` function call.

Parameters

- > **channelMask**

Contains the channel numbers for the synchronous data.

- > **device**

Describes the device address:

`XL_MOST_DEVICE_CASE_LINE_IN`

`XL_MOST_DEVICE_CASE_LINE_OUT`

- > **mode**

section `xIMostCtrlSyncAudio` on page 169

Tag

`XL_MOST_CTRL_SYNC_AUDIO`

10.7.26 XL_MOST_CTRL_SYNC_AUDIO_EX

Syntax

```
typedef struct s_xl_most_ctrl_sync_audio_ex {
    unsigned int channelMask[16];
    unsigned int device;
    unsigned int mode;
} XL_MOST_CTRL_SYNC_AUDIO_EX_EV;
```

Description

Response on an `xIMostCtrlSyncAudioEx()` function call.

Parameters

- > **channelMask**

Contains the channel numbers for the synchronous data.

- > **device**

Describes the device address:

`XL_MOST_DEVICE_CASE_LINE_IN`

`XL_MOST_DEVICE_CASE_LINE_OUT`

`XL_MOST_DEVICE_SPDIF_IN`

`XL_MOST_DEVICE_SPDIF_OUT`

`XL_MOST_DEVICE_SPDIF_IN_OUT_SYNC`

- > **mode**
section `xIMostCtrlSyncAudioEx` on page 170

Tag `XL_MOST_CTRL_SYNC_AUDIO_EX`

10.7.27 XL_MOST_SYNC_MUTES_STATUS_EV

Syntax

```
typedef struct s_xl_most_sync_mutes_status {
    unsigned int device;
    unsigned int mute;
} XL_MOST_SYNC_MUTES_STATUS_EV;
```

Description Reports the mute status for the line in and the line out ports.

Parameters

- > **device**

Describes the device address:

`XL_MOST_DEVICE_CASE_LINE_IN`
`XL_MOST_DEVICE_CASE_LINE_OUT`

- > **mute**

Mute status for the addressed device:

`XL_MOST_NO_MUTE`
 Audio device is not muted.

`XL_MOST_MUTE`

Audio device is muted.

Tag `XL_MOST_SYNC_MUTES_STATUS`

10.7.28 XL_MOST_LIGHT_POWER_EV

Syntax

```
typedef struct s_xl_most_light_power {
    unsigned int lightPower;
} XL_MOST_LIGHT_POWER_EV;
```

Description Reports the light power on the FOT.

Parameters

- > **lightPower**

Power status of the FOT:

`XL_MOST_LIGHT_FULL`
 Normal light power.

`XL_MOST_LIGHT_3DB`

Reduced light power.

Tag `XL_MOST_TXLIGHT_POWER`

10.7.29 XL_MOST_GEN_LIGHT_ERROR_EV



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_gen_light_error {
    unsigned int lightOnTime;
    unsigned int lightOffTime;
    unsigned int repeat;
} XL_MOST_GEN_LIGHT_ERROR_EV;
```

Description

This event signals start and stop of the light-on/light-off stress mode (see section [xIMostGenerateLightError](#) on page 176).

Parameters

- > **lockOnTime**
Time of modulated light emission.
- > **lockOffTime**
Time of unmodulated light emission.
- > **repeat**
0
Light (ON/OFF) changes.

>0
Count of the ON/OFF changes.

Tag

XL_MOST_GENLIGHTERROR

10.7.30 XL_MOST_GEN_LOCK_ERROR_EV

**Note**

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_gen_lock_error {
    unsigned int lockOnTime;
    unsigned int lockOffTime;
    unsigned int repeat;
} XL_MOST_GEN_LOCK_ERROR_EV;
```

Description

This event signals start and stop of the lock-unlock stress mode (see section [xIMostGenerateLockError](#) on page 177).

Parameters

- > **lockOnTime**
The on time in ms.
- > **lockOffTime**
The off time in ms.
- > **repeat**
0
After the test has expired.

! 0
At the beginning (value is the same like in the command [xIMostGenerateLockError](#)()).

Tag

XL_MOST_GENLOCKERROR

10.7.31 XL_MOST_CTRL_BUSLOAD_EV


Note

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_ctrl_busload {
    unsigned int busloadCtrlStarted;
} XL_MOST_CTRL_BUSLOAD_EV;
```

Description

This is the response event for the `xIMostCtrlGenerateBusload()` and shows the start/stop of the bus load generation. The `xIMostCtrlConfigureBusload()` must be called first.

Parameters
> busloadCtrlStarted

`XL_MODE_ACTIVATE`
Busload test started.

`XL_MODE_DEACTIVATE`
Busload test stopped.

Tag

`XL_MOST_CTRL_BUSLOAD`

10.7.32 XL_MOST_ASYNC_BUSLOAD_EV


Note

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_async_busload {
    unsigned int busloadAsyncStarted;
} XL_MOST_ASYNC_BUSLOAD_EV;
```

Description

This is the response event on an `xIMostAsyncGenerateBusload()` function call and shows the start/stop of the busload generation. The `xIMostAsyncConfigureBusload()` must be called first.

Parameters
> busloadAsyncStarted

`XL_MODE_ACTIVATE`
Busload test started.

`XL_MODE_DEACTIVATE`
Busload test stopped.

Tag

`XL_MOST_ASYNC_BUSLOAD`

10.7.33 XL_MOST_STREAM_BUFFER

Syntax

```
typedef struct s_xl_most_stream_buffer {
    unsigned int streamHandle;
    unsigned char *POINTER_32 pBuffer;
    unsigned int validBytes;
    unsigned int status;
    unsigned int pBuffer_highpart;
}
```

```
    } XL_MOST_STREAM_BUFFER_EV;
```

Description This event reports the availability of a buffer for read and write operations to the application.

Parameters

- > **streamHandle**
Handle to the stream.
- > **pBuffer**
Pointer to the buffer.
- > **validBytes**
Count of valid bytes in the buffer (Rx) or count of sent bytes from the buffer (Tx).
- > **status**
 - XL_SUCCESS
 - OK
 - XL_BUFFER_ERROR
Data is lost
- > **pBuffer_highpart**
The upper DWORD of the data pointer on 64 bit systems.

10.7.34 XL_MOST_STREAM_STATE_EV

Syntax

```
typedef struct s_xl_most_stream_state {
    unsigned int streamHandle;
    unsigned int streamState;
    unsigned int streamError;
    unsigned int reserved;
} XL_MOST_STREAM_STATE_EV;
```

Description This event is received by all applications to inform about the availability of the resource „streaming“.

Parameters

- > **streamHandle**
Handle to the stream.
- > **streamState**
State of the stream.
 - XL_MOST_STREAM_STATE_CLOSED
 - XL_MOST_STREAM_STATE_OPENED
 - XL_MOST_STREAM_STATE_STARTED
 - XL_MOST_STREAM_STATE_STOPPED
 - XL_MOST_STREAM_STATE_START_PENDING
Still processing start command.
 - XL_MOST_STREAM_STATE_STOP_PENDING
Still processing stop command.
 - XL_MOST_STREAM_STATE_UNKNOWN

- > **streamError**
 - XL_MOST_STREAM_ERR_NO_ERROR
 - XL_MOST_STREAM_ERR_INVALID_HANDLE
 - XL_MOST_STREAM_ERR_NO_MORE_BUFFERS_AVAILABLE
 - XL_MOST_STREAM_ERR_ANY_BUFFER_LOCKED
 - XL_MOST_STREAM_ERR_WRITE_RE_FAILED
 - XL_MOST_STREAM_ERR_STREAM_ALREADY_STARTED
 - XL_MOST_STREAM_ERR_TX_BUFFER_UNDERRUN
 - XL_MOST_STREAM_ERR_RX_BUFFER_OVERFLOW
 - XL_MOST_STREAM_ERR_INSUFFICIENT_RESOURCES

10.7.35 XL_MOST_SYNC_TX_UNDERFLOW_EV



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_sync_tx_underflow {
    unsigned int streamHandle;
    unsigned int reserved;
} XL_MOST_SYNC_TX_UNDERFLOW_EV;
```

Description

This event is reported in case no data was available to send due to an empty transmit buffer.

Parameters

- > **streamHandle**
Stream handle (returned by `xIMostStreamOpen()`).
- > **reserved**
For future use.

Tag

XL_MOST_SYNC_TX_UNDERFLOW

10.7.36 XL_MOST_SYNC_RX_OVERFLOW_EV



Note

This feature is available in the MOST Analysis Library only.

Syntax

```
typedef struct s_xl_most_sync_rx_overflow {
    unsigned int streamHandle;
    unsigned int reserved;
} XL_MOST_SYNC_RX_OVERFLOW_EV;
```

Description

This event is reported in case no data was available to send due to an empty transmit buffer.

Parameters

- > **streamHandle**
Stream handle (returned by `xIMostStreamOpen()`).
- > **reserved**
For future use.

Tag

XL_MOST_SYNC_RX_OVERFLOW

10.8 Application Examples

10.8.1 xIMOSTView

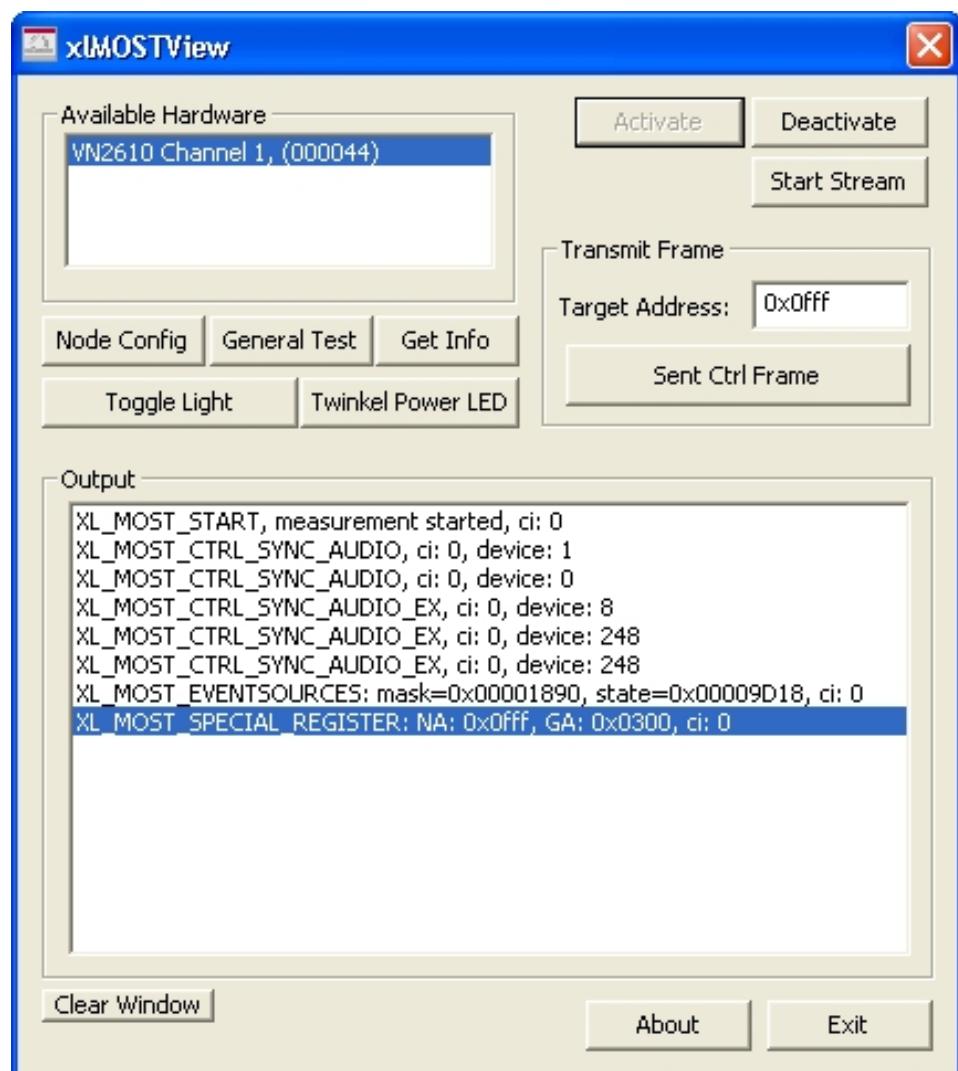
10.8.1.1 General Information

Description

This example demonstrates the basic handling of the XL MOST API. After execution, it searches for available MOST devices and assigns them automatically in the **Vector Hardware Configuration** tool. The found devices are shown in the **Available Hardware** box and are activated.

You can select and parameterize the devies with the button **[Node Config]** (or by a double click on the device). To send a control frame, you have to define the source and target address and then press the **[Send Ctrl Frame]** button. The **Output** box shows the return events of every function call or incoming messages.

The **[General Test]** and the **[Start Stream]** button are only available if the MOST Analysis Library is being used. The streaming function can be used with the CANoe StreamFromFile.cfg.



10.8.1.2 Classes

Description	The example has the following class structure:
> CGeneral	Every MOST device has a parameter class. The node group address is saved there for example.
> CNodeParam	Contains the MOST node parameter.
> CMOSTFunctions	Implementation of all library functions.
> CMOSTGeneralTest	Implementation of the General Test dialog box.
> CMOSTNodeConfig	Implementation of the Node Config dialog box.
> CMOSTParseEvent	Contains an event parser to display the received events.
> CMOSTStreaming	Includes the streaming feature.

10.8.1.3 Functions

Description	> CGeneral Contains only general functions for handling, e. g. string converting.
-------------	---

> CMOSTFunctions

Implementation for the XL MOST API handling.

MOSTInit

Initializes all connected MOST devices. For every device a thread is created. Every device gets a separate port which is activated. The first MOST interface is set up as timing master.

MOSTClose

Closes the threads and port handles.

MOSTActivate

Activates the selected MOST channel.

MOSTDeactivate

Deactivates the selected MOST channel.

MOSTCtrlTransmit

Transmits a control frame to the selected channel.

MOSTToggleLight

Toggles the FOT light from on, off to modulated and back.

MOSTSetupNode

Sets up the MOST node (node group address, bypass mode, timing mode and frequency).

MOSTGetInfo

Requests the information of a MOST channel (like timing mode, bypass mode...).

MOSTTwinklePowerLED

Twinkles the power LEDs.

MOSTGenerateLightError

Generates light errors depending on the counter.

MOSTGenerateLockError

Generates lock errors depending on the counter.

> CMOSTGeneralTest

Handles the dialog box General Test.

> CMOSTNodeConfig

Handles the dialog box Node Config.

> CMOSTStreaming

MOSTStreamInit

Opens the stream, allocates the streaming buffers and starts the MOST streaming. All streaming data will be stored within the `most.bin` logfile

MOSTStreamClose

Closes the stream and frees up the allocated memory.

MOSTStreamParse

Parses the streaming events. Handles the buffer events and stores the data into the logfile. Initiates the corresponding functions to handle the MOST state events.

11 MOST 150 Commands

In this chapter you find the following information:

11.1 Introduction	220
11.2 Flowchart	221
11.3 MOST150 Analysis Library and Node Functions	223
11.4 Functions	226
11.5 Structs	265
11.6 Events	272
11.7 Application Examples	306

11.1 Introduction

Description

The **XL Driver Library** enables the development of MOST applications for supported Vector devices (see section System Requirements on page 28). A MOST application always requires **init access**(see section `xlOpenPort` on page 37)multiple MOST applications cannot use a common physical MOST channel at the same time.

Depending on the channel property **init access** (see page 25), the application's main features are as follows:

With init access

- > channel parameters can be changed/configured
- > MOST frames can be transmitted on the channel
- > MOST frames 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.

Generally, the Vector MOST150 interface can be parametrized without activating the channel. However, it is recommended to activate the channel before, otherwise the responding events are not recognized. To address the event to the corresponding function call, a user handle within the event is available. If the `userHandle` is non zero the event is a response to a function call, otherwise it is a message or state change event. The `userHandle` can be set up on function call and returns on the responding event.

Reset of VN2600 interface family

When the VN2610/VN2640 interface is plugged in, the following default values for a MOST150 node are set:

frequency	44.1 kHz
Node address	0xFFFF
Group address	0x300
MAC address	0xFFFFFFFFFFFF

11.2 Flowchart

Calling sequence

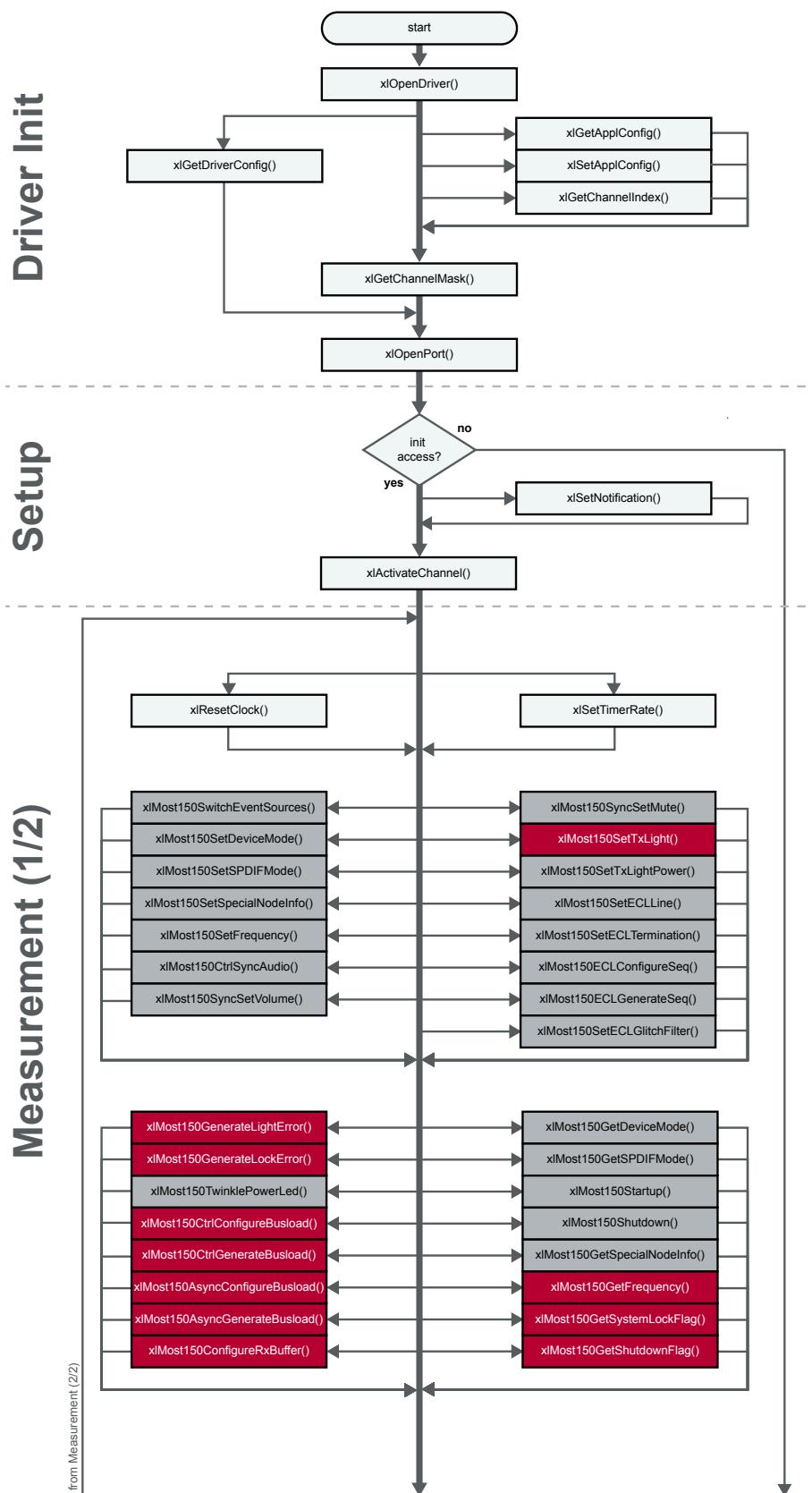


Figure 19: Function calls for MOST150 applications (1/2)

Calling sequence

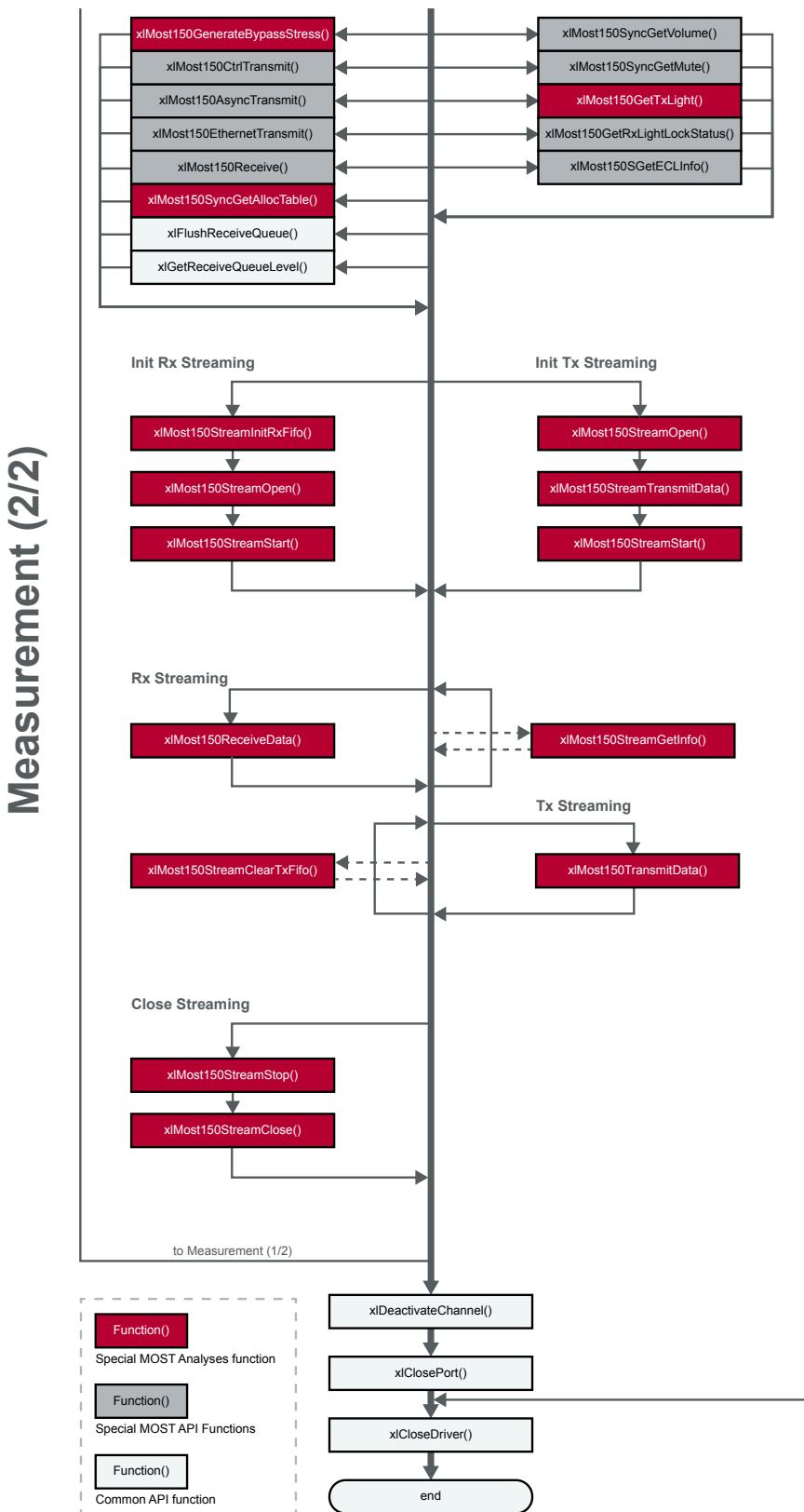


Figure 20: Function calls for MOST150 applications (2/2)

11.3 MOST150 Analysis Library and Node Functions

	MOST150 Command	Node Function	MOST150 Analysis Lib
Administration/ configuration	xIMost150SwitchEventSources	Limited*	X
Node Configuration	MOST150 Command	Node Function	MOST150 Analysis Lib
	xIMost150SetDeviceMode	X	X
	xIMost150GetDeviceMode	X	X
	xIMost150SetSPDIFMode	X	X
	xIMost150GetSPDIFMode	X	X
	xIMost150SetSpecialNode	X	X
	xIMost150GetSpecialNode	X	X
	xIMost150SetFrequency	X	X
	xIMost150GetFrequency	-	X
	xIMost150CtrlTransmit	X	X
	xIMost150AsyncTransmit	X	X
	xIMost150GetSystemLockFlag	-	X
	xIMost150GetShutdownFlag	-	X
	xIMost150Shutdown	X	X
	xIMost150Startup	X	X
Messages	MOST150 Command	Node Function	MOST150 Analysis Lib
	xIMost150CtrlTransmit	X	X
	xIMost150AsyncTransmit	X	X
	xIMost150EthernetTransmit	X	X
Channel allocation	MOST150 Command	Node Function	MOST150 Analysis Lib
	xIMost150SyncGetAllocTable	-	X
Synchronous channel I/O	MOST150 Command	Node Function	MOST150 Analysis Lib
	xIMost150CtrlSyncAudio	X	X
	xIMost150SyncSetVolume	X	X
	xIMost150SyncGetVolume	X	X
	xIMost150SyncSetMute	X	X
	xIMost150SyncGetMute	X	X
Optical interface	MOST150 Command	Node Function	MOST150 Analysis Lib
	xIMost150GetRxLightLockStatus	X	X
	xIMost150SetTxLight	-	X
	xIMost150GetTxLight	-	X
	xIMost150SetTxLightPower	X	X

General tester

MOST150 Command	Node Function	MOST150 Analysis Lib
xIMost150GenerateLightError	-	X
xIMost150GenerateLockError	-	X
xIMost150CtrlConfigureBusload	-	X
xIMost150CtrlGenerateBusload	-	X
xIMost150AsyncConfigureBusload	-	X
xIMost150AsyncGenerateBusload	-	X
xIMost150GenerateBypassStress	-	X
xIMost150ConfigureRxBuffer	-	X

ECL commands

MOST150 Command	Node Function	MOST150 Analysis Lib
xIMost150SetECLine	X	X
xIMost150SetECLTermination	X	X
xIMost150GetECLInfo	X	X
xIMost150ECLConfigureSeq	-	X
xIMost150ECLGenerateSeq	-	X
xIMost150SetECLGlitchFilter	-	X

Synchronous channel streaming

MOST150 Command	Node Function	MOST150 Analysis Lib
xIMost150StreamOpen	-	X
xIMost150StreamClose	-	X
xIMost150StreamStart	-	X
xIMost150StreamStop	-	X
xIMost150StreamTransmitData	-	X
xIMost150StreamInitRxFifo	-	X
xIMost150StreamReceiveData	-	X
xIMost150StreamGetInfo	-	X

General MOST commands

MOST150 Command	Node Function	MOST150 Analysis Lib
xIMost150Receive	Limited*	X
xIMost150TwinklePowerLED	X	X
xIGenerateSyncPulse	-	X

Possible Rx events
(for xIMostReceive)

MOST150 Command	Node Function	MOST150 Analysis Lib
XL_START	X	X
XL_STOP	X	X
XL_TIMER	X	X
XL_SYNC_PULSE	-	X
XL_MOST150_EVENT_SOURCE	Limited*	X
XL_MOST150_DEVICE_MODE	X	X
XL_MOST150_SPDIFMODE	X	X
XL_MOST150_FREQUENCY	X	X
XL_MOST150_SPECIAL_NODE_INFO	X	X
XL_MOST150_CTRL_SPY	-	X
XL_MOST150_CTRL_RX	X	X

Possible Rx events
(for xlMostReceive)

MOST150 Command	Node Function	MOST150 Analysis Lib
XL_MOST150_CTRL_TX_ACK	X	X
XL_MOST150_ASYNC_SPY	-	X
XL_MOST150_ASYNC_RX	X	X
XL_MOST150_ASYNC_TX_ACK	X	X
XL_MOST150_SYNC_ALLOC_INFO	-	X
XL_MOST150_TX_LIGHT	X	X
XL_MOST150_RXLIGHT_LOCKSTATUS	X	X
XL_MOST150_ERROR	X	X
XL_MOST150_CTRL_SYNC_AUDIO	X	X
XL_MOST150_SYNC_VOLUME_STATUS	X	X
XL_MOST150_SYNC_MUTE_STATUS	X	X
XL_MOST150_LIGHT_POWER	X	X
XL_MOST150_GEN_LIGHT_ERROR	-	X
XL_MOST150_GEN_LOCK_ERROR	-	X
XL_MOST150_CONFIGURE_RX_BUFFER	-	X
XL_MOST150_CTRL_BUSLOAD	-	X
XL_MOST150_ASYNC_BUSLOAD	-	X
XL_MOST150_ETHERNET_SPY	-	X
XL_MOST150_ETHERNET_RX	X	X
XL_MOST150_ETHERNET_TX_ACK	X	X
XL_MOST150_SYSTEMLOCK_FLAG	-	X
XL_MOST150_SHUTDOWN_FLAG	-	X
XL_MOST150_NW_STARTUP	X	X
XL_MOST150_NW_SHUTDOWN	X	X
XL_MOST150_ECL_LINE_CHANGED	X	X
XL_MOST150_ECL_TERMINATION_CHANGED	X	X
XL_MOST150_ECL_SEQUENCE	-	X
XL_MOST150_ECL_GLITCH_FILTER	-	X
XL_MOST150_STREAM_STATE	-	X
XL_MOST150_STREAM_TX_BUFFER	-	X
XL_MOST150_STREAM_TX_LABEL	-	X
XL_MOST150_STREAM_TX_UNDERFLOW	-	X
XL_MOST150_STREAM_RX_BUFFER	-	X
XL_MOST150_GEN_BYPASS_STRESS	-	X

* No control spy events, no asynchronous spy events, no allocation table events, no synchronous events.

11.4 Functions

11.4.1 xlMost150SwitchEventSources

Syntax

```
XLstatus xlMost150SwitchEventSources (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle
    unsigned int sourceMask
)
```

Description

Switches the different MOST150 events (like data packets or control messages) depending on the license on/off. Events from closed channels are not transmitted to the PC.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **sourceMask**

This flag describes the switched events (event will be passed when bit is set).

Free library:

`XL_MOST150_SOURCE_SPECIAL_NODE`

Switch on the `XL_MOST150_SPECIAL_NODE_INFO_EV` events.

`XL_MOST150_SOURCE_LIGHTLOCK_INIC`

Switch on the `XL_MOST150_RXLIGHT_LOCKSTATUS_EV` events.

`XL_MOST150_SOURCE_ECL_CHANGE`

Switch on the `XL_MOST150_ECL_EV` events.

`XL_MOST150_ECL_TERMINATION_CHANGED`

Switch on the `XL_MOST150_ECL_TERMINATION_EV` events.

`XL_MOST150_SOURCE_CTRL_MLB`

Switch on the `XL_MOST150_CTRL_RX_EV` events.

`XL_MOST150_SOURCE_ASYNC_MLB`

Switch on the `XL_MOST150_ASYNC_RX_EV` events.

`XL_MOST150_SOURCE_ETH_MLB`

Switch on the `XL_MOST150_ETH_RX_EV` events.

`XL_MOST150_SOURCE_TXACK_MLB`

Switch on the `XL_MOST150_CTRL_TX_ACK_EV`, `XL_MOST150_ASYNC_TX_ACK_EV` and `XL_MOST150_ETH_TX_ACK_EV` events.

MOST150 analysis library:

`XL_MOST150_SOURCE_SYNC_ALLOC_INFO`

Switch on the `XL_MOST150_SYNC_ALLOC_INFO_EV` events.

`XL_MOST150_SOURCE_CTRL_SPY`

Switch on the `XL_MOST150_CTRL_SPY_EV` events.

`XL_MOST150_SOURCE_ASYNC_SPY`

Switch on the `XL_MOST150_ASYNC_SPY_EV` events.

`XL_MOST150_SOURCE_ETH_SPY`

Switch on the `XL_MOST150_ETH_SPY_EV` events.

`XL_MOST150_SOURCE_SHUTDOWN_FLAG`

Switch on the `XL_MOST150_SHUTDOWN_FLAG_EV` events.

`XL_MOST150_SOURCE_SYSTEMLOCK_FLAG`

Switch on the `XL_MOST150_SYSTEMLOCK_FLAG_EV` events.

`XL_MOST150_SOURCE_LIGHT_STRESS`

Switch on the `XL_MOST150_GEN_LIGHT_ERROR_EV` events.

`XL_MOST150_SOURCE_LOCK_STRESS`

Switch on the `XL_MOST150_GEN_LOCK_ERROR_EV` events.

`XL_MOST150_SOURCE_BUSLOAD_CTRL`

Switch on the `XL_MOST150_CTRL_BUSLOAD_EV` events.

`XL_MOST150_SOURCE_BUSLOAD_ASYNC`

Switch on the `XL_MOST150_ASYNC_BUSLOAD_EV` events.

`XL_MOST150_SOURCE_STREAM_UNDERFLOW`

switch on the Tx Stream underflow events.

`XL_MOST150_SOURCE_STREAM_OVERFLOW`

switch on the Rx Stream overflow events.

`XL_MOST150_SOURCE_STREAM_RX_DATA`

switch on the Rx Stream data events.

`XL_MOST150_SOURCE_ECL_SEQUENCE`

switch on the ECL sequence events.

Return event `XL_MOST150_EVENT_SOURCE`

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.2 `xIMost150SetDeviceMode`

Syntax

```
XLstatus xIMost150SetDeviceMode (
    XLportHandle portHandle,
```

```

XLaccess      accessMask,
XLuserHandle userHandle,
unsigned int deviceMode
)

```

Description

Sets the timing mode (timing master / timing slave / bypass).

**Note**

In case the timing mode is switched from timing master to timing slave and vice versa, a shutdown is performed by the VN2640 since INIC can only switch from master to slave and vice versa in 'NetOff' state (refer to INIC User Manual). After timing mode was switched, the application has to perform a wake up if required. We always recommend performing a shutdown by calling `xIMost150Shutdown()` to set INIC in NetOff state prior switching the device mode from master to slave and vice versa.

Input parameters**> portHandle**

The port handle retrieved by `xIOpenPort()`.

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xIGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> userHandle

The handle is created by the application and is used for the event assignment.

> deviceMode

Describes the timing mode.

```

XL_MOST150_DEVICEMODE_SLAVE
XL_MOST150_DEVICEMODE_MASTER
XL_MOST150_DEVICEMODE_STATIC_MASTER
XL_MOST150_DEVICEMODE_RETIMED_BYPASS_SLAVE
XL_MOST150_DEVICEMODE_RETIMED_BYPASS_MASTER

```

Return event

`XL_MOST150_DEVICE_MODE`

Return value

Returns an error code (see section **Error Codes** on page 423).

11.4.3 xIMost150GetDeviceMode

Syntax

```

XLstatus xIMost150GetDeviceMode (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle userHandle
)

```

Description

Requests the timing mode (timing master / timing slave / bypass).

Input parameters**> portHandle**

The port handle retrieved by `xIOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

Return event XL_MOST150_DEVICE_MODE

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.4 xlMost150SetSPDIFMode

Syntax

```
XLstatus xlMost150SetSPDIFMode (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle,
    unsigned int   spdifMode
)
```

Description Sets the S/PDIF mode either as S/PDIF master and S/PDIF slave.

Input parameters

> **portHandle**

The port handle retrieved by [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **spdifMode**

Describes the S/PDIF mode.

```
XL_MOST150_SPDIF_MODE_SLAVE
XL_MOST150_SPDIF_MODE_MASTER
```

Return event XL_MOST150_SPDIFMODE

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.5 xlMost150GetSPDIFMode

Syntax

```
XLstatus xlMost150GetSPDIFMode (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle
)
```

Description Requests the S/PDIF mode either as S/PDIF master and S/PDIF slave.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.

Return event

XL_MOST150_SPDIFMODE

Return valueReturns an error code (see section `Error Codes` on page 423).**11.4.6 xlMost150SetSpecialNodeInfo****Syntax**

```
XLstatus xlMost150SetSpecialNodeInfo (
    XLportHandle          portHandle,
    XLaccess              accessMask,
    XLuserHandle          userHandle,
    XLmost150SetSpecialNodeInfo *pSpecialNodeInfo
)
```

Description

Sets the node address, group address, synchronous bandwidth control, retry parameter for the control and packet channel and the MAC address.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **pSpecialNodeInfo**
Contains all data (see section `XLmost150SetSpecialNodeInfo` on page 268).

Return event

XL_MOST150_SPECIAL_NODE_INFO

Return valueReturns an error code (see section `Error Codes` on page 423).**11.4.7 xlMost150GetSpecialNodeInfo****Syntax**

```
XLstatus xlMost150GetSpecialNodeInfo (
    XLportHandle portHandle,
    XLaccess      accessMask,
```

```

    XLuserHandle userHandle,
    unsigned int requestMask
)

```

Description Requests the node address, group address, synchronous bandwidth control, retries parameters for the control and packet channel and the MAC address. Additionally, the node position, the number of devices, and the NetInterface state from INIC can be requested.

Input parameters**> portHandle**

The port handle retrieved by `xlOpenPort()`.

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> userHandle

The handle is created by the application and is used for the event assignment.

> requestMask

Mask of the values to be requested.

```

XL_MOST150_NA_CHANGED
XL_MOST150_GA_CHANGED
XL_MOST150_NPR_CHANGED
XL_MOST150_MPR_CHANGED
XL_MOST150_SBC_CHANGED
XL_MOST150_CTRL_RETRY_PARAMS_CHANGED
XL_MOST150_ASYNC_RETRY_PARAMS_CHANGED
XL_MOST150_MAC_ADDR_CHANGED
XL_MOST150_NPR_SPY_CHANGED (only MOST150 Analysis Library)
XL_MOST150_MPR_SPY_CHANGED (only MOST150 Analysis Library)
XL_MOST150_SBC_SPY_CHANGED (only MOST150 Analysis Library)
XL_MOST150_INIC_NISTATE_CHANGED

```

Return event

`XL_MOST150_SPECIAL_NODE_INFO`

Return value

Returns an error code (see section `Error Codes` on page 423).

11.4.8 `xIMost150SetFrequency`

Syntax

```

XLstatus xlMost150SetFrequency (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int frequency
)

```

Description

Sets the frame rate of the MOST network.

**Note**

Switching the frequency will lead to a broken connection to INIC. Therefore some send requests may get lost and no Tx acknowledge event will be reported. So we recommend always stop sending and perform a shutdown before switching the frequency.

Input parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **frequency**

Frame rate in kHz.

`XL_MOST150_FREQUENCY_44100`

`XL_MOST150_FREQUENCY_48000`

Return event

`XL_MOST150_FREQUENCY`

Return value

Returns an error code (see section `Error Codes` on page 423).

11.4.9 `xlMost150GetFrequency`

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150GetFrequency (
    XLportHandle portHandle,
    XLaaccess accessMask,
    XLuserHandle userHandle
)
```

Description

Requests the configured frame rate of the MOST network.

Input parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

Return event

`XL_MOST150_FREQUENCY`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.10 xlMost150GetSystemLockFlag

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150GetSystemLockFlag (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
)
```

Description

Requests the state of the `SystemLock` flag detected by the spy.

Input parameters

- > **portHandle**

The port handle retrieved by `xlOpenPort()`.

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **userHandle**

The handle is created by the application and is used for the event assignment.

Return event

`XL_MOST150_SYSTEMLOCK_FLAG`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.11 xlMost150GetShutdownFlag

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150GetShutdownFlag (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle
)
```

Description

Requests the state of the shutdown flag detected by the spy.

Input parameters

- > **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

Return event XL_MOST150_SHUTDOWN_FLAG

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.12 xlMost150Shutdown

Syntax

```
XLstatus xlMost150Shutdown (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle
)
```

Description

Performs a shutdown of the network, by calling the function `INIC.NWShutdown()`. The INIC then first sets the shutdown flag and starts the timer `tSSO_Shutdown` (100 ms). As soon as the `tSSO_Shutdown` expires, the MOST signal will be switched off. This does not include sending of `NetBlock.Shutdown()` messages.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

Return event XL_MOST150_NW_SHUTDOWN

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.13 xlMost150Startup

Syntax

```
XLstatus xlMost150Startup (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle
)
```

Description

Performs a start of the network, by calling the function `INIC.NWStartup()`. The INIC will perform a startup depending on the timing mode as described in the MOST Specification.

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment.
Return event	<code>XL_MOST150_NW_STARTUP</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.14 `xlMost150SetSSOResult`

Syntax	<pre>xlStatus xlMost150SetSSOResult (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle, unsigned int ssoCUstatus)</pre>
Description	Sets the "Sudden Signal Off" (SSO) result value - needed for resetting the value to 0x00 (no result) after a shutdown result analysis has been done.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > ssoCUStatus SSO result value to be set. Only <code>XL_MOST150_SSO_RESULT_NO_RESULT</code> is allowed.
Return event	<code>XL_MOST150_SSO_RESULT</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.15 `xlMost150GetSSOResult`

Syntax	<pre>xlStatus xlMost150GetSSOResult (XLportHandle portHandle, XLaccess accessMask,</pre>
--------	--

```
    XLuserHandle userHandle,
)
```

Description	Requests the stored SSO result value.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment.
Return event	<code>XL_MOST150_SSO_RESULT</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.16 `xlMost150CtrlTransmit`

Syntax	<pre style="background-color: #f0f0f0; padding: 5px;">XLstatus xlMost150CtrlTransmit (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle, XLmost150CtrlTxMsg *pCtrlTxMsg)</pre>
Description	Transmits a message over the control channel. The transmit confirmation is reported as <code>XL_MOST150_CTRL_TX_ACK_EV</code> .
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > pCtrlTxMsg Control message to be transmitted (see section <code>XLmost150CtrlTxMsg</code> on page 267).
Return event	<code>XL_MOST150_CTRL_TX_ACK</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.17 xlMost150AsyncTransmit

Syntax

```
XLstatus xlMost150AsyncTransmit (
    XLportHandle      portHandle,
    XLaaccess         accessMask,
    XLuserHandle      userHandle,
    XLmost150AsyncTxMsg *pAsyncTxMsg
)
```

Description

Transmits a data packet (MDP) over the asynchronous channel und returns the point of time of transmission as confirmation. The transmit confirmation is reported as `XL_MOST150_ASYNC_TX_ACK`.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **pAsyncTxMsg**
Asynchronous packet to be transmitted (see section `XLmost150AsyncTxMsg` on page 274).

Return event

`XL_MOST150_ASYNC_TX_ACK`

Return value

Returns an error code (see section `Error Codes` on page 423).

11.4.18 xlMost150EthernetTransmit

Syntax

```
XLstatus xlMost150EthernetTransmit (
    XLportHandle      portHandle,
    XLaaccess         accessMask,
    XLuserHandle      userHandle,
    XLmost150EthernetTxMsg *pEthernetTxMsg
)
```

Description

Transmits an Ethernet packet (MEP) over the asynchronous channel und returns the point of time of transmission as confirmation. The transmit confirmation is reported as `XL_MOST150_ETHERNET_TX_ACK_EV`.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **pEthernetTxMsg**
Ethernet packet to be transmitted (see section [XLmost150EthernetTxMsg](#) on page 275).

Return event XL_MOST150_ETHERNET_TX_ACK

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.19 xlMost150SyncGetAllocTable



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150SyncGetAllocTable (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle
)
```

Description

Requests allocation information for synchronous channels.

Input parameters

- > **portHandle**
The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.

Return event

XL_MOST150_SYNC_ALLOC_INFO

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.20 xlMost150CtrlSyncAudio

Syntax

```
XLstatus xlMost150CtrlSyncAudio (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    XLmost150SyncAudioParameter *pSyncAudioParameter
)
```

Description

Defines the channels for synchronous input/output including analog signals (line in / out) as well as digital signals (S/PDIF in/out). The channel routing is done by the INIC. Additionally only bandwidth can be allocated without routing data.

Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > pSyncAudioParameter Audio parameter to be transmitted (see section <code>XLmost150SyncAudioParameter</code> on page 270).
Return event	<code>XL_MOST150_CTRL_SYNC_AUDIO</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.21 `xlMost150SyncSetVolume`

Syntax	<pre>XLstatus xlMost150SyncSetVolume (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle, unsigned int device, unsigned int volume)</pre>
Description	Sets the input gain of the device (line in/out). 100% means maximum level, 0% minimum level (no level).
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > device <code>XL_MOST150_DEVICE_LINE_IN</code> <code>XL_MOST150_DEVICE_LINE_OUT</code> > volume Value range 0...255 (means 0%...100%).
Return event	<code>XL_MOST150_SYNC_VOLUME_STATUS</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.22 xlMost150SyncGetVolume

Syntax

```
XLstatus xlMost150SyncGetVolume (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int device
)
```

Description Requests the input gain of line in/out ports.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **device**

`XL_MOST150_DEVICE_LINE_IN`
`XL_MOST150_DEVICE_LINE_OUT`

Return event

`XL_MOST150_SYNC_VOLUME_STATUS`

Return value

Returns an error code (see section **Error Codes** on page 423).

11.4.23 xlMost150SyncSetMute

Syntax

```
XLstatus xlMost150SyncSetMute (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int device,
    unsigned int mute
)
```

Description

Sets the mute state of the audio device.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

- > **device**
 XL_MOST150_DEVICE_LINE_IN
 XL_MOST150_DEVICE_LINE_OUT
 XL_MOST150_DEVICE_SPDIF_IN
 XL_MOST150_DEVICE_SPDIF_OUT
- > **mute**
 XL_MOST150_NO_MUTE
 XL_MOST150_MUTE

Return event XL_MOST150_SYNC_MUTE_STATUS

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.24 xlMost150SyncGetMute

Syntax

```
XLstatus xlMost150SyncGetMute (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int device
)
```

Description Requests the mute state of a given audio device.

Input parameters

- > **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **userHandle**

The handle is created by the application and is used for the event assignment.

- > **device**

- XL_MOST150_DEVICE_LINE_IN
- XL_MOST150_DEVICE_LINE_OUT
- XL_MOST150_DEVICE_SPDIF_IN
- XL_MOST150_DEVICE_SPDIF_OUT

Return event XL_MOST150_SYNC_MUTE_STATUS

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.25 xlMost150GetRxLightLockStatus

Syntax

```
XLstatus xlMost150GetRxLightLockStatus (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int fromSpy
)
```

Description	Requests light & lock state either from INIC (light state at FOT and the PLL state) or from the spy.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > fromSpy Indicates whether the light & lock state should be retrieved from the spy or the node (INIC). 0: Request light & lock status from INIC. 1: Request light & lock status from SPY.
Return event	<code>XL_MOST150_RXLIGHT_LOCKSTATUS</code> The <code>flagsChip</code> member in the event header determines whether the event is from spy (see <code>flagsChip</code> parameter values).
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.26 `xlMost150SetTxLight`



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax	<pre>XLstatus xlMost150SetTxLight (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle, unsigned int txLight)</pre>
Description	Sets light status at FOT.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment.

> **txLight**

Tx light status at FOT.

`XL_MOST150_LIGHT_OFF`

`XL_MOST150_LIGHT_FORCE_ON` (currently not supported!)

`XL_MOST150_LIGHT_MODULATED`

Return event

`XL_MOST150_TX_LIGHT`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.27 xlMost150GetTxLight

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150GetTxLight (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle
)
```

Description

Requests light status at FOT.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

Return event

`XL_MOST150_TX_LIGHT`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.28 xlMost150SetTxLightPower

Syntax

```
XLstatus xlMost150SetTxLightPower (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int attenuation
)
```

Description

Sets the attenuation of the modulated light at FOT.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **attenuation**

`XL_MOST150_LIGHT_FULL`
`XL_MOST150_LIGHT_3DB`

Return event `XL_MOST150_LIGHT_POWER`

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.29 `xIMost150GenerateLightError`


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xIMost150GenerateLightError (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int lightOffTime,
    unsigned int lightOnTime,
    unsigned int repeat
)
```

Description

Starts/stops the generation of light-off/on changes. Point of time of start and stop are signalled to the application by `XL_MOST150_GEN_LIGHT_ERROR_EV` events.

Input parameters
> **portHandle**

The port handle retrieved by `xIOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **lightOffTime**

Time of unmodulated light emission in [ms].

> **lightOnTime**

Time of modulated light emission in [ms].

> **repeat**

The value determines the number of changes that will be generated:

0: Light (ON/OFF) changes stopped

>0: Light (ON/OFF) changes started.

The changes are generated continuously:

0xFFFFFFFF: Light (ON/OFF) changes started.

Return event XL_MOST150_GEN_LIGHT_ERROR

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.30 xlMost150GenerateLockError

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150GenerateLockError (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int unlockTime,
    unsigned int lockTime,
    unsigned int repeat
)
```

Description

Starts/stops the generation of light unmodulated/modulated changes. Point of time of start and stop are signalled to the application by [XL_MOST150_GEN_LOCK_ERROR_EV](#) events.

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **unlockTime**

Unlock duration in [ms].

> **lockTime**

Lock duration in [ms].

> **repeat**

0: Stop generation

>0: Number of changes.

0xFFFFFFFF: Generation of continual changes.

Return event

XL_MOST150_GEN_LOCK_ERROR

Return value	Returns an error code (see section Error Codes on page 423).
--------------	--

11.4.31 xlMost150CtrlConfigureBusload



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150CtrlConfigureBusload (
    XLportHandle          portHandle,
    XLaccess              accessMask,
    XLuserHandle          userHandle,
    XLmost150CtrlBusloadConfig *pCtrlBusLoad
)
```

Description

Configures busload generation with MOST control messages.

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **pCtrlBusLoad**

Pointer to structure `XLmost150CtrlBusloadConfig` containing the control message used for busload generation and configuration, its storage has to be supplied by the caller.

Note: The INIC will only send valid control messages, i.e. FBlockID..TelLen have to be correct. A counter will only be available in the payload bytes.

Return event

None.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.32 xlMost150CtrlGenerateBusload



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150CtrlGenerateBusload (
    XLportHandle  portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle,
    unsigned long numberCtrlFrames
)
```

Description	Starts/stops busload generation with MOST control messages.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > numberCtrlFrames 0: Stop busload generation. >0: Number of busload control messages 0xFFFFFFFF (-1): Infinite number of messages.
Return event	<code>XL_MOST150_CTRL_BUSLOAD</code>
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.33 `xlMost150AsyncConfigureBusload`



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax	<pre>XLstatus xlMost150AsyncConfigureBusload (XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle, XLmost150AsyncBusloadConfig *pAsyncBusLoad)</pre>
Description	Configures busload generation of MOST Data or Ethernet packets.
Input parameters	<ul style="list-style-type: none"> > portHandle The port handle retrieved by <code>xlOpenPort()</code>. > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment. > pAsyncBusLoad Pointer to an <code>XLmost150AsyncBusloadConfig</code> structure containing the asynchronous message used for busload generation and configuration, its storage has to be supplied by the caller.

Return event	None.
Return value	Returns an error code (see section Error Codes on page 423).

11.4.34 xlMost150AsyncGenerateBusload



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150AsyncGenerateBusload (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned long numberAsyncPackets
)
```

Description

Starts/stops busload generation with MOST Data or Ethernet packets.



Note

In case the bandwidth of the asynchronous channel is changed, any running MDP or MEP busload is automatically stopped.

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **numberAsyncPackets**

0: Stop busload generation.

>0: Number of busload packets

0xFFFFFFFF (-1): Infinite number of packets.

Return event

`XL_MOST150_ASYNC_BUSLOAD`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.35 xlMost150ConfigureRxBuffer



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150ConfigureRxBuffer (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
```

```
    unsigned int bufferType,
    unsigned int bufferMode
)
```

Description Configures the receive buffer for control messages and packets of the INIC.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **bufferType**

Bitmask which specifies the receive buffer type.

`XL_MOST150_RX_BUFFER_TYPE_CTRL`

`XL_MOST150_RX_BUFFER_TYPE_ASYNC`

> **bufferMode**

Block or unblock processing the respective receive buffer.

`XL_MOST150_RX_BUFFER_NORMAL_MODE`

`XL_MOST150_RX_BUFFER_BLOCK_MODE`

Return event

`XL_MOST150_CONFIGURE_RX_BUFFER`

Return value

Returns an error code (see section `Error Codes` on page 423).

11.4.36 `xlMost150GenerateBypassStress`



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150GenerateBypassStress (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int bypassCloseTime,
    unsigned int bypassOpenTime,
    unsigned int repeat
)
```

Description

Starts/stops the generation of bypass close/open changes.

**Note**

The bypass stress can only be started in case the VN2640 device mode is currently `XL_MOST150_DEVICEMODE_SLAVE` or `XL_MOST150_DEVICEMODE_RETIMED_BYPASS_SLAVE` and the MOST network is already started up, i. e. the NetInterface is in NetOn state.

Additionally, the bypass stress is automatically stopped in case the network is shutdown or the device mode is set through `xlMost150SetDeviceMode()`. The value range for the bypass close / open duration is: 10..65535 ms

Input parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **bypassCloseTime**

Time the bypass is closed in [ms].

> **bypassOpenTime**

Time the bypass is opened in [ms].

> **repeat**

0: Stop Bypass (close/open) changes.

>0: Start Bypass (close/open) changes with given number of changes.

`0xFFFFFFFF (-1)`: Start Bypass (close/open) changes with infinite number of changes.

Return event

`XL_MOST150_GEN_BYPASS_STRESS`

Return value

Returns an error code (see section **Error Codes** on page 423).

11.4.37 xlMost150SetECLLine**Syntax**

```
XLstatus xlMost150SetECLLine (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int eclLineState
)
```

**Note**

In case the ECL is pulled down to low level by another device, it cannot be pulled up to high level!

Input parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **eclLineState**

The new ECL line state.

`XL_MOST150_ECL_LINE_LOW`
`XL_MOST150_ECL_LINE_HIGH`

Return event `XL_MOST150_ECL_LINE_CHANGED`

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.38 IMost150SetECLTermination

Syntax

```
XLstatus xlMost150SetECLTermination (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int eclLineTermination
)
```

Description Sets the ECL line termination resistor.

Input parameters

> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **eclLineTermination**

The new ECL termination.

`XL_MOST150_ECL_LINE_PULL_UP_NOT_ACTIVE`
`XL_MOST150_ECL_LINE_PULL_UP_ACTIVE`

Return event `XL_MOST150_ECL_TERMINATION_CHANGED`

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.39 xlMost150GetECLInfo

Syntax

```
XLstatus xlMost150GetECLInfo (
    XLportHandle portHandle,
```

```

XLaccess      accessMask,
XLuserHandle  userHandle
)

```

Description Requests the ECL Info (ECL line and ECL termination resistor state as well as the glitch filter setting).

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

Return event

```

XL_MOST150_ECL_LINE_CHANGED,
XL_MOST150_ECL_TERMINATION_CHANGED,
XL_MOST150_ECL_GLITCH_FILTER

```

Return value

Returns an error code (see section **Error Codes** on page 423).

11.4.40 `xlMost150ECLConfigureSeq`



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```

XLstatus xlMost150ECLConfigureSeq (
    XLportHandle  portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle,
    unsigned int   numStates,
    unsigned int*  pEclStates,
    unsigned int*  pEclStatesDuration
)

```

Description

Configure a sequence for the ECL line (e. g. to trigger a System Test). The sequence can be triggered by calling `xlMost150EclGenerateSeq()`.



Note

In case the ECL glitch filter is configured such that short pulses are filtered, no `XL_MOST150_ECL_EV` event will be reported during the sequence.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **numStates**

Number of ECL states (max. 200).

> **pEclStates**

Pointer to a buffer containing the ECL sequence states (1: High, 0: Low).

> **pEclStatesDuration**

Pointer to a buffer containing the ECL sequence states duration in multiple of 100 µs. Value range: 1 ... 655350 → 100 µs ... 65535 ms.

Return event None.

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.41 xlMost150ECLGenerateSeq

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150ECLGenerateSeq (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle,
    unsigned int   start
)
```

Description

Starts or stops a previously configured ECL sequence.

**Note**

In case the ECL is pulled down to low level before (or during) the sequence, no (further) [XL_MOST150_ECL_EV](#) event will be reported. The ECL remains in low level state.

Input parameters

> **portHandle**

The port handle retrieved by [xiOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

- > **start**
0: Stop ECL sequence
1: Start ECL sequence

Return event XL_MOST150_ECL_SEQUENCE

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.42 xlMost150SetECLGlitchFilter



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150SetECLGlitchFilter (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int duration
)
```

Description

Configures the glitch filter for detecting ECL line state changes.



Note

The higher the duration the more short pulses (up to 50 ms) will not be reported by an XL_MOST150_ECL_EV event.

Input parameters

- > **portHandle**
The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **duration**
Duration (in μ s) of glitches to be filtered. Value range: 50 μ s .. 50 ms
Default value: 1 ms

Return event XL_MOST150_ECL_GLITCH_FILTER

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.43 Streaming

11.4.43.1 General Information

**Note**

This feature is available in the MOST 150 Analysis Library only.

Streaming functions The streaming functions of the XL MOST150 API can be used for transmission of data from or to synchronous MOST channels. Minimum requirements are a VN2640 interface and USB2.0. The streaming functionality is only available in MOST150 Analysis Library.

Tx Stream

The VN2640 allows one Tx stream with a bandwidth of 1...152 byte per MOST frame. The driver's FIFO size for transmitting streaming data is 8 MB.

**Step by Step Procedure**

1. Initially a stream has to be opened by calling `xIMost150StreamOpen()`. The stream handle is valid if the return value is `XL_SUCCESS`.
2. As soon as the event `XL_MOST150_STREAM_STATE(state = XL_MOST150_STREAM_STATE_OPENED)` is received, the application may prepare buffers for sending stream data. The desired bandwidth is allocated and the respective connection label is reported by a `XL_MOST150_STREAM_TX_LABEL` event.
3. The application may provide data by calling `xIMost150StreamTransmitData()` before starting the stream, thus avoiding to stream "0" data initially just after starting the stream.
4. The stream is started by calling `xIMost150StreamStart()`. The successful start is acknowledged with an `XL_MOST150_STREAM_STATE` event (`state = XL_MOST150_STREAM_STATE_STARTED`).
5. The application is then cyclically informed by a `MOST150_STREAM_TX_BUFFER` event to provide further streaming data to be transmitted by calling `xIMost150StreamTransmitData()`. This cyclic notification is done until the stream is stopped.
6. The stream is stopped by calling `xIMost150StreamStop()`. This is acknowledged with an `XL_MOST150_STREAM_STATE` event (`state = XL_MOST150_STREAM_STATE_STOPPED`).
7. The stream is closed by calling `xIMost150StreamClose()`. This is acknowledged with an `XL_MOST150_STREAM_STATE` event (`state = XL_MOST150_STREAM_STATE_CLOSED`). The allocated bandwidth is freed.

Rx Stream

The VN2640 allows one Rx stream with up to 8 connection labels. The driver's FIFO size for receiving streaming data is 8 MB.



Step by Step Procedure

1. The application has to call `xIMost150StreamInitRxFifo()` once to initialize the Rx FIFO.
2. Initially a stream has to be opened by calling `xIMost150StreamOpen()`. The stream handle is valid if the return value is `XL_SUCCESS`.
3. As soon as the event `XL_MOST150_STREAM_STATE` (`state = XL_MOST150_STREAM_STATE_OPENED`) the application may prepare buffers for receiving stream data.
4. The stream is started by calling `xIMost150StreamStart()`. The successful start is acknowledged with an `XL_MOST150_STREAM_STATE` event (`state = XL_MOST150_STREAM_STATE_STARTED`).
5. The application is then cyclically informed by an `XL_MOST150_STREAM_RX_BUFFER` event that streaming data is available in the Rx FIFO. Streaming data can be read out by calling `xIMost150StreamReceiveData()`. This cyclic notification is done until the stream is stopped.
6. The stream is stopped by calling `xIMost150StreamStop()`. This is acknowledged with an `XL_MOST150_STREAM_STATE` event (`state = XL_MOST150_STREAM_STATE_STOPPED`). A last `XL_MOST150_STREAM_RX_BUFFER` event may be reported to the application.
7. The stream is closed by calling `xIMost150StreamClose()`. This is acknowledged with an `XL_MOST150_STREAM_STATE` event (`state = XL_MOST150_STREAM_STATE_CLOSED`).

Clearing Tx FIFO

The application is able to clear the driver's Tx FIFO by calling `xIMost150StreamClearTxFifo()`. This can be used by the application e.g. to simulate a track change of a disc player.

Over- and underflow

In case the application does not process the `XL_MOST150_STREAM_RX_BUFFER` events fast enough, an overflow might occur leading to a loss of streaming data. This is reported in the status field of the event by the `XL_MOST150_STREAM_BUFFER_ERROR_OVERFLOW` flag. In case the application does not process the `XL_MOST150_STREAM_TX_BUFFER` events in time to provided further data, an underflow might occur which is reported by an `XL_MOST150_STREAM_TX_UNDERFLOW` event.

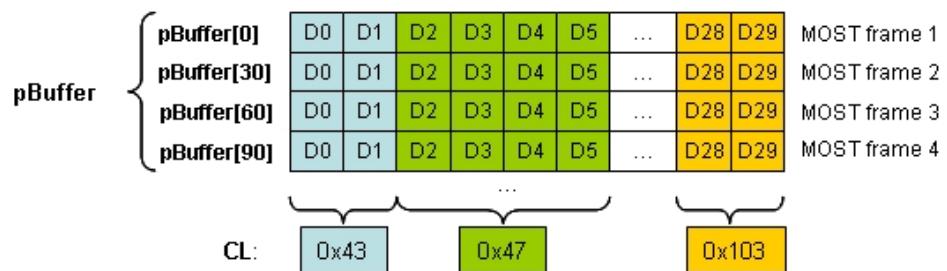
11.4.43.2 Layout of Streaming Data

Tx

The format of the Tx streaming data is in raw format. This means that every byte of the buffer is fed into the INIC in the given order. Please also remark that the data should be MOST frame aligned in order to keep the correct format e.g. for a 24 bit stereo audio signal.

Rx

The format of the Rx streaming data is in raw format and always MOST frame aligned. The streaming data is arranged by connection labels in the order as the labels are given by application (refer to `xIMost150StreamStart()`). The number of bytes (width) per connection label is reported to the application by an `XL_MOST150_SYNC_ALLOC_INFO` event. Thus the application can determine which byte belongs to which connection label. Example: Totally 30 bytes per MOST frame are streamed with labels 0x0043, 0x0047, ..0x0103 given in `xIMost150StreamStart()`.



11.4.44 xlMost150StreamOpen



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150StreamOpen (
    XLportHandle      portHandle,
    XLaccess          accessMask,
    XLuserHandle      userHandle,
    XLmost150StreamOpen* pStreamOpen
)
```

Description

Opens a stream (Tx / Rx) for routing synchronous data to or from the MOST bus (synchronous channel). Additionally for a Tx stream, the desired bandwidth will be allocated.

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **pStreamOpen**
Pointer to `XLmost150StreamOpen` structure.

Return event

`XL_MOST150_STREAM_STATE`

Return value

Returns an error code (see section `Error Codes` on page 423).

11.4.45 xlMost150StreamClose



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150StreamClose (
    XLportHandle portHandle,
```

```

XLaccess      accessMask,
XLuserHandle userHandle,
unsigned int streamHandle
)

```

Description

Closes an opened a stream (Tx / Rx) used for routing synchronous data to or from the MOST bus (synchronous channel). Additionally for a Tx stream, the allocated bandwidth will be freed.

Input parameters**> portHandle**

The port handle retrieved by `xlOpenPort()`.

> accessMask

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> userHandle

The handle is created by the application and is used for the event assignment.

> streamHandle

Stream handle (returned by `xlMost150StreamOpen()`).

Return event

`XL_MOST150_STREAM_STATE`

Return value

Returns an error code (see section `Error Codes` on page 423).

11.4.46 xlMost150StreamStart

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```

XLstatus xlMost150StreamStart (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle userHandle,
    unsigned int streamHandle,
    unsigned int numConnLabels,
    unsigned int* pConnLabels
)

```

Description

Starts the streaming (Tx / Rx) of synchronous data to or from the MOST bus (synchronous channel). The application will cyclically be informed either by `XL_MOST150_STREAM_TX_BUFFER_EV` events to provide further streaming data or `XL_MOST150_STREAM_RX_BUFFER_EV` events to read out received streaming data by calling `xlMost150StreamReceiveData()`. The event type depends on the stream direction set in `xlMost150StreamOpen()`.

Input parameters**> portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Stream handle (returned by [xIMost150StreamOpen\(\)](#)).

> **numConnLabels** (only used for Rx Streaming!)

Number of connection labels to be streamed. Currently maximum 8 CLs can be streamed at a time.

> **pConnLabels** (only used for Rx Streaming!)

Pointer to a buffer containing the connection labels.

Return event XL_MOST150_STREAM_STATE

Return value Returns an error code (see section [Error Codes](#) on page 423).

11.4.47 xIMost150StreamStop


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xIMost150StreamStop (
    XLportHandle portHandle,
    XLaccess      accessMask,
    XLuserHandle  userHandle,
    unsigned int   streamHandle
)
```

Description

Stops the streaming (Tx / Rx) of synchronous data to or from the MOST bus (synchronous channel). For Rx Streaming the application gets informed about the last received data by an [XL_MOST150_STREAM_RX_BUFFER_EV](#) event.

Input parameters
> **portHandle**

The port handle retrieved by [xIOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xIGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Stream handle (returned by [xIMost150StreamOpen\(\)](#)).

Return event XL_MOST150_STREAM_STATE

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.48 xlMost150StreamTransmitData

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150StreamTransmitData (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int streamHandle,
    unsigned char* pBuffer,
    unsigned int* pNumberOfBytes
)
```

Description

This function passes a buffer containing the transmit data to be streamed. In case this function is called several times in a row, the driver appends the data to Tx FIFO in the same order as it is passed by the application. An `XL_MOST150_STREAM_TX_BUFFER_EV` event is used to inform the application that further data can be inserted into the Tx FIFO.

Input parameters

> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Stream handle (returned by `xlMost150StreamOpen()`).

> **pBuffer**

Pointer to a buffer containing the data to be streamed (PC → MOST).

> **pNumberOfBytes**

Pointer to a buffer containing:

IN: Number of bytes in the buffer `pBuffer`.

OUT: Number of bytes actually copied from the buffer `pBuffer`.

Return event

None.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.49 xlMost150StreamClearTxFifo

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150StreamClearTxFifo (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int streamHandle
)
```

Description

This function can be used to clear the Tx FIFO in the driver in order to perform a fast muting or to simulate a CD track change, without stopping and re-starting the stream.

Input parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **streamHandle**

Stream handle (returned by `xlMost150StreamOpen()`).

Return event

None.

Return value

Returns an error code (see section **Error Codes** on page 423).

11.4.50 `xlMost150StreamInitRxFifo`

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150StreamInitRxFifo (
    XLportHandle portHandle,
    XLaccess accessMask
)
```

Description

This function initializes the Rx FIFO in the driver and should be called once before initializing the Rx stream. In case this function is not called, Rx Streaming cannot be started.

Input parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

Return event

None.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.51 xlMost150StreamReceiveData

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150StreamReceiveData (
    XLportHandle    portHandle,
    XLaccess        accessMask,
    XLuserHandle    userHandle
    unsigned char*  pBuffer,
    unsigned int*   pBufferSize
)
```

Description

This function fetches the received streaming data from the Rx FIFO. The application is notified to call this function by an [XL_MOST150_STREAM_RX_BUFFER_EV](#) event.

Input parameters> **portHandle**

The port handle retrieved by [xlOpenPort\(\)](#).

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **userHandle**

The handle is created by the application and is used for the event assignment.

> **pBuffer**

Pointer to a buffer into which the received data should be stored.

> **pBufferSize**

Pointer to a buffer containing:

IN: Size of the buffer `pBuffer`.

OUT: Number of bytes actually copied into the buffer `pBuffer` (<= input size).

Return event

None.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.52 xlMost150StreamGetInfo

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
XLstatus xlMost150StreamGetInfo (
    XLportHandle      portHandle,
    XLaccess         accessMask,
    XLuserHandle     userHandle,
```

```
XLmost150StreamInfo* pStreamInfo  
)
```

Description	This function retrieves the streaming information of the respective stream determined by the <code>streamHandle</code> parameter. In case the stream is closed there is no valid stream handle and the function return an error <code>XL_ERR_WRONG_PARAMETER</code> .
Input parameters	<ul style="list-style-type: none">> portHandle The port handle retrieved by <code>xlOpenPort()</code>.> accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23.> userHandle The handle is created by the application and is used for the event assignment.
Output parameters	<ul style="list-style-type: none">> pStreamInfo Pointer to structure <code>XLmost150StreamInfo</code>.
Return event	None.
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

11.4.53 xlMost150Receive

Syntax

```
XLstatus xlMost150Receive (
    XLportHandle portHandle,
    XLmost150event* pEventBuffer
)
```

Description

Reads one event from the MOST150 receive queue. An overrun of the receive queue can be determined by the message flag `XL_MOST150_QUEUE_OVERFLOW` in `XLmost150event.flagsChip`.

Input parameters

- > **portHandle**

The port handle retrieved by `xlOpenPort()`.

- > **pEventBuffer**

Pointer the event buffer (see section [XLmost150event](#) on page 272).

Buffer size: `XL_MOST150_MAX_EVENT_DATA_SIZE`.

Return event

None.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.4.54 xlMost150TwinklePowerLed

Syntax

```
XLstatus xlMost150TwinklePowerLed (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle
)
```

Description

The VN2640 power LED will twinkle three times.

Input parameters

- > **portHandle**

The port handle retrieved by `xlOpenPort()`.

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **userHandle**

The handle is created by the application and is used for the event assignment.

Return event

None.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

11.5 Structs

11.5.1 XLmost150AsyncBusloadConfig

Syntax

```
typedef struct s_xl_most150_async_busload_config {
    unsigned int busloadType;
    unsigned int transmissionRate;
    unsigned int counterType;
    unsigned int counterPosition;

    union {
        unsigned char          rawBusloadPkt[1540];
        XLmost150AsyncTxMsg   busloadAsyncPkt;
        XLmost150EthernetTxMsg busloadEthernetPkt;
    } busloadPkt;
} XLmost150AsyncBusloadConfig;
```

Parameters

> **busloadType**

Specifies whether MOST Data packets (MDP) or MOST Ethernet packets (MEP) should be transmitted.

Values:

XL_MOST150_BUSLOAD_TYPE_DATA_PACKET
XL_MOST150_BUSLOAD_TYPE_ETHERNET_PACKET

> **transmissionRate**

Number of packets per second to be transmitted.

Counter type values:

XL_MOST150_BUSLOAD_COUNTER_TYPE_NONE
XL_MOST150_BUSLOAD_COUNTER_TYPE_1_BYTE
XL_MOST150_BUSLOAD_COUNTER_TYPE_2_BYTE
XL_MOST150_BUSLOAD_COUNTER_TYPE_3_BYTE
XL_MOST150_BUSLOAD_COUNTER_TYPE_4_BYTE

> **counterPosition**

Position in the payload of the MDP (0..1523) / MEP (0..1505).

Note: The counter position depends on the `countertype`:

Counter Type	Counter Position	
	MDP	MEP
1 Byte	0..1523	0..1505
2 Byte	1..1523	1..1505
3 Byte	2..1523	2..1505
4 Byte	3..1523	3..1505

> **busloadAsyncPkt**

See section [XLmost150AsyncTxMsg](#) on page 274.

> **busloadEthernetPkt**

See section [XLmost150EthernetTxMsg](#) on page 275.

11.5.2 XLmost150AsyncTxMsg

Syntax

```
typedef struct s_xl_most150_async_tx_msg {
```

```

    unsigned int priority;
    unsigned int asyncSendAttempts;
    unsigned int length;
    unsigned int targetAddress;
    unsigned char asyncData[XL_MOST150_ASYNC_SEND_PAYLOAD_MAX_SIZE];
} XLmost150AsyncTxMsg;

```

Parameters> **priority**

Transmission priority. Bit 0..3 can be set for priority. However, the INIC currently only accepts the default value of 0x00.

> **asyncSendAttempts**

Transmission send attempts. Value range: 0x01..0x10 (0...15 retries). For using the default send attempt value this parameter has to be set to 0xFF. The default value is set with `XLMost150SetSpecialNodeInfo()` function.

> **length**

Number of bytes.

Note: It is possible to send a data packet with more than 1524 bytes. This can be used for testing purpose. However, the return event `XL_MOST150_ASYNC_TX_ACK` will report a maximum of 1524 byte.

> **targetAddress**

Logical target address of the data packet.

> **asyncData**

Payload data (depending on length).

11.5.3 XLmost150CtrlBusloadConfig**Syntax**

```

typedef struct s_xl_most150_ctrl_busload_config {
    unsigned int      transmissionRate;
    unsigned int      counterType;
    unsigned int      counterPosition;
    XLmost150CtrlTxMsg busloadCtrlMsg;
} XLmost150CtrlBusloadConfig;

```

Parameters> **transmissionRate**

Number of control messages per second to be transmitted.

> **counterType**

Counter type values:

`XL_MOST150_BUSLOAD_COUNTER_TYPE_NONE`
`XL_MOST150_BUSLOAD_COUNTER_TYPE_1_BYTE`
`XL_MOST150_BUSLOAD_COUNTER_TYPE_2_BYTE`
`XL_MOST150_BUSLOAD_COUNTER_TYPE_3_BYTE`
`XL_MOST150_BUSLOAD_COUNTER_TYPE_4_BYTE`

> **counterPosition**

Position in the payload of the control message (0..44).

Note: The counter position depends on the countertype:

In case of a one byte counter, the position can be in the range 0..44.

In case of a two byte counter, the position can only be in the range 1..44.

In case of a three byte counter, the position can only be in the range 2..44.

In case of a four byte counter, the position can only be in the range 3..44.

> **busloadCtrlMsg**

See section [XLmost150CtrlTxMsg](#) on page 267.

11.5.4 XLmost150CtrlTxMsg

Syntax

```
typedef struct s_xl_most150_ctrl_tx_msg {
    unsigned int ctrlPrio;
    unsigned int ctrlSendAttempts;
    unsigned int targetAddress;
    unsigned char ctrlData[51];
} XLmost150CtrlTxMsg;
```

Parameters

> **ctrlPrio**

Transmission priority. Bit 0..3 can be set for priority. However, the INIC currently only accepts the default value of 0x01.

> **ctrlSendAttempts**

Transmission send attempts. Value range: 0x01..0x10 (0...15 retries). For using the default send attempt value this parameter has to be set to 0xFF. The default value is set with [XLMost150SetSpecialNodeInfo\(\)](#) function.

> **targetAddress**

Destination address of the control message.

> **ctrlData**

Contains the control message to be transmitted. The structure is as follows:

FBlockId: 8 bit
 InstId: 8 bit
 FunctionId: 12 bit
 OpType: 4 bit
 TelId: 4 bit
 TelLen: 12 bit
 Payload: 0..45 byte

```
ctrlData[0]: FBlockID
ctrlData[1]: InstID
ctrlData[2]: FunctionID (upper 8 bits)
ctrlData[3]: FunctionID (lower 4 bits) + OpType (4 bits)
ctrlData[4]: TelId (4 bits) + TelLen (upper 4 bits)
ctrlData[5]: TelLen (lower 8 bits)
ctrlData[6..50]: Payload
```

11.5.5 XLmost150EthernetTxMsg

Syntax

```
typedef struct s_xl_most150_ethernet_tx_msg {
    unsigned int priority;
    unsigned int ethSendAttempts;
    unsigned char sourceAddress[6];
    unsigned char targetAddress[6];
    unsigned int length;
    unsigned char ethernetData[XL_MOST150_ETHERNET_
                                SEND_PAYLOAD_MAX_SIZE];
} XLmost150EthernetTxMsg;
```

Parameters

- > **priority**
Priority of the Ethernet packet. Can be 0x0 (for lowest priority) to 0x3 (for highest priority). Currently the INIC only accepts the default value of 0x00.
- > **ethSendAttempts**
Transmission send attempts. Value range: 0x01..0x10 (0...15 retries). For using the default send attempt value this parameter has to be set to 0xFF. The default value is set with `XLMost150SetSpecialNodeInfo()` function.
- > **sourceAddress**
Source MAC address of the Ethernet packet.
- > **targetAddress**
Target MAC address of the Ethernet packet.
- > **length**
Number of data bytes of the Ethernet packet.

Note: It is possible to send an Ethernet packet with more than 1506 payload bytes. This can be used for testing purpose. However, the return event `XL_MOST150_ETHERNET_TX_ACK` will report a maximum of 1506 byte.

- > **ethernetData**
Payload of the Ethernet packet (depends on length).

11.5.6 XLmost150SetSpecialNodeInfo

Syntax

```
typedef struct s_xl_set_most150_special_node_info {
    unsigned int changeMask;
    unsigned int nodeAddress;
    unsigned int groupAddress;
    unsigned int sbc;
    unsigned int ctrlRetryTime;
    unsigned int ctrlSendAttempts;
    unsigned int asyncRetryTime;
    unsigned int asyncSendAttempts;
    unsigned char macAddr[6];
} XLmost150SetSpecialNodeInfo;
```

Parameters

- > **changeMask**
Mask for the changes to be set.
`XL_MOST150_NA_CHANGED`
`XL_MOST150_GA_CHANGED`
`XL_MOST150_SBC_CHANGED`
`XL_MOST150_CTRL_RETRY_PARAMS_CHANGED`
`XL_MOST150_ASYNC_RETRY_PARAMS_CHANGED`
`XL_MOST150_MAC_ADDR_CHANGED`
- > **nodeAddress**
Node address of hardware device.
Value range: 0x0010..0x02FF, 0x0500..0x0FFF, 0xFFFF
- > **groupAddress**
Group address of hardware device.
Value range: 0x0300..0x03FF (excluding: 0x03C8) sbc (only for timing master):
Synchronous bandwidth control in number of quadlets.
Value range: 0x00..0x5D

- > **ctrlRetryTime**
Transmit retry time for control messages in time units of 16 MOST frames.
Value range: 3..31
- > **ctrlSendAttempts**
Default number of send attempts for control messages.
Value range: 1..16
- > **asyncRetryTime**
Transmit retry time for packets (MDP and MEP) in number of MOST frames.
Value range: 0..255
- > **asyncSendAttempts**
Default number of send attempts for packets (MDP and MEP).
Value range: 1..16
- > **macAddr**
MAC address of hardware device.
Value range: complete range.

11.5.7 XLmost150StreamInfo

Syntax

```
typedef struct s_xl_most150_stream_get_info {
    unsigned int streamHandle;
    unsigned int numBytesPerFrame;
    unsigned int direction;
    unsigned int reserved;
    unsigned int latency;
    unsigned int streamState;
    unsigned int connLabels[XL_MOST150_STREAM_RX_NUM_CL_MAX];
} XLmost150StreamInfo;
```

Parameters

- > **streamHandle**
Stream handle returned by xlMost150StreamOpen().
- > **numBytesPerFrame**
Number of bytes per MOST frame which are streamed.
- > **direction**
Streaming direction.
- > **reserved**
Reserved for future use.
- > **latency**
Streaming latency.
- > **streamState**
Current stream state.
- > **connLabels**
Connection label(s) from (Rx) or to (Tx) which data is streamed.

11.5.8 XLmost150StreamOpen

Syntax

```
typedef struct s_xl_most150_stream_open {
    unsigned int* pStreamHandle;
    unsigned int direction;
    unsigned int numBytesPerFrame;
```

```
unsigned int reserved;
unsigned int latency;
} XLmost150StreamOpen;
```

Parameters**> pStreamHandle**

Returns the stream handle in case the stream could successfully be opened.

> direction

Streaming direction.

`XL_MOST150_STREAM_RX_DATA`
`XL_MOST150_STREAM_TX_DATA`

> numBytesPerFrame

Number of bytes per MOST frame to be streamed.

> latency

Streaming latency. This parameter controls the notification of the application and CPU load respectively. There are five latency levels defined:

`XL_MOST150_STREAM_LATENCY_VERY_LOW`
Very low notification cycles, very high CPU load

`XL_MOST150_STREAM_LATENCY_LOW`
`XL_MOST150_STREAM_LATENCY_MEDIUM`
`XL_MOST150_STREAM_LATENCY_HIGH`

`XL_MOST150_STREAM_LATENCY_VERY_HIGH`
Very high notification cycles, very low CPU load

11.5.9 XLmost150SyncAudioParameter

Syntax

```
typedef struct s_xl_most150_sync_audio_parameter {
    unsigned int label;
    unsigned int width;
    unsigned int device;
    unsigned int mode;
} XLmost150SyncAudioParameter;
```

Parameters**> label**

Connection Label used for routing data to line or S/PDIF out. In case of de-allocating bandwidth only, this parameter specifies the respective CL. For de-allocating each previously allocated CLs, the special CL value `XL_MOST150_CL_DEALLOC_ALL` (0xFFFF) can be used. This parameter is ignored in case of line or S/PDIF in routing.

> width

Number channels to be routed in case of line or S/PDIF in routing. Valid values are for line in 4 and for S/PDIF In 4 (currently only audio data is routed!). In case of allocating bandwidth only, this value specifies the bandwidth to be allocated. This parameter is ignored in case of line or S/PDIF out routing.

> device

`XL_MOST150_DEVICE_LINE_IN`
`XL_MOST150_DEVICE_LINE_OUT`
`XL_MOST150_DEVICE_SPDIF_IN`
`XL_MOST150_DEVICE_SPDIF_OUT`
`XL_MOST150_DEVICE_ALLOC_BANDWIDTH`

> mode

XL_MOST150_DEVICE_MODE_OFF
XL_MOST150_DEVICE_MODE_ON

11.6 Events

11.6.1 XLmost150event

Syntax

```

struct s_xl_event_most150 {
    unsigned int size;
    XLmostEventTag tag;
    unsigned short channelIndex;
    unsigned int userHandle;
    unsigned short flagsChip;
    unsigned short reserved;
    XLuint64      timeStamp;
    XLuint64      timeStampSync;

    union {
        unsigned char rawData[XL_MOST150_MAX_EVENT_DATA_SIZE];
        XL_MOST150_EVENT_SOURCE_EV          mostEventSource;
        XL_MOST150_DEVICE_MODE_EV         mostDeviceMode;
        XL_MOST150_SPDIF_MODE_EV         mostSpdifMode;
        XL_MOST150_FREQUENCY_EV          mostFrequency;
        XL_MOST150_SPECIAL_NODE_INFO_EV   mostSpecialNodeInfo;
        XL_MOST150_CTRL_SPY_EV           mostCtrlSpy;
        XL_MOST150_CTRL_RX_EV            mostCtrlRx;
        XL_MOST150_CTRL_TX_ACK_EV       mostCtrlTxAck;
        XL_MOST150_ASYNC_SPY_EV          mostAsyncSpy;
        XL_MOST150_ASYNC_RX_EV           mostAsyncRx;
        XL_MOST150_ASYNC_TX_ACK_EV      mostAsyncTxAck;
        XL_MOST150_SYNC_ALLOC_INFO_EV    mostSyncAllocInfo;
        XL_MOST150_TX_LIGHT_EV          mostTxLight;
        XL_MOST150_RXLIGHT_LOCKSTATUS_EV mostRxLightLockStatus;
        XL_MOST150_ERROR_EV             mostError;
        XL_MOST150_CTRL_SYNC_AUDIO_EV   mostCtrlSyncAudio;
        XL_MOST150_SYNC_VOLUME_STATUS_EV mostSyncVolumeStatus;
        XL_MOST150_SYNC_MUTE_STATUS_EV   mostSyncMuteStatus;
        XL_MOST150_LIGHT_POWER_EV       mostLightPower;
        XL_MOST150_GEN_LIGHT_ERROR_EV   mostGenLightError;
        XL_MOST150_GEN_LOCK_ERROR_EV    mostGenLockError;
        XL_MOST150_CONFIGURE_RX_BUFFER_EV mostConfigureRxBuffer;
        XL_MOST150_CTRL_BUSLOAD_EV      mostCtrlBusload;
        XL_MOST150_ASYNC_BUSLOAD_EV     mostAsyncBusload;
        XL_MOST150_ETHERNET_SPY_EV      mostEthernetSpy;
        XL_MOST150_ETHERNET_RX_EV       mostEthernetRx;
        XL_MOST150_ETHERNET_TX_ACK_EV   mostEthernetTxAck;
        XL_MOST150_SYSTEMLOCK_FLAG_EV   mostSystemLockFlag;
        XL_MOST150_SHUTDOWN_FLAG_EV     mostShutdownFlag;
        XL_MOST150_NW_STARTUP_EV        mostStartup;
        XL_MOST150_NW_SHUTDOWN_EV       mostShutdown;
        XL_MOST150_ECL_EV               mostEclEvent;
        XL_MOST150_ECL_TERMINATION_EV   mostEclTermination;
        XL_MOST150_ECL_SEQUENCE_EV      mostEclSequence;
        XL_MOST150_ECL_GLITCH_FILTER_EV mostEclGlitchFilter;
        XL_MOST150_HW_SYNC_EV           mostHWSync;
        XL_MOST150_STREAM_STATE_EV      mostStreamState;
        XL_MOST150_STREAM_TX_BUFFER_EV  mostStreamTxBuffer;
        XL_MOST150_STREAM_TX_LABEL_EV   mostStreamTxLabel;
        XL_MOST150_STREAM_TX_UNDERFLOW_EV mostStreamTxUnderflow;
        XL_MOST150_STREAM_RX_BUFFER_EV  mostStreamRxBuffer;
        XL_MOST150_GEN_BYPASS_STRESS_EV mostGenBypassStress;
        XL_MOST150_SSO_RESULT_EV        mostSsoResult;
    } tagData;
} XLmost150event;

```

Parameters

- > **size**
Overall size of the event (in bytes).
 - > **tag**
Specifies the event (see following sections).
 - > **channelIndex**
Channel of the received event.
 - > **userHandle**
Enables the assignment of requests and results, e. g. while sending messages or read/write of registers.
 - > **flagsChip**
XL_MOST150_VN2640 (common VN2640 event)
XL_MOST150_INIC (event was generated by INIC)
XL_MOST150_SPY (event was generated by spy)
- The upper 8 bits specifies the flags:
XL_MOST150_QUEUE_OVERFLOW
- > **reserved**
For future use.
 - > **timeStamp**
64 bit hardware time stamp with 1 ns resolution and 8 µs granularity.
 - > **timestamp_sync**
64 bit driver synchronized time stamp with 1 ns resolution and 8 µs granularity.
 - > **tagData**
Event data, depending on the tag and size.

11.6.2 XLmost150AsyncBusloadConfig

Syntax

```
typedef struct s_xl_most150_async_busload_config {
    unsigned int busloadType;
    unsigned int transmissionRate;
    unsigned int counterType;
    unsigned int counterPosition;

    union {
        unsigned char          rawBusloadPkt[1540];
        XLmost150AsyncTxMsg   busloadAsyncPkt;
        XLmost150EthernetTxMsg busloadEthernetPkt;
    } busloadPkt;
} XLmost150AsyncBusloadConfig;
```

Parameters

- > **busloadType**
Specifies whether MOST Data packets (MDP) or MOST Ethernet packets (MEP) should be transmitted.

Values:

XL_MOST150_BUSLOAD_TYPE_DATA_PACKET
XL_MOST150_BUSLOAD_TYPE_ETHERNET_PACKET

> **transmissonRate**

Number of packets per second to be transmitted.

Counter type values:

- XL_MOST150_BUSLOAD_COUNTER_TYPE_NONE
- XL_MOST150_BUSLOAD_COUNTER_TYPE_1_BYTE
- XL_MOST150_BUSLOAD_COUNTER_TYPE_2_BYTE
- XL_MOST150_BUSLOAD_COUNTER_TYPE_3_BYTE
- XL_MOST150_BUSLOAD_COUNTER_TYPE_4_BYTE

> **counterPosition**

Position in the payload of the MDP (0..1523) / MEP (0..1505).

Note: The counter position depends on the `countertype`:

Counter Type	Counter Position	
	MDP	MEP
1 Byte	0..1523	0..1505
2 Byte	1..1523	1..1505
3 Byte	2..1523	2..1505
4 Byte	3..1523	3..1505

> **busloadAsyncPkt**

See section [XLmost150AsyncTxMsg](#) on page 274.

> **busloadEthernetPkt**

See section [XLmost150EthernetTxMsg](#) on page 275.

11.6.3 XLmost150AsyncTxMsg

Syntax

```
typedef struct s_xl_most150_async_tx_msg {
    unsigned int priority;
    unsigned int asyncSendAttempts;
    unsigned int length;
    unsigned int targetAddress;
    unsigned char asyncData[XL_MOST150_ASYNC_SEND_PAYLOAD_MAX_SIZE];
} XLmost150AsyncTxMsg;
```

Parameters

> **priority**

Transmission priority. Bit 0..3 can be set for priority. However, the INIC currently only accepts the default value of 0x00.

> **asyncSendAttempts**

Transmission send attempts. Value range: 0x01..0x10 (0...15 retries). For using the default send attempt value this parameter has to be set to 0xFF. The default value is set with `xLMost150SetSpecialNodeInfo()` function.

> **length**

Number of bytes.

Note: It is possible to send a data packet with more than 1524 bytes. This can be used for testing purpose. However, the return event `XL_MOST150_ASYNC_TX_ACK` will report a maximum of 1524 byte.

> **targetAddress**

Logical target address of the data packet.

- > **asyncData**
Payload data (depending on length).

11.6.4 XLmost150EthernetTxMsg

Syntax

```
typedef struct s_xl_most150_ethernet_tx_msg {
    unsigned int priority;
    unsigned int ethSendAttempts;
    unsigned char sourceAddress[6];
    unsigned char targetAddress[6];
    unsigned int length;
    unsigned char ethernetData[XL_MOST150_ETHERNET_
        SEND_PAYLOAD_MAX_SIZE];
} XLmost150EthernetTxMsg;
```

Parameters

- > **priority**
Priority of the Ethernet packet. Can be 0x0 (for lowest priority) to 0x3 (for highest priority). Currently the INIC only accepts the default value of 0x00.
- > **ethSendAttempts**
Transmission send attempts. Value range: 0x01..0x10 (0...15 retries). For using the default send attempt value this parameter has to be set to 0xFF. The default value is set with `xIMost150SetSpecialNodeInfo()` function.
- > **sourceAddress**
Source MAC address of the Ethernet packet.
- > **targetAddress**
Target MAC address of the Ethernet packet.
- > **length**
Number of data bytes of the Ethernet packet.

Note: It is possible to send an Ethernet packet with more than 1506 payload bytes. This can be used for testing purpose. However, the return event `XL_MOST150_ETHERNET_TX_ACK` will report a maximum of 1506 byte.

- > **ethernetData**
Payload of the Ethernet packet (depends on length).

11.6.5 XL_START

Description

This event is returned after an `xIActivateChannel()` function call and contains data of time stamp counter at measuring start without event data.

Tag

`XL_START`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.6 XL_STOP

Description

This event is returned after an `xIDeactivateChannel()` function call, without event data

Tag

`XL_STOP`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.7 XL_MOST150_EVENT_SOURCE_EV

Syntax

```
typedef struct s_xl_most150_event_source{
    unsigned int sourceMask;
} XL_MOST150_EVENT_SOURCE_EV;
```

Description

This event is returned after `xIMost150SwitchEventSources()`.

Parameters

> **sourceMask**

- XL_MOST150_SOURCE_SPECIAL_NODE
- XL_MOST150_SOURCE_SYNC_ALLOC_INFO
- XL_MOST150_SOURCE_CTRL_SPY
- XL_MOST150_SOURCE_ASYNC_SPY
- XL_MOST150_SOURCE_ETH_SPY
- XL_MOST150_SOURCE_SHUTDOWN_FLAG
- XL_MOST150_SOURCE_SYSTEMLOCK_FLAG
- XL_MOST150_SOURCE_LIGHT_LOCK_SPY
- XL_MOST150_SOURCE_LIGHT_LOCK_INIC
- XL_MOST150_SOURCE_ECL_CHANGE
- XL_MOST150_SOURCE_LIGHT_STRESS
- XL_MOST150_SOURCE_LOCK_STRESS
- XL_MOST150_SOURCE_BUSLOAD_CTRL
- XL_MOST150_SOURCE_BUSLOAD_ASYNC
- XL_MOST150_SOURCE_CTRL_MLB
- XL_MOST150_SOURCE_ASYNC_MLB
- XL_MOST150_SOURCE_ETH_MLB
- XL_MOST150_SOURCE_TXACK_MLB
- XL_MOST150_SOURCE_STREAM_UNDERFLOW
- XL_MOST150_SOURCE_STREAM_OVERFLOW
- XL_MOST150_SOURCE_STREAM_RX_DATA
- XL_MOST150_SOURCE_ECL_SEQUENCE

Tag

`XL_MOST150_EVENT_SOURCE`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.8 XL_MOST150_DEVICE_MODE_EV

Syntax

```
typedef struct s_xl_most150_device_mode {
    unsigned int deviceMode;
} XL_MOST150_DEVICE_MODE_EV;
```

Description

Reports state of timing mode (master/slave/bypass, see `xIMost150SetDeviceMode()`, `xIMost150GetDeviceMode()`).

Parameters

> **deviceMode**

- XL_MOST150_DEVICEMODE_SLAVE
- XL_MOST150_DEVICEMODE_MASTER
- XL_MOST150_DEVICEMODE_STATIC_MASTER
- XL_MOST150_DEVICEMODE_RETIMED_BYPASS_SLAVE
- XL_MOST150_DEVICEMODE_RETIMED_BYPASS_MASTER

Tag

`XL_MOST150_DEVICE_MODE`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.9 XL_MOST150_SPDIF_MODE_EV

Syntax

```
typedef struct s_xl_most150_spdif_mode {
    unsigned int spdifMode;
    unsigned int spdifError;
} XL_MOST150_SPDIF_MODE_EV;
```

Description

Reports state of S/PDIF mode (master/slave, see `xIMost150SetSPDIFMode()`, `xIMost150GetSPDIFMode()`).

Parameters

- > **spdifMode**
`XL_MOST150_SPDIF_MODE_MASTER`
`XL_MOST150_SPDIF_MODE_SLAVE`
- > **spdifError**
Status of changed / requested S/PDIF mode.
`XL_MOST150_SPDIF_ERR_NO_ERROR`
`XL_MOST150_SPDIF_ERR_HW_COMMUNICATION`

Tag

`XL_MOST150_SPDIFMODE`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.10 XL_MOST150_FREQUENCY_EV

Syntax

```
typedef struct s_xl_most150_frequency {
    unsigned int frequency;
} XL_MOST150_FREQUENCY_EV;
```

Description

Reports frame rate of the MOST network.

Parameters

- > **frequency**
`XL_MOST150_FREQUENCY_44100`
`XL_MOST150_FREQUENCY_48000`
`XL_MOST150_FREQUENCY_ERROR`

Tag

`XL_MOST150_FREQUENCY`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.11 XL_MOST150_SPECIAL_NODE_INFO_EV

Syntax

```
typedef struct s_xl_most150_special_node_info{
    unsigned int changeMask;
    unsigned short nodeAddress;
    unsigned short groupAddress;
    unsigned char npr;
    unsigned char mpr;
    unsigned char sbc;
    unsigned char ctrlRetryTime;
    unsigned char ctrlSendAttempts;
    unsigned char asyncRetryTime;
    unsigned char asyncSendAttempts;
    unsigned char macAddr[6];
    unsigned char nprSpy;
    unsigned char mprSpy;
    unsigned char sbcSpy;
    unsigned char inicNIState;
} XL_MOST150_SPECIAL_NODE_INFO_EV;
```

Description This event reports spontaneously changes of specific node or spy info values. It may also be generated in case the value(s) are explicitly requested.

Parameters

> **changeMask**

Mask for the changes.

XL_MOST150_NA_CHANGED
XL_MOST150_GA_CHANGED
XL_MOST150_NPR_CHANGED
XL_MOST150_MPR_CHANGED
XL_MOST150_SBC_CHANGED
XL_MOST150_CTRL_RETRY_PARAMS_CHANGED
XL_MOST150_ASYNC_RETRY_PARAMS_CHANGED
XL_MOST150_MAC_ADDR_CHANGED
XL_MOST150_NPR_SPY_CHANGED
XL_MOST150_MPR_SPY_CHANGED
XL_MOST150_SBC_SPY_CHANGED
XL_MOST150_INIC_NI_STATE_CHANGED

> **nodeAddress**

Node address.

> **groupAddress**

Group address.

> **npr**

Node position detected by INIC.

> **mpr**

Number of nodes in the ring detected by INIC.

> **sbc**

Synchronous bandwidth control detected by INIC.

> **ctrlRetryTime**

Transmit retry time for control messages.

> **ctrlSendAttempts**

Default number of send attempts for control messages.

> **asyncRetryTime**

Transmit retry time for packets (MDP and MEP).

> **asyncSendAttempts**

Default number of send attempts for packets (MDP and MEP). Used if not set when sending a MDP or MEP.

> **nprSpy**

Node position detected from spy.

> **mprSpy**

Number of nodes in the ring detected by spy.

> **sbcSpy**

Synchronous bandwidth control detected by spy.

> **inicNIState**

Current state of INIC's NetInterface

```
XL_MOST150_INIC_NISTATE_NET_OFF
XL_MOST150_INIC_NISTATE_NET_INIT
XL_MOST150_INIC_NISTATE_NET_RBD
XL_MOST150_INIC_NISTATE_NET_ON
XL_MOST150_INIC_NISTATE_NET_RBD_RESULT
```

Tag

Syntax `XL_MOST150_SPECIAL_NODE_INFO`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.12 XL_MOST150_CTRL_SPY_EV

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_ctrl_spy{
    unsigned int    frameCount;
    unsigned int    msgDuration;
    unsigned char   priority;
    unsigned short  targetAddress;
    unsigned char   pAck;
    unsigned short  ctrlDataLenAnnounced;
    unsigned char   reserved0;
    unsigned char   pIndex;
    unsigned short  sourceAddress;
    unsigned short  reserved1;
    unsigned short  crc;
    unsigned short  crcCalculated;
    unsigned char   cAck;
    unsigned short  ctrlDataLen; ]
    unsigned char   reserved2;
    unsigned int    status;
    unsigned int    validMask;
    unsigned char   ctrlData[51];
} XL_MOST150_CTRL_SPY_EV;
```

Description

Reports a received control message from the spy.

Parameters> **frameCounter**

Current frame number.

> **msgDuration**

Duration of control message transmission in [ns].

> **priority**

Priority of the control message.

> **targetAddress**

Received target address.

> **pAck**

Pre-emptive acknowledge code of the control message:

```
XL_MOST150_PACK_OK
XL_MOST150_PACK_BUFFER_FULL
XL_MOST150_PACK_NO_RESPONSE
```

- > **ctrlDataLenAnnounced**
Number of data bytes announced by sender.
- > **pIndex**
Packet index of the control message.
- > **sourceAddress**
Received source address.
- > **crc**
CRC of the control message.
- > **crcCalculated**
FPGA calculated CRC (currently not filled).
- > **cAck**
CRC acknowledge code of the control message:
XL_MOST150_CACK_OK
XL_MOST150_CACK_CRC_ERROR
XL_MOST150_CACK_NO_RESPONSE
- > **ctrlDataLen**
Number of data bytes contained in `ctrlData []`.
- > **status**
Currently not used.
- > **validMask**
Mask signalizing which field is valid from this message event:
XL_MOST150_VALID_DATALENANNOUNCED
XL_MOST150_VALID_SOURCEADDRESS
XL_MOST150_VALID_TARGETADDRESS
XL_MOST150_VALID_PACK
XL_MOST150_VALID_CACK
XL_MOST150_VALID_PINDEX
XL_MOST150_VALID_PRIORITY
XL_MOST150_VALID_CRC
XL_MOST150_VALID_CRCCALCULATED
XL_MOST150_VALID_MESSAGE

Note: A set `XL_MOST150_VALID_MESSAGE` bit means a complete message transmission and that all fields are valid. Otherwise this is a “pre-terminated” message transmission and the validMask bits show which field is valid.

> **ctrlData**

Data of the control message (number of valid bytes: ctrlDataLen). The structure is as follows:

FBlockId: 8 bit
 InstId: 8 bit
 FunctionId: 12 bit
 OpType: 4 bit
 TelId: 4 bit
 TelLen: 12 bit
 Payload: 0..45 byte

```
ctrlData[0]: FBlockID
ctrlData[1]: InstID
ctrlData[2]: FunctionID (upper 8 bits)
ctrlData[3]: FunctionID (lower 4 bits) + OpType (4 bits)
ctrlData[4]: TelId (4 bits) + TelLen (upper 4 bits)
ctrlData[5]: TelLen (lower 8 bits)
ctrlData[6..50]: Payload
```

Tag

XL_MOST150_CTRL_SPY

See `s_xl_event_most150.tag` in section XLmost150event on page 272.**11.6.13 XL_MOST150_CTRL_RX_EV****Syntax**

```
typedef struct s_xl_most150_ctrl_rx {
    unsigned short targetAddress;
    unsigned short sourceAddress;
    unsigned char fblockId;
    unsigned char instId;
    unsigned short functionId;
    unsigned char opType;
    unsigned char telId;
    unsigned short telLen;
    unsigned char ctrlData[45];
} XL_MOST150_CTRL_RX_EV;
```

Description

This event reports a received control message from the node (INIC).

Parameters> **targetAddress**

Own address on receiving.

> **sourceAddress**

Unused for transmit.

> **fblockId**

Function block ID of the control message.

> **instId**

Instance ID of the control message.

> **functionId**

Function ID of the control message.

> **opType**

OpType of the control message.

> **telId**

Telegram ID of the control message.

- > **telLen**
Telegram length of the control message.
- > **ctrlData**
Payload (number of valid bytes: 0..45).

Tag

XL_MOST150_CTRL_RX

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).**11.6.14 XL_MOST150_CTRL_TX_ACK_EV****Syntax**

```
typedef struct s_xl_most150_ctrl_tx_ack {
    unsigned short targetAddress;
    unsigned short sourceAddress;
    unsigned char ctrlPrio;
    unsigned char ctrlSendAttempts;
    unsigned char reserved[2];
    unsigned int status;
    unsigned char ctrlData[51];
} XL_MOST150_CTRL_TX_ACK_EV;
```

Description

This event reports a transmit acknowledge of a control message. Refer to [xIMost150CtrlTransmit\(\)](#).

Parameters

- > **targetAddress**
Destination address of the control message.
- > **sourceAddress**
Own logical node address.
- > **ctrlPrio**
Transmission priority. Bit 0..3 can be set for priority. However, the INIC currently only accepts the default value of 0x01.
- > **ctrlSendAttempts**
Transmission send attempts. Value range: 0x01..0x10 (0.. 15 retries). For using the default send attempt value this parameter has to be set to 0xFF. The default value is set with [xIMost150SetSpecialNodeInfo\(\)](#) function.
- > **Status**

Transmit Status Register (see INIC User Manual, “FIFO Status Messages”):

```
XL_MOST150_TX_OK
XL_MOST150_TX_FAILED_FORMAT_ERROR
XL_MOST150_TX_FAILED_NETWORK_OFF
XL_MOST150_TX_FAILED_TIMEOUT
XL_MOST150_TX_FAILED_WRONG_TARGET
XL_MOST150_TX_OK_ONE_SUCCESS
XL_MOST150_TX_FAILED_BAD_CRC
XL_MOST150_TX_FAILED_RECEIVER_BUFFER_FULL
```

> **ctrlData**

Control data (number of valid bytes: 6..51). The structure is as follows:

FBlockId: 8 bit
 InstId: 8 bit
 FunctionId: 12 bit
 OpType: 4 bit
 TelId: 4 bit
 TelLen: 12 bit
 Payload: 0..45 byte

```
ctrlData[0]: FBlockID
ctrlData[1]: InstID
ctrlData[2]: FunctionID (upper 8 bits)
ctrlData[3]: FunctionID (lower 4 bits) + OpType (4 bits)
ctrlData[4]: TelId (4 bits) + TelLen (upper 4 bits)
ctrlData[5]: TelLen (lower 8 bits)
ctrlData[6..50]: Payload
```

Tag

`XL_MOST150_CTRL_TX_ACK`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.15 XL_MOST150_ASYNC_SPY_EV

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_async_spy_msg {
    unsigned int frameCount;
    unsigned int pktDuration;
    unsigned short asyncDataLenAnnounced;
    unsigned short targetAddress;
    unsigned char pAck;
    unsigned char pIndex;
    unsigned short sourceAddress;
    unsigned int crc;
    unsigned int crcCalculated;
    unsigned char cAck;
    unsigned short asyncDataLen;
    unsigned char reserved;
    unsigned int status;
    unsigned int validMask;
    unsigned char asyncData[1524];
} XL_MOST150_ASYNC_SPY_EV;
```

Description

The event reports a spy data packet (MDP).

Parameters> **frameCounter**

Current frame number.

> **pktDuration**

Duration of the data packet transmission in [ns].

> **priority**

Priority of the data packet.

> **targetAddress**

Received target address.

> **pAck**

Pre-emptive acknowledge code of the data packet:

XL_MOST150_PACK_OK
 XL_MOST150_PACK_BUFFER_FULL
 XL_MOST150_PACK_NO_RESPONSE

> **asyncDataLenAnnounced**

Number of data bytes announced by sender.

> **pIndex**

Packet index of packet.

> **sourceAddress**

Received source address.

> **crc**

CRC of the control message.

> **crcCalculated**

FPGA calculated CRC (currently not filled).

> **cAck**

CRC aacknowledge code of the data packet:

XL_MOST150_CACK_OK
 XL_MOST150_CACK_CRC_ERROR
 XL_MOST150_CACK_NO_RESPONSE

> **asyncDataLen**

Number of data bytes contained in `asyncData`.

> **status**

Currently not used.

> **validMask**

Mask signalizing which field is valid from this data packet event:

XL_MOST150_VALID_DATALENANNOUNCED
 XL_MOST150_VALID_SOURCEADDRESS
 XL_MOST150_VALID_TARGETADDRESS
 XL_MOST150_VALID_PACK
 XL_MOST150_VALID_CACK
 XL_MOST150_VALID_PINDEX
 XL_MOST150_VALID_PRIORITY
 XL_MOST150_VALID_CRC
 XL_MOST150_VALID_CRCCALCULATED
 XL_MOST150_VALID_MESSAGE

Note: In case `XL_MOST150_VALID_MESSAGE` bit is set, this a complete data packet transmission and all fields are valid. Otherwise this is a “pre-terminated” data packet transmission and the `validMask` bits show which field is valid.

Additionally it is possible to send a data packet with more than 1524 bytes. Upon detection of such a “too long” data packet, the flag `XL_MOST150_VALID_MESSAGE` will not be set. The `asyncDataLen` parameter will show the maximum value of 1524 but the `asyncDataLenAnnounced` parameter will show the actual length value.

> **asyncData**

Payload (depending on `asyncDataLen`).

Tag	XL_MOST150_ASYNC_SPY See <code>s_xl_event_most150.tag</code> in section XLmost150event on page 272.
------------	--

11.6.16 XL_MOST150_ASYNC_RX_EV

Syntax

```
typedef struct s_xl_most150_async_msg {
    unsigned short length;
    unsigned short targetAddress;
    unsigned short sourceAddress;
    unsigned char  asyncData[1524];
} XL_MOST150_ASYNC_RX_EV;
```

Description

The event reports a received data packet (MDP) from the node (INIC).

Parameters

> **length**

Number of bytes.

Note: It is possible to send a data packet with more than 1524 bytes. Upon reception of such a “too long” data packet, the flag `XL_MOST150_ASYNC_INVALID_RX_LENGTH` will be set in the length parameter.

> **targetAddress**

Logical target address of the data packet.

> **sourceAddress**

Logical source address of the data packet.

> **asyncData**

Payload (depending on length).

Tag

XL_MOST150_ASYNC_RX

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.17 XL_MOST150_ASYNC_TX_ACK_EV

Syntax

```
typedef struct s_xl_most150_async_tx_ack{
    unsigned char priority;
    unsigned char asyncSendAttempts;
    unsigned short length;
    unsigned short targetAddress;
    unsigned short sourceAddress;
    unsigned int   status;
    unsigned char  asyncData[1524];
} XL_MOST150_ASYNC_TX_ACK_EV;
```

Description

The event reports a transmit acknowledge of a data packet (MDP). Refer to [xIMost150AsyncTransmit\(\)](#).

Parameters

> **priority**

Transmission priority. Bit 0..3 can be set for priority. However, the INIC currently only accepts the default value of 0x00.

> **asyncSendAttempts**

Transmission send attempts. Value range: 0x01..0x10 (0..15 retries). For using the default send attempt value, this parameter has to be set to 0xFF. The default value is set with [xIMost150SetSpecialNodeInfo\(\)](#) function.

> **length**

Number of bytes.

Note: It is possible to send a data packet with more than 1524 bytes. This can be used for testing purpose. However, this event will report a maximum of 1524 byte.

> **targetAddress**

Logical target address of the data packet.

> **sourceAddress**

Logical source address of the data packet.

> **status**

Transmit result (currently not used since INIC does not report a transmit result).

> **asyncData**

Payload data (depending on length).

Tag

`XL_MOST150_ASYNC_TX_ACK`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.18 XL_MOST150_CL_INFO**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
#define MOST150_SYNC_ALLOC_INFO_SIZE (unsigned int) 372
typedef struct s_xl_most150_cl_info {
    unsigned short label;
    unsigned short channelWidth;
} XL_MOST150_CL_INFO;
```

Description

The event is generated when changes within the synchronous area of the allocation table occur or the application requested the information by calling `xIMost150SyncGetAllocTable()`.

Parameters> **label**

Connection Label.

> **channelWidth**

Number of bytes which belong to Connection Label.

`channelWidth > 0`: Channels have been allocated

`channelWidth = 0`: Channels have been de-allocated

Tag

`XL_MOST150_SYNC_ALLOC_INFO`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.19 XL_MOST150_SYNC_ALLOC_INFO_EV**Syntax**

```
typedef struct s_xl_most150_sync_alloc_info {
    XL_MOST150_CL_INFO allocTable[MOST150_SYNC_ALLOC_INFO_SIZE];
} XL_MOST150_SYNC_ALLOC_INFO_EV;
```

Parameters> **allocTable**

section [XL_MOST150_CL_INFO](#) on page 286

11.6.20 XL_MOST150_TX_LIGHT_EV


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_tx_light {
    unsigned int light;
} XL_MOST150_TX_LIGHT_EV;
```

Description

The event reports changes on the FOT or answers to `xIMost150SetTxLight()` and `xIMost150GetTxLight()` requests.

Parameters
> light

- `XL_MOST150_LIGHT_OFF`
- `XL_MOST150_LIGHT_FORCE_ON` (currently not supported!)
- `XL_MOST150_LIGHT_MODULATED`

Tag

`XL_MOST150_TX_LIGHT`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.21 XL_MOST150_RXLIGHT_LOCKSTATUS_EV

Syntax

```
typedef struct s_xl_most150_rx_light_lock_status {
    unsigned int status;
} XL_MOST150_RXLIGHT_LOCKSTATUS_EV;
```

Description

This event reports light&lock changes or reports an answer to `xIMostGetRxLightLockStatus()`. The `flagsChip` value determines whether the event is reported by the node (INIC) or spy.

Parameters
> status

- `XL_MOST150_LIGHT_OFF`
- `XL_MOST150_LIGHT_ON_UNLOCK`
- `XL_MOST150_LIGHT_ON_LOCK`
- `XL_MOST150_LIGHT_ON_STABLE_LOCK`
- `XL_MOST150_LIGHT_ON_CRITICAL_UNLOCK`

Tag

`XL_MOST150_RXLIGHT_LOCKSTATUS`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.22 XL_MOST150_ERROR_EV

Syntax

```
typedef struct s_xl_most150_error {
    unsigned int errorCode;
    unsigned int parameter[3];
} XL_MOST150_ERROR_EV;
```

Description

This event reports an error.

Parameters

- > **errorCode**
 XL_MOST150_ERROR_ASYNC_TX_ACK_HANDLE
 Invalid Tx Data Packet handle received.

XL_MOST150_ERROR_ETH_TX_ACK_HANDLE
 Invalid Tx Ethernet Packet handle received.

- > **parameter**
 Reserved for future use.

Tag
 XL_MOST150_ERROR
 See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.23 XL_MOST150_CTRL_SYNC_AUDIO_EV

Syntax

```
typedef struct s_xl_most150_ctrl_sync_audio {
    unsigned int label;
    unsigned int width;
    unsigned int device;
    unsigned int mode;
} XL_MOST150_CTRL_SYNC_AUDIO_EV;
```

Description

The event is the response for the `xIMost150CtrlSyncAudio()` function. The content is the same like within the command.

Parameters

- > **label**
 Connection label used for routing data to line or S/PDIF out or bandwidth allocation and respectively de-allocation. This parameter can be ignored in case if line or S/PDIF in routing.
- > **width**
 Number channels to be routed in case of line or S/PDIF in routing or used for allocating bandwidth. This parameter can be ignored in case if line or S/PDIF out routing.
- > **device**
 Describes the device address:

XL_MOST150_DEVICE_LINE_IN
 XL_MOST150_DEVICE_LINE_OUT
 XL_MOST150_DEVICE_SPDIF_IN
 XL_MOST150_DEVICE_SPDIF_OUT
 XL_MOST150_DEVICE_ALLOC_BANDWIDTH

> mode

XL_MOST150_DEVICE_MODE_ON
 XL_MOST150_DEVICE_MODE_OFF

Additionally there are the following values in case an error occurred:

XL_MOST150_DEVICE_MODE_OFF_BYPASS_CLOSED

Bypass is closed. If bypass is closed neither data can be routed nor is allocating of any bandwidth possible. Any active routings are deactivated and allocated bandwidth is freed automatically.

XL_MOST150_DEVICE_MODE_OFF_NOT_IN_NETON

NetInterface is not in state NetOn. Routing is not possible respectively bandwidth cannot be allocated.

XL_MOST150_DEVICE_MODE_OFF_NO_MORE_RESOURCES

The maximum number of allocated CLs (10) is already reached.

XL_MOST150_DEVICE_MODE_OFF_NOT_ENOUGH_FREE_BW

There is not enough free bandwidth available. Line or S/PDIF in routing is not activated respectively bandwidth is not allocated.

XL_MOST150_DEVICE_MODE_OFF_DUE_TO_NET_OFF

NetInterface is in state NetOff. Neither data routing nor allocating of any bandwidth possible. Any active routings are deactivated and allocated bandwidth is freed automatically.

XL_MOST150_DEVICE_MODE_OFF_DUE_TO_CFG_NOT_OK

The Network Configuration state switched to 'NotOk'. Any active routings are deactivated and allocated bandwidth is freed automatically.

XL_MOST150_DEVICE_MODE_OFF_COMMUNICATION_ERROR

A communication error with INIC occurred. This may happen if e. g. line or S/PDIF out should be activated for a non-existing CL.

XL_MOST150_DEVICE_MODE_OFF_STREAM_CONN_ERROR

A Stream Socket Connection Error occurred. This may happen in case line or S/PDIF out routing is active and the respective CL is de-allocated. (refer also to INIC UM – "SCError").

XL_MOST150_DEVICE_MODE_OFF_CL_ALREADY_USED

The given CL is already used by line or S/PDIF out. This can only happen in case line or S/PDIF out routing should be activated on the same CL.

XL_MOST150_DEVICE_MODE_CL_NOT_ALLOCATED

The given CL which should be de-allocated was previously not allocated by the VN2640.

Tag

XL_MOST150_CTRL_SYNC_AUDIO

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.24 XL_MOST150_SYNC_VOLUME_STATUS_EV

Syntax

```
typedef struct s_xl_most150_sync_volume_status {
    unsigned int device;
```

```
    unsigned int volume;
} XL_MOST150_SYNC_VOLUME_STATUS_EV;
```

Description Reports the volume level for the line in and line out ports.

Parameters

> **device**

Describes the device address:

XL_MOST150_DEVICE_LINE_IN
XL_MOST150_DEVICE_LINE_OUT

> **volume**

Volume level from 0...255 (0...100%).

Tag

XL_MOST150_SYNC_VOLUME_STATUS

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.25 XL_MOST150_SYNC_MUTE_STATUS_EV

Syntax

```
typedef struct s_xl_most150_sync_mute_status {
    unsigned int device;
    unsigned int mute;
} XL_MOST150_SYNC_MUTE_STATUS_EV;
```

Description

Reports the mute status for the line / S/PDIF in and the line / S/PDIF out ports.

Parameters

> **device**

Describes the device address:

XL_MOST150_DEVICE_LINE_IN
XL_MOST150_DEVICE_LINE_OUT
XL_MOST150_DEVICE_SPDIF_IN
XL_MOST150_DEVICE_SPDIF_OUT

> **mute**

Mute status for the addressed device:

XL_MOST_NO_MUTE
XL_MOST_MUTE

Tag

XL_MOST150_SYNC_MUTE_STATUS

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.26 XL_MOST150_LIGHT_POWER_EV

Syntax

```
typedef struct s_xl_most150_tx_light_power {
    unsigned int lightPower;
} XL_MOST150_LIGHT_POWER_EV;
```

Description

Reports the light power on the FOT.

Parameters

> **lightPower**

Power status of the FOT:

XL_MOST150_LIGHT_FULL
XL_MOST150_LIGHT_3DB

Tag

XL_MOST150_LIGHT_POWER

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.27 XL_MOST150_GEN_LIGHT_ERROR_EV


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_gen_light_error {
    unsigned int stressStarted;
} XL_MOST150_GEN_LIGHT_ERROR_EV;
```

Description

This event signals the start and stop of the lightOn-lightOff stress mode (see [xIMost150GenerateLightError\(\)](#)).

Parameters

> **stressStarted**

XL_MOST150_MODE_DEACTIVATED
Stress stopped.

XL_MOST150_MODE_ACTIVATED
Stress started.

Tag

XL_MOST150_GEN_LIGHT_ERROR

See [s_xl_event_most150.tag](#) in section [XLmost150event](#) on page 272.

11.6.28 XL_MOST150_GEN_LOCK_ERROR_EV


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_gen_lock_error {
    unsigned int stressStarted;
} XL_MOST150_GEN_LOCK_ERROR_EV;
```

Description

This event signals the start and stop of the lock-unlock stress mode (see [xIMost150GenerateLockError\(\)](#)).

Parameters

> **stressStarted**

XL_MOST150_MODE_DEACTIVATED
Stress stopped.

XL_MOST150_MODE_ACTIVATED
Stress started.

Tag

XL_MOST150_GEN_LOCK_ERROR

See [s_xl_event_most150.tag](#) in section [XLmost150event](#) on page 272.

11.6.29 XL_MOST150_CONFIGURE_RX_BUFFER_EV


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_configure_rx_buffer {
```

```
unsigned int bufferType;
unsigned int bufferMode;
} XL_MOST150_CONFIGURE_RX_BUFFER_EV;
```

Description

This event signals the buffer mode of the receive buffer for control messages and packets.

Parameters**> bufferType**

Bitmask which specifies the receive buffer type
XL_MOST150_RX_BUFFER_TYPE_CTRL
Control message buffer.

XL_MOST150_RX_BUFFER_TYPE_ASYNC
Packet buffer (MDP and MEP).

> bufferMode

Block or unblock processing the respective receive buffer.
XL_MOST150_RX_BUFFER_NORMAL_MODE
Messages and/or packets are processed.

XL_MOST150_RX_BUFFER_BLOCK_MODE
Messages and/or packets are not processed.

Tag

XL_MOST150_CONFIGURE_RX_BUFFER

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.30 XL_MOST150_CTRL_BUSLOAD_EV

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_ctrl_busload {
    unsigned int busloadStarted;
} XL_MOST150_CTRL_BUSLOAD_EV;
```

Description

This is the response event for the `xIMost150CtrlGenerateBusload()` and shows the start/stop of the busload generation. The function `xIMost150CtrlConfigureBusload()` must be called first.

Parameters**> busloadStarted**

XL_MOST150_MODE_DEACTIVATED
Busload stopped.

XL_MOST150_MODE_ACTIVATED
Busload started.

Tag

XL_MOST150_CTRL_BUSLOAD

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.31 XL_MOST150_ASYNC_BUSLOAD_EV


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_async_busload {
    unsigned int busloadStarted;
} XL_MOST150_ASYNC_BUSLOAD_EV;
```

Description

This is the response event on a `xIMost150AsyncGenerateBusload()` function call and shows the start/stop of the busload generation. The function `xIMost150ASyncConfigureBusload()` must be called first.

Parameters
> busloadStarted

`XL_MOST150_MODE_DEACTIVATED`
Busload stopped.

`XL_MOST150_MODE_ACTIVATED`
Busload started.

Tag

`XL_MOST150_ASYNC_BUSLOAD`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.32 XL_MOST150_ETHERNET_SPY_EV


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_ethernet_spy {
    unsigned int frameCount;
    unsigned int pktDuration;
    unsigned short ethernetDataLenAnnounced;
    unsigned char targetAddress[6];
    unsigned char pAck;
    unsigned char sourceAddress[6];
    unsigned char reserved0;
    unsigned int crc;
    unsigned int crcCalculated;
    unsigned char cAck;
    unsigned short ethernetDataLen; // bytes in ethernetData[]
    unsigned char reserved1;
    unsigned int status; // currently not used
    unsigned int validMask;
    unsigned char ethernetData[1506];
} XL_MOST150_ETHERNET_SPY_EV;
```

Description

Shows a received Ethernet packet from the spy.

Parameters
> frameCounter

Current frame number.

> pktDuration

Duration of the Ethernet packet transmission in [ns].

> ethernetDataLenAnnounced

Number of data bytes announced by sender.

- > **targetAddress**
Target MAC address of the Ethernet packet.
- > **pAck**
Pre-emptive acknowledge code of the Ethernet packet:
XL_MOST150_PACK_OK
XL_MOST150_PACK_BUFFER_FULL
XL_MOST150_PACK_NO_RESPONSE
- > **sourceAddress**
Source MAC address of the Ethernet packet.
- > **crc**
CRC value of the Ethernet packet.
- > **crcCalculated**
FPGA calculated CRC (currently not filled).
- > **cAck**
CRC acknowledge code of the Ethernet packet:
XL_MOST150_CACK_OK
XL_MOST150_CACK_CRC_ERROR
XL_MOST150_CACK_NO_RESPONSE
- > **ethernetDataLen**
Number of data bytes contained in `ethernetData[]`.
- > **status**
Currently not used.
- > **validMask**
Mask signaling which field is valid from this Ethernet packet event:
XL_MOST150_VALID_DATALENANNOUNCED
XL_MOST150_VALID_SOURCEADDRESS
XL_MOST150_VALID_TARGETADDRESS
XL_MOST150_VALID_PACK
XL_MOST150_VALID_CACK
XL_MOST150_VALID_CRC
XL_MOST150_VALID_CRCCALCULATED
XL_MOST150_VALID_MESSAGE

Note: In case `XL_MOST150_VALID_MESSAGE` bit is set, this a complete Ethernet packet transmission and all fields are valid.

Otherwise this is a “pre-terminated” Ethernet packet transmission and the validMask bits show which field is valid.

Additionally it is possible to send a Ethernet packet with more than 1506 bytes. Upon detection of such a “too long” Ethernet packet, the flag `XL_MOST150_VALID_MESSAGE` will not be set. The `ethernetDataLen` parameter will show the maximum value of 1506 but the `ethernetDataLenAnnounced` parameter will show the actual length value.

- > **ethernetData**
Payload of the Ethernet packet (depends on `ethernetDataLen`).

Tag

`XL_MOST150_ETHERNET_SPY`

See `s_xl_event_most150.tag` in section [XLmost150Event](#) on page 272.

11.6.33 XL_MOST150_ETHERNET_RX_EV

Syntax

```
typedef struct s_xl_most150_ethernet_rx {
    unsigned char sourceAddress[6];
    unsigned char targetAddress[6];
    unsigned int length;
    unsigned char data[1510];
} XL_MOST150_ETHERNET_RX_EV;
```

Description This event reports the receiving of an Ethernet packet from the node (INIC).

Parameters

- > **sourceAddress**
Source MAC address of the Ethernet packet.
- > **targetAddress**
Target MAC address of the Ethernet packet.
- > **length**
Number of data bytes of the Ethernet packet.

Note: It is possible to send an Ethernet packet with more than 1506 bytes. Upon reception of such a “too long” Ethernet packet, the flag `XL_MOST150_ETHERNET_INVALID_RX_LENGTH` will be set in the `length` parameter.

- > **data**
Payload of the Ethernet packet (depends on length).

Tag

`XL_MOST150_ETHERNET_RX`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.34 XL_MOST150_ETHERNET_TX_ACK_EV

Syntax

```
typedef struct s_xl_most150_ethernet_tx {
    unsigned char priority;
    unsigned char ethSendAttempts;
    unsigned char sourceAddress[6];
    unsigned char targetAddress[6];
    unsigned char reserved[2];
    unsigned int length;
    unsigned char ethernetData[1510];
} XL_MOST150_ETHERNET_TX_ACK_EV;
```

Description This event reports a transmit acknowledge of an Ethernet packet. Refer to `xIMost150EthernetTransmit()`.

Parameters

- > **priority**
Priority of the Ethernet packet. Can be 0x0 (for lowest priority) to 0x3 (for highest priority). Currently the INIC only accepts the default value of 0x00.
- > **ethSendAttempts**
Transmission send attempts. Value range: 0x01..0x10 (0..15 retries). For using the default send attempt value this parameter has to be set to 0xFF. The default value is set with `xIMost150SetSpecialNodeInfo()`.
- > **sourceAddress**
Source MAC address of the Ethernet packet.
- > **targetAddress**
Target MAC address of the Ethernet packet.

> **length**

Number of data bytes of the Ethernet packet.

Note: It is possible to send an Ethernet packet with more than 1506 payload bytes. This can be used for testing purpose. However, this event will report a maximum of 1506 byte.

> **ethernetData**

Payload of the Ethernet packet (depends on length).

Tag

`XL_MOST150_ETHERNET_TX_ACK`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.35 XL_MOST150_SYSTEMLOCK_FLAG_EV**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_systemlock_flag {
    unsigned char state;
} XL_MOST150_SYSTEMLOCK_FLAG_EV;
```

Description

This event reports the state of SystemLock flag.

Parameters> **state**

```
XL_MOST150_SYSTEMLOCK_FLAG_SET
XL_MOST150_SYSTEMLOCK_FLAG_NOT_SET
```

Tag

`XL_MOST150_SYSTEMLOCK_FLAG`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.36 XL_MOST150_SHUTDOWN_FLAG_EV**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_shutdown_flag {
    unsigned char state;
} XL_MOST150_SHUTDOWN_FLAG_EV;
```

Description

This event reports the state of shutdown flag.

Parameters> **state**

```
XL_MOST150_SHUTDOWN_FLAG_SET
XL_MOST150_SHUTDOWN_FLAG_NOT_SET
```

Tag

`XL_MOST150_SHUTDOWN_FLAG`

See `s_xl_event_most150.tag` in section [XLmost150event on page 272](#).

11.6.37 XL_MOST150_NW_STARTUP_EV

Syntax

```
typedef struct s_xl_most150_nw_startup {
    unsigned int error;
    unsigned int errorInfo;
} XL_MOST150_NW_STARTUP_EV;
```

Description

Reports the result for a startup of the network (see `xIMost150Startup()`).

Parameters

> **error**

`XL_MOST150_STARTUP_NO_ERROR`

Otherwise the respective MOST ErrorCode from INIC is reported.

> **errorInfo**

`XL_MOST150_STARTUP_NO_ERRORINFO`

Otherwise the respective MOST ErrorInfo from INIC is reported.

Tag

`XL_MOST150_NW_STARTUP`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.38 XL_MOST150_NW_SHUTDOWN_EV

Syntax

```
typedef struct s_xl_most150_nw_shutdown {
    unsigned int error;
    unsigned int errorInfo;
} XL_MOST150_NW_SHUTDOWN_EV;
```

Description

Reports the result for a shutdown of the network (see `xIMost150Shutdown()`).

Parameters

> **error**

`XL_MOST150_SHUTDOWN_NO_ERROR`

Otherwise the respective MOST ErrorCode from INIC is reported.

> **errorInfo**

`XL_MOST150_SHUTDOWN_NO_ERRORINFO`

Otherwise the respective MOST ErrorInfo from INIC is reported.

Tag

`XL_MOST150_NW_SHUTDOWN`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.39 XL_MOST150_ECL_EV

Syntax

```
typedef struct s_xl_most150_ecl {
    unsigned int eclLineState;
} XL_MOST150_ECL_EV;
```

Description

Reports an ECL line signal change.

Parameters

> **eclLineState**

`XL_MOST150_ECL_LINE_LOW`

`XL_MOST150_ECL_LINE_HIGH`

Tag

`XL_MOST150_ECL_LINE_CHANGED`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.40 XL_MOST150_ECL_TERMINATION_EV

Syntax

```
typedef struct s_xl_most150_ecl_termination {
    unsigned int resistorEnabled;
} XL_MOST150_ECL_TERMINATION_EV;
```

Description

Reports a termination change of ECL.

Parameters

> **resistorEnabled**

XL_MOST150_ECL_LINE_PULL_UP_NOT_ACTIVE
XL_MOST150_ECL_LINE_PULL_UP_ACTIVE

Tag

XL_MOST150_ECL_TERMINATION_CHANGED

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.41 XL_MOST150_ECL_SEQUENCE_EV



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_ecl_sequence {
    unsigned int sequenceStarted;
} XL_MOST150_ECL_SEQUENCE_EV;
```

Description

This is the response event on an `xIMost150ECLGenerateSeq()` function call and shows the start/stop of the sequence generation. The function `xIMost150ECLConfigureSeq()` must be called first.

Parameters

> **sequenceStarted**

XL_MOST150_MODE_DEACTIVATED
Sequence stopped.

XL_MOST150_MODE_ACTIVATED
Sequence started.

Tag

XL_MOST150_ECL_SEQUENCE

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.42 XL_MOST150_ECL_GLITCH_FILTER_EV



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_ecl_glitch_filter {
    unsigned int duration;
} XL_MOST150_ECL_GLITCH_FILTER_EV;
```

Description

Reports the duration for the ECL glitch filter.

Parameters> **duration**

Duration (in μ s) of glitches to be filtered. Value range: 50 μ s .. 50 ms.

Default: 1 ms

Tag

XL_MOST150_ECL_GLITCH_FILTER

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.43 XL_MOST150_STREAM_STATE_EV

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_stream_state {  
    unsigned int streamHandle;  
    unsigned int streamState;  
    unsigned int streamError;  
} XL_MOST150_STREAM_STATE_EV;
```

Description

Reports the stream state of an Rx or Tx stream.

Parameters> **streamHandle**

Stream handle (returned by `xIMost150StreamOpen()`).

> **streamState**

Stream state:

XL_MOST150_STREAM_STATE_CLOSED
XL_MOST150_STREAM_STATE_OPENED
XL_MOST150_STREAM_STATE_STARTED
XL_MOST150_STREAM_STATE_STOPPED

> **streamError**

Reports additional error information:

`XL_MOST150_STREAM_STATE_ERROR_NO_ERROR`
No error occurred.

`XL_MOST150_STREAM_STATE_ERROR_NOT_ENOUGH_BW`
The desired bandwidth for a Tx stream cannot be allocated.

`XL_MOST150_STREAM_STATE_ERROR_NET_OFF`

NetInterface is in state NetOff. No streaming is possible. In case streaming was activated it is automatically stopped. Additionally a Tx stream is closed.

`XL_MOST150_STREAM_STATE_ERROR_CONFIG_NOT_OK`

The Network Configuration state switched to 'NotOk'. Any active streaming is stopped. Additionally a Tx stream is closed.

`XL_MOST150_STREAM_STATE_ERROR_CL_DISAPPEARED`

Every connection label from the Rx stream disappeared, thus streaming is automatically stopped.

`XL_MOST150_STREAM_STATE_ERROR_INIC_SC_ERROR`

INIC reported a socket connection error for the Tx stream. Streaming is automatically stopped and stream is closed.

`XL_MOST150_STREAM_STATE_ERROR_DEVICEMODE_BYPASS`

INIC's bypass was closed by application request. With closed bypass no streaming is possible, so streaming will be stopped automatically. Additionally a Tx stream is closed.

`XL_MOST150_STREAM_STATE_ERROR_NISTATE_NOT_NETON`

NetInterface is not in NetOn, thus no streaming is possible. This error might be reported when opening the Tx stream.

`XL_MOST150_STREAM_STATE_ERROR_INIC_BUSY`

INIC is currently busy processing other requests. The application may perform a retry.

`XL_MOST150_STREAM_STATE_ERROR_CL_MISSING`

One ore more connection labels are missing when trying to start the Rx stream.

`XL_MOST150_STREAM_STATE_ERROR_NUM_BYTES_MISMATCH`

The number of bytes per MOST frame given by the application does not match the number of bytes actually given by the connection labels for the Rx stream.

> `XL_MOST150_STREAM_STATE_ERROR_INIC_COMMUNICATION`

A communication error with INIC occurred.

Tag

`XL_MOST150_STREAM_STATE`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.44 XL_MOST150_STREAM_TX_BUFFER_EV

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_stream_tx_buffer {
    unsigned int streamHandle;
    unsigned int numberOfBytes;
    unsigned int status;
} XL_MOST150_STREAM_TX_BUFFER_EV;
```

Description

The event notifies the application that the fill level of the Tx FIFO has dropped below a given watermark and so further data is required for streaming in order to avoid a data underflow. The application should call `xIMost150StreamTransmitData()`.

Parameters**> streamHandle**

Stream handle (returned by `xIMost150StreamOpen()`).

> numberOfBytes

Number of bytes that can at least be written into the Tx FIFO (see `xIMost150StreamTransmitData()`).

> status

Status information:

`XL_MOST150_STREAM_BUFFER_ERROR_NO_ERROR`

No error occurred.

`XL_MOST150_STREAM_BUFFER_ERROR_NOT_ENOUGH_DATA`

This can happen in case the application started the Tx stream but did not yet provide any streaming data. "0" data is streamed until application provided data by calling `xIMost150StreamTransmitData()`.

Note: In this case the application should provide at least $2 \times \text{numberOfBytes}$ of data to avoid an immediate underflow.

Tag

`XL_MOST150_STREAM_TX_BUFFER`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.45 XL_MOST150_STREAM_TX_LABEL_EV

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_stream_tx_label {
    unsigned int streamHandle;
    unsigned int errorInfo;
    unsigned int connLabel;
    unsigned int width;
} XL_MOST150_STREAM_TX_LABEL_EV;
```

Description

Reports the connection label of the Tx stream.

Parameters**> streamHandle**

Stream handle (returned by `xIMost150StreamOpen()`).

> connLabel

Connection label of the Tx stream.

> **width**

Width of the connection label.

In case of `errorInfo = XL_MOST150_STREAM_STATE_ERROR_NO_ERROR:`
`width > 0: Connection label allocated`
`width = 0: Connection label de-allocated`

In case of an error, the connection label is de-allocated or could not be allocated at all.

> **errorInfo**

Error information:

`XL_MOST150_STREAM_STATE_ERROR_NO_ERROR`
`No error occurred.`

`XL_MOST150_STREAM_STATE_ERROR_NOT_ENOUGH_BW`
The desired bandwidth for a Tx stream cannot be allocated.

`XL_MOST150_STREAM_STATE_ERROR_NET_OFF`

NetInterface is in state NetOff. The allocated bandwidth is automatically freed and connection label is invalid.

`XL_MOST150_STREAM_STATE_ERROR_CONFIG_NOT_OK`

The Network Configuration state switched to 'NotOk'. The allocated bandwidth is automatically freed and connection label is invalid.

`XL_MOST150_STREAM_STATE_ERROR_INIC_SC_ERROR`

INIC reported a socket connection error for the Tx stream. The allocated bandwidth is automatically freed and connection label is invalid.

`XL_MOST150_STREAM_STATE_ERROR_DEVICEMODE_BYPASS`

INIC's bypass was closed by application request. The allocated bandwidth is automatically freed and connection label is invalid.

Tag

`XL_MOST150_STREAM_TX_LABEL`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.46 XL_MOST150_STREAM_TX_UNDERFLOW_EV

**Note**

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_stream_tx_underflow {
    unsigned int streamHandle;
    unsigned int reserved;
} XL_MOST150_STREAM_TX_UNDERFLOW_EV;
```

Description

This event is reported in case no data was available to send due to an empty transmit buffer.

Parameters> **streamHandle**

Stream handle (returned by `xIMost150StreamOpen()`).

Tag

`XL_MOST150_STREAM_TX_UNDERFLOW`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.47 XL_MOST150_STREAM_RX_BUFFER_EV


Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_stream_rx_buffer {
    unsigned int streamHandle;
    unsigned int numberofBytes;
    unsigned int status;
    unsigned int labelInfo;
} XL_MOST150_STREAM_RX_BUFFER_EV;
```

Description

The event reports the number of received streaming bytes available in the Rx FIFO. The application should call `xIMost150StreamReceiveData()` as soon as possible to avoid data overflows. The reported time stamp refers to the MOST frame of the last data bytes and can be used for synchronization purpose to other MOST events.

Parameters

> **streamHandle**

Stream handle (returned by `xIMost150StreamOpen()`).

> **numberOfBytes**

Number of bytes available in the Rx FIFO (see `xIMost150StreamReceiveData()`)

> **status**

Status information:

`XL_MOST150_STREAM_BUFFER_ERROR_NO_ERROR`

No error occurred, Rx stream active.

`XL_MOST150_STREAM_BUFFER_ERROR_STOP_BY_APP`
Rx streaming stopped by application.

`XL_MOST150_STREAM_BUFFER_ERROR_MOST_SIGNAL_OFF`
Rx streaming stopped since MOST signal was switched off.

`XL_MOST150_STREAM_BUFFER_ERROR_UNLOCK`

Rx streaming was stopped due to an unlock and now is continued since lock is regained. The status indicates a gap in streaming data between this buffer event and the preceding one.

`XL_MOST150_STREAM_BUFFER_ERROR_CL_MISSING`

One or more connection labels are missing, i.e. they have been de-allocated. Fill bytes are inserted for the respective connection label(s) to keep MOST frame alignment. Rx stream still active.

`XL_MOST150_STREAM_BUFFER_ERROR_ALL_CL_MISSING`

Rx streaming stopped since all connection labels have been de-allocated.

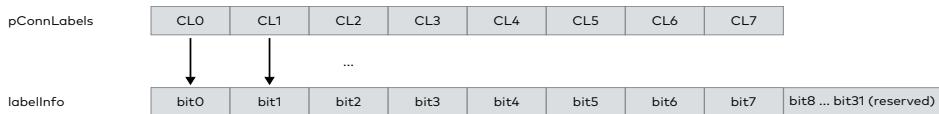
`XL_MOST150_STREAM_BUFFER_ERROR_OVERFLOW`

Overflow bit signalizing that data got lost. The status indicates a gap in streaming data between this buffer event and the preceding one. Rx stream still active.

> **labelInfo**

Bit field containing the state of connection label(s). After Rx streaming is started, one or more CL(s) may be de-allocated. This will be reported in the `labelInfo` and fill bytes will be inserted in order to keep MOST frame alignment.

The CL(s) are provided when calling `xIMost150StreamStart()` (parameter `pConnLabels`). The first CL corresponds to bit 0, the second to bit1 and so on:



Values: 1 → CL available; 0 → CL not available (fill bytes inserted)

Tag

`XL_MOST150_STREAM_RX_BUFFER`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.48 XL_MOST150_GEN_BYPASS_STRESS_EV



Note

This feature is available in the MOST 150 Analysis Library only.

Syntax

```
typedef struct s_xl_most150_gen_bypass_stress {
    unsigned int stressStarted;
} XL_MOST150_GEN_BYPASS_STRESS_EV;
```

Description

This event signals the start and stop of the bypass (closed) – bypass (opened) stress mode (see `xIMost150GenerateBypassStress()`).

Parameters

> **stressStarted**

`XL_MOST150_BYPASS_STRESS_STARTED`
Stress started.

`XL_MOST150_BYPASS_STRESS_STOPPED`
Stress stopped (due to application request).

`XL_MOST150_BYPASS_STRESS_STOPPED_LIGHT_OFF`
Stress stopped since MOST signal off.

`XL_MOST150_BYPASS_STRESS_STOPPED_DEVICE_MODE`
Stress stopped since current device mode is neither `XL_MOST150_DEVICEMODE_SLAVE` nor `XL_MOST150_DEVICEMODE_RETIMED_BYPASS_SLAVE` or the application called `xIMost150SetDeviceMode()`.

Tag

`XL_MOST150_GEN_BYPASS_STRESS`

See `s_xl_event_most150.tag` in section [XLmost150event](#) on page 272.

11.6.49 XL_MOST150_SSO_RESULT_EV

Syntax

```
typedef struct s_xl_most150_sso_result {
    unsigned int status;
} XL_MOST150_SSO_RESULT_EV;
```

Description	This event is reported either by a notification after a network shutdown or after a <code>xiMost150GetSSOResult()</code> call. The event stores the reason for a MOST network shutdown.
Parameters	<p>> status</p> <p><code>XL_MOST150_SSO_RESULT_NO_RESULT</code> No result available or reset (see <code>xiMost150SetSSOResult()</code>).</p> <p><code>XL_MOST150_SSO_RESULT_NO_FAULT_SAVED</code> No fault saved - normal MOST network shutdown.</p> <p><code>XL_MOST150_SSO_RESULT_SUDDEN_SIGNAL_OFF</code> Sudden signal off detected.</p> <p><code>XL_MOST150_SSO_RESULT_CRITICAL_UNLOCK</code> Critical unlock detected.</p>
Tag	<code>XL_MOST150_SSO_RESULT</code> See <code>s_xl_event_most150.tag</code> in section XLmost150event on page 272 .

11.7 Application Examples

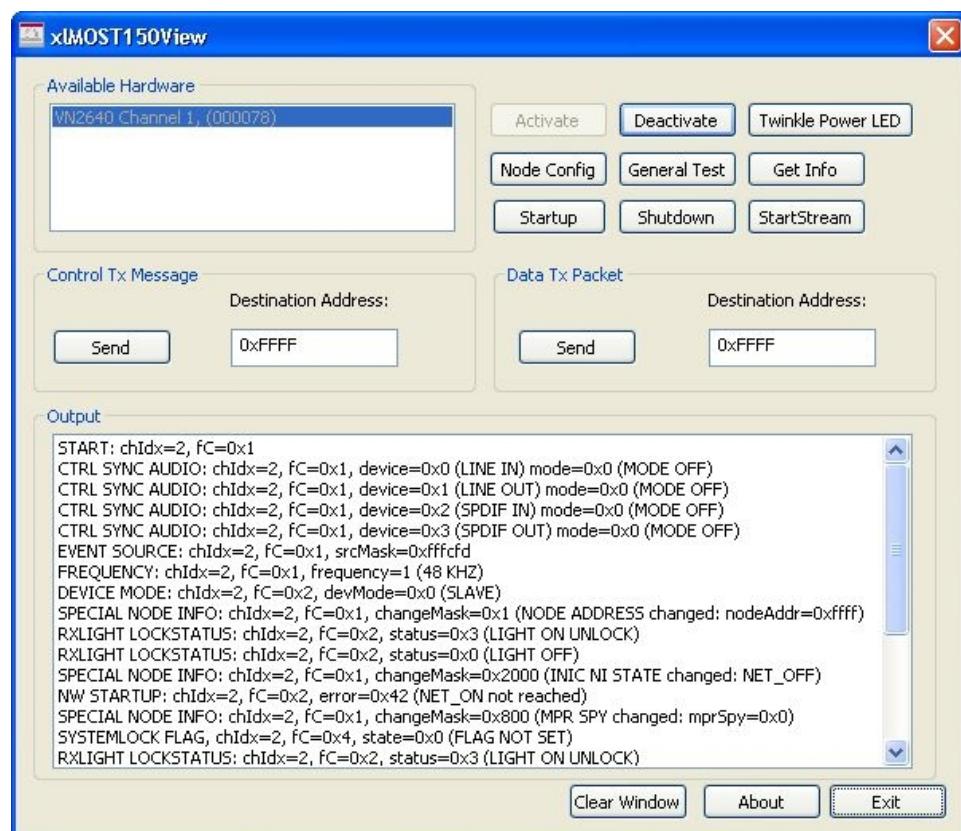
11.7.1 xIMOST150View

11.7.1.1 General Information

Description

This example demonstrates the basic handling of the XL MOST 150 API. After execution, it searches for available MOST150 devices and assigns them automatically in the **Vector Hardware Configuration** tool. The found devices are shown in the **Available Hardware** box and are activated. You can select and parameterize them with the button **[Node Config]**. To send a control frame, you have to define the destination address and then press the **[Send]** button in the field **Control Tx Message**. To send a data packet, you have to define the destination address and then press the **[Send]** button in the field **Data Tx Packet**. The **Output** box shows the return events of every function call or incoming messages.

The `xIMost150StreamTransmitData()` and the `xIMost150StreamTransmitData()` button are only available if the MOST150 Analysis Library is being used. The streaming function can be used e. g. with CANoe and audio data routing via line in. The audio data will be streamed to a file



11.7.1.2 Classes

Description

The example has the following class structure:

- > **CGeneral**
Every MOST150 device has a parameter class. There the node group address is saved for example.
- > **CNodeParam**
Contains the MOST150 node parameter.
- > **CMOST150Functions**
Implementation of all library functions.
- > **CMOST150GeneralTest**
Implementation of the General Test dialog box.
- > **CMOST150NodeConfig**
Implementation of the Node Config dialog box.
- > **CMOST150ParseEvent**
Contains an event parser to display the received events.
- > **CMOST150Streaming**
Includes the streaming feature.

11.7.1.3 Functions

Description	> CGeneral
	Contains only general functions for handling, e. g. string converting.

> **CMOST150Functions**

Implementation for the XL MOST API handling.

MOST150Init

Initializes all connected MOST150 devices. For every device a thread is created. Every device gets a separate port which is activated.

MOST150Close

Close the threads and port handles.

MOST150Activate

Activates the selected MOST150 channel.

MOST150Deactivate

Deactivates the selected MOST150 channel.

MOSTCtrlTransmit

Transmits a control frame to the selected channel.

MOST150AsyncTransmit

Transmits an asynchronous frame to the selected channel.

MOST150SetupNode

Sets up the MOST node (node group address, device mode and .frequency).

MOST150NwStartup

Triggers a network startup.

MOST150NwShutdown

Triggers a network shutdown.

MOST150GetInfo

Requests the information of a MOST150 channel (like timing mode, bypass mode...).

MOST150TwinklePowerLed

Twinkles the power LEDs.

MOST150GenerateLightError

Generates light errors depending on the counter.

MOST150GenerateLockError

Generates lock errors depending on the counter.

> **CMOST150GeneralTest**

Handles the dialog box MOST150 General Test.

> **CMOST150NodeConfig**

Handles the dialog box MOST150 Node Config.

> CMOST150Streaming

MOST150StreamStart

Checks for available connection labels (CL) and opens the stream for a given CL. As soon as the stream was successfully opened, streaming is automatically started and the streaming data is stored in `most150.bin` log file.

MOST150StreamStop

Stop streaming. As soon as the streaming is stopped, the stream is automatically closed.

MOST150StreamParseEvent

Parses streaming events as well as allocation information and MOST state events. Additionally the buffer events are handled and streaming data is stored into the log file-

12 FlexRay Commands

In this chapter you find the following information:

12.1 Introduction	311
12.2 Flowchart	312
12.3 Free Libray and Advanced Library	313
12.4 FlexRay Basics	314
12.5 Functions	320
12.6 Structs	327
12.7 Events	335
12.8 Application Examples	349

12.1 Introduction

Description

The **XL Driver Library** enables the development of FlexRay applications for supported Vector devices (see section [System Requirements](#) on page 28).

Depending on the channel property **init access** (see page 25), 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.

12.2 Flowchart

Calling sequence

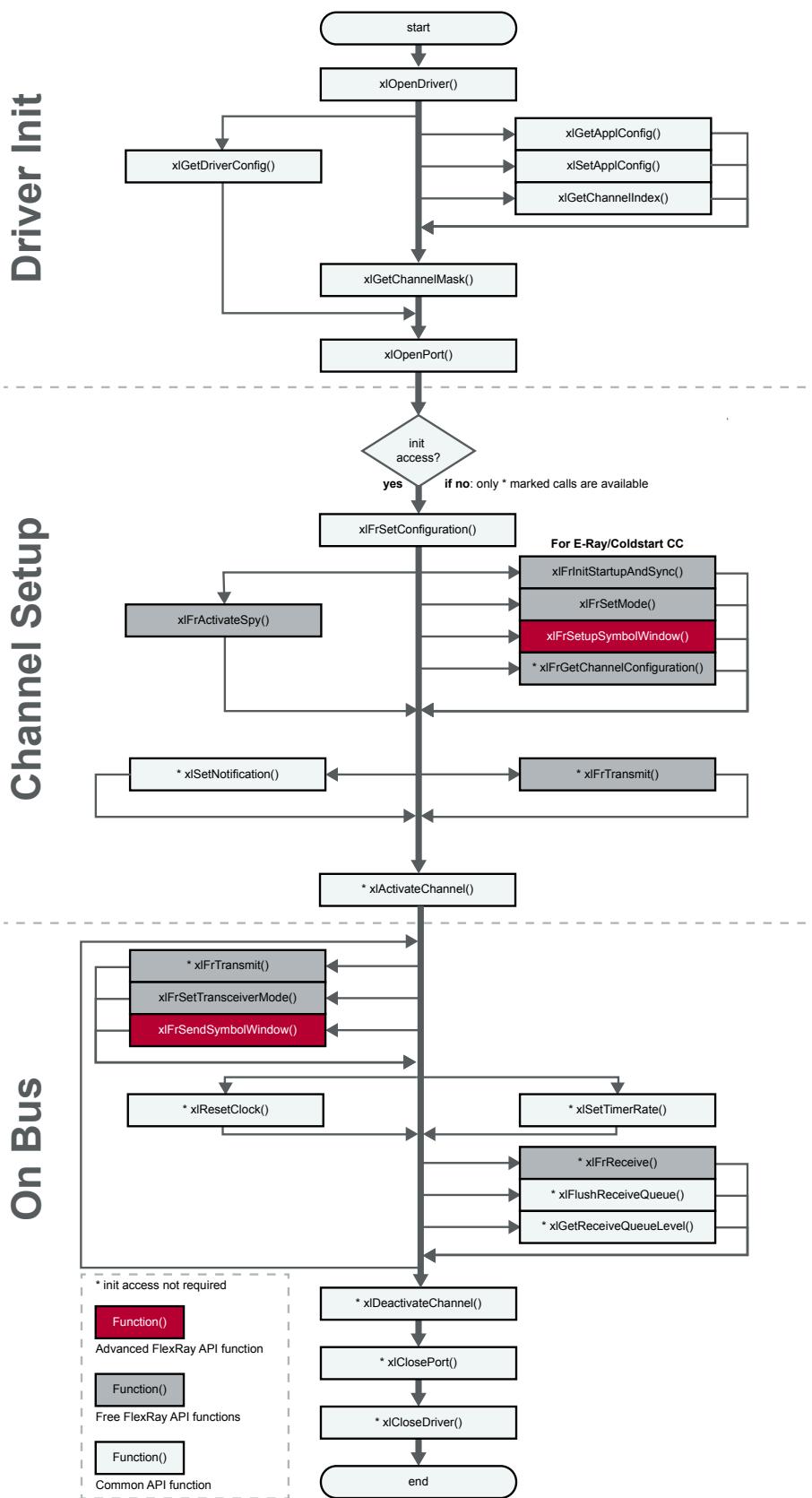


Figure 21: Function calls for FlexRay applications

12.3 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
xlFrSetTransceiverMode	X	X
xlFrAcceptanceFilter	-	X



Note

The advanced version requires a license in the FlexRay interface.

12.4 FlexRay Basics

12.4.1 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 requirements due to its event-driven and priority-driven bus access, its limited bandwidth of 500 KBit/sec based on physical constraints in the automobile, and lack of fault-tolerant structures and mechanisms.	

12.4.2 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 number 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.

12.4.3 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 application-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 message (Time Division Multiple Access - TDMA) and thereby prescribes the send times of all FlexRay messages.

FlexRay communication

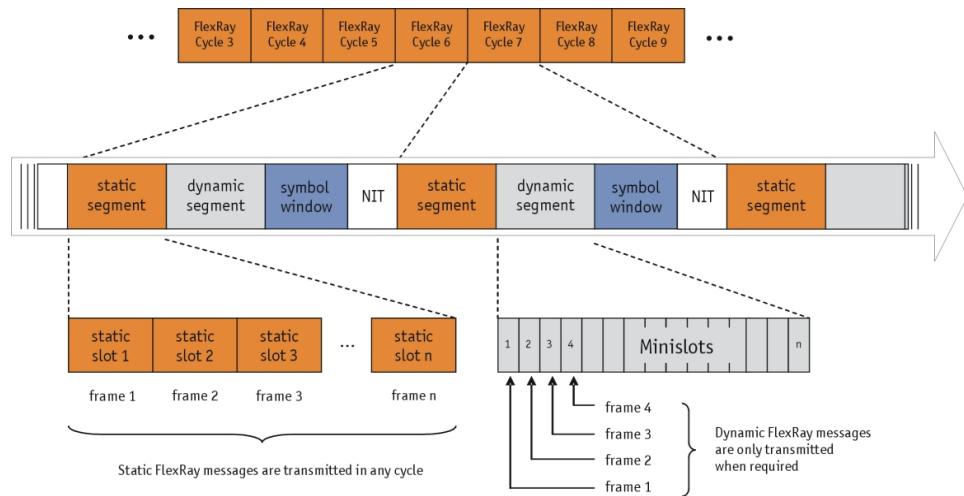


Figure 22: Principle of FlexRay communication

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 underlying 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 communication 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 transmission is electrical.

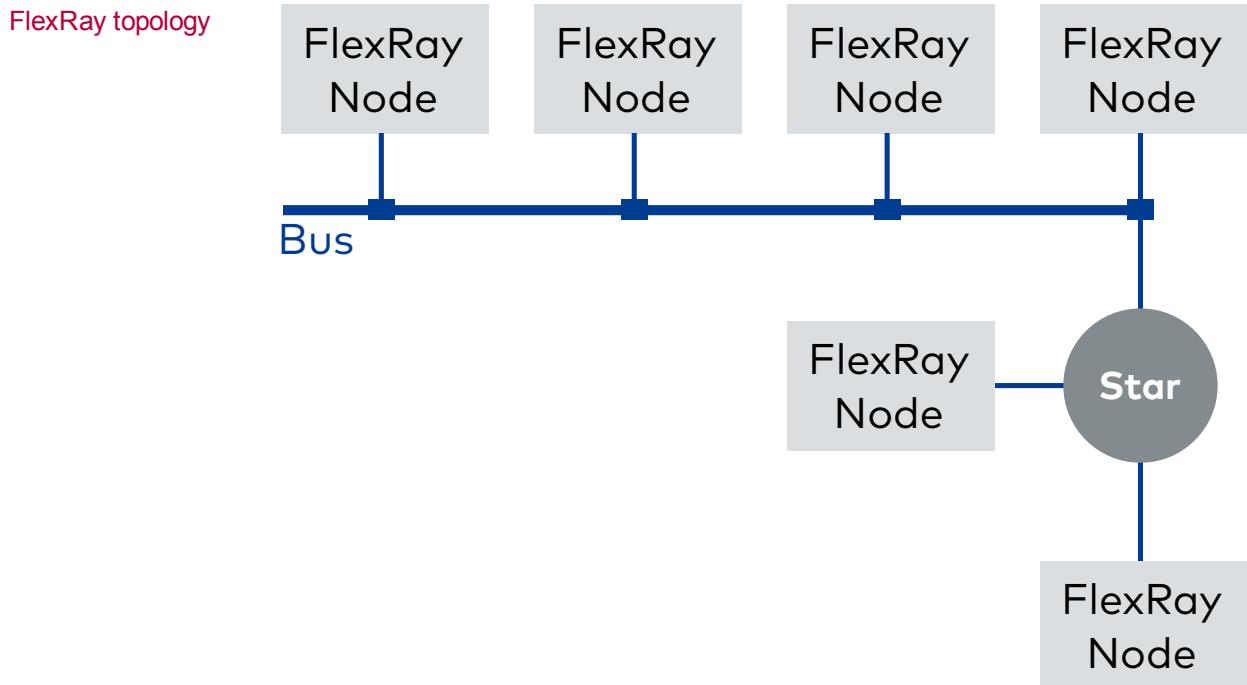


Figure 23: Combined topology of passive bus and active star

Redundant communication

To minimize failure risk, FlexRay offers redundant layout of the communication channel. 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

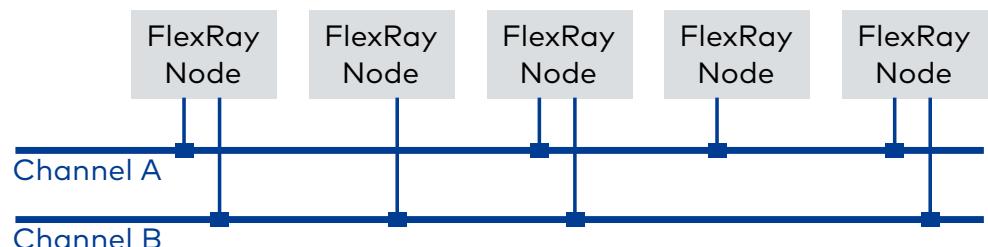


Figure 24: Passive bus structure with two communication channels minimizes failure risk

12.4.4 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

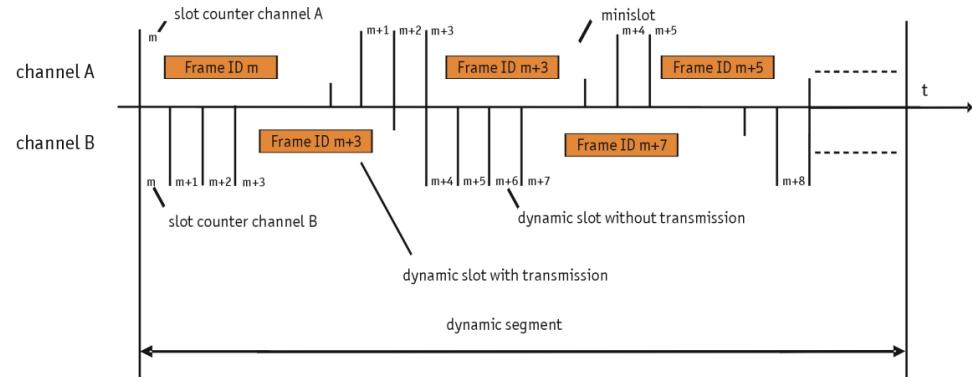


Figure 25: Passive bus structure with two communication channels minimizes failure risk

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 transmission 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.

12.4.5 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

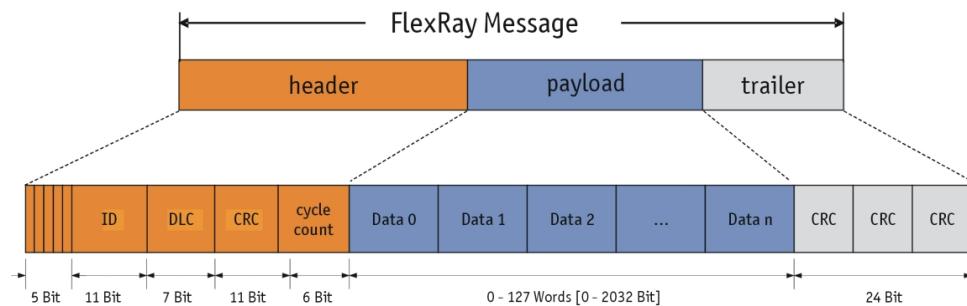


Figure 26: Structure of the FlexRay message with header, payload and trailer

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.

12.5 Functions

12.5.1 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **pxlFrClusterConfig**
Pointer to the cluster config structure (see section `XLfrClusterConfig` on page 327).

Return value

Returns an error code (see section `Error Codes` on page 423).

12.5.2 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.
- > **pxlFrChannelConfig**
Pointer the config structure (see section `XLfrChannelConfig` on page 332). Contains the cluster configuration parameters.

Return value

Returns an error code (see section `Error Codes` on page 423).

12.5.3 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 cold-start (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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **frMode**

Structure of different operational modes (see section `XLfrMode` on page 332).

Return value

Returns an error code (see section `Error Codes` on page 423).

12.5.4 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 `pKeySlotUsedForSync` 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **pEventBuffer**

Pointer to the event buffer which includes the sync frame (see section `XLfrEvent` on page 335). 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 423).

12.5.5 xlFrSetupSymbolWindow

**Note**

This function is available in the advanced version only.

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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).

12.5.6 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 `xIFrSetConfiguration()` and requires **init access**.

Input parameters

> **portHandle**

The port handle retrieved by `xIOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xIGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **mode**

Mode of the Spy:

`XL_FR_SPY_MODE_ASYNCROUS`

Return value

Returns an error code (see section `Error Codes` on page 423).

12.5.7 `xISetTimerBaseNotify`

Syntax

```
XLstatus xISetTimerBaseNotify(
    XLportHandle portHandle,
    XLhandle      *pHandle)
```

Description

Sets up an event to notify the application based on the timerrate which can be set by `xISetTimerRate()` and `xISetTimerRateAndChannel()`.

Input parameters

> **portHandle**

The port handle retrieved by `xIOpenPort()`.

Output parameters

> **pHandle**

Pointer to a WIN32 event handle.

Return value

Returns an error code (see section `Error Codes` on page 423).

12.5.8 `xIFrReceive`

Syntax

```
XLstatus xIFrReceive(
    XLportHandle portHandle,
    XLfrEvent     *pEventBuffer)
```

Description

Reads one event from the FlexRay receive queue. Calls to `xIFrReceive()` can be triggered by a notification event (see section `xISetNotification` on page 41). 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 `xIOpenPort()`.

- > **pEventBuffer**
Pointer to an application buffer in which the received event is copied (see section [XLfrEvent](#) on page 335).

Return value Returns an error code (see section [Error Codes](#) on page 423).

12.5.9 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 function returns with error as well.

Input parameters

- > **portHandle**
The port handle retrieved by [xlOpenPort\(\)](#).
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **pEventBuffer**
Pointer to the event buffer (see section [XLfrEvent](#) on page 335).
Buffersize: `XL_FR_MAX_EVENT_SIZE`

Return value

Returns an error code (see section [Error Codes](#) on page 423).

12.5.10 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).

12.5.11 xlFrSendSymbolWindow

**Note**

This function is available in the advanced version only.

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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> **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 423).

12.5.12 xlFrSetAcceptanceFilter

**Note**

This function is available in the advanced version only.

Syntax

```
XLstatus xlFrSetAcceptanceFilter(  
    XLportHandle      portHandle,  
    XLaaccess         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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

> 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 333).

Return value

Returns an error code (see section [Error Codes](#) on page 423).

12.6 Structs

12.6.1 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 vExternoOffsetControl;
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 support. 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.

12.6.2 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_ASYNCNOMOUS
- > **reserved**
Reserved for future use.
- > **xlFrClusterConfig**
The cluster config (see section [XLfrClusterConfig](#) on page 327).

12.6.3 XLfrMode

Syntax

```
struct s_xl_fr_set_modes {
    unsigned int frMode;
    unsigned int frStartupAttributes;
    unsigned int reserved[30];
} XLfrMode
```

Parameters> **portHandle**

The port handle retrieved by `xlOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **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.

> **frStartupAttributes**

`XL_FR_MODE_NONE`

No startup attribute set (default).

`XL_FR_MODE_WAKEUP`

Sets up the CC into the wakeup 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.

> **reserved**

Reserved for future use. Has to be set to 0.

12.6.4 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

12.7 Events

12.7.1 XLfrEvent

Syntax

```
struct s_xl_fr_event {  
    unsigned int          size;  
    XLfrEventTag         tag;  
    unsigned short        channelIndex;  
    unsigned int          userHandle;  
    unsigned short        flagsChip;  
    unsigned short        reserved;  
    XLuint64              timeStamp;  
    XLuint64              timeStampSync;  
    union s_xl_fr_tag_data tagData;  
} XLfrEvent;
```

Description

> **size**

Overall size of the event (in bytes).

The maximum size is defined by `XL_FR_MAX_EVENT_SIZE`.

> **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_TRANSEIVER_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 synchronization 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 **Vector Hardware Config** tool. 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.

12.7.2 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.

12.7.3 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).

12.7.4 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 intended 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).

12.7.5 XL_FR_TXACK_FRAME**Reference**

Same as XL_FLEXRAY_RX_FRAME.

12.7.6 XL_FR_INVALID_FRAME**Reference**

Same as XL_FLEXRAY_RX_FRAME.

12.7.7 XL_FR_WAKEUP_EV**Syntax**

```
struct s_xl_fr_wakeup {
    unsigned char cycleCount;
    unsigned char wakeupStatus;
    unsigned char reserved[6];
```

```
}
```

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 expiration 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.

12.7.8 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.

12.7.9 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.

12.7.10 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.

12.7.11 XL_FR_ERROR_SYNC_FRAMES_BELOWMIN**Description**

Not enough sync frames received in cycle.

12.7.12 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.

12.7.13 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.

12.7.14 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.

12.7.15 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

12.7.16 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.

12.7.17 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 327).

12.7.18 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**

12.7.19 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.

12.7.20 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.

12.8 Application Examples

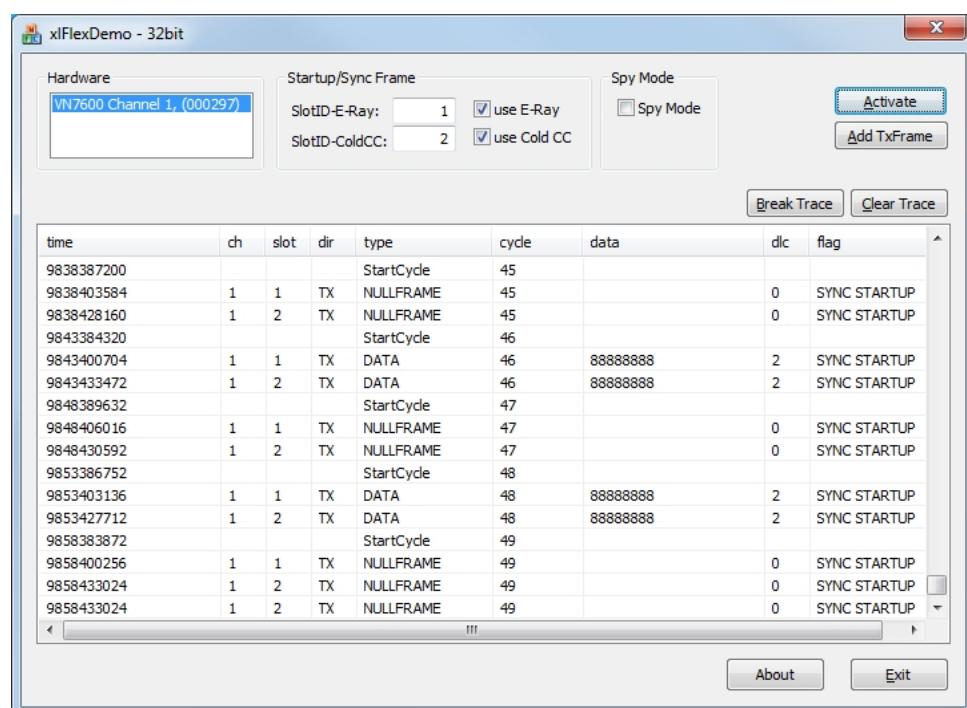
12.8.1 xlFlexDemo

12.8.1.1 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...).



12.8.1.2 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.

12.8.1.3 Functions

Description

> **FrInit**

Opens the driver and checks the FlexRay channels.

- > **FrToggleTrace**
Switches on/off the “Trace Window”.
- > **FrAddTxFrame**
Calls `xIFrTransmit()` 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.

12.8.1.4 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”.

12.8.2 xlFlexDemoCmdLine

12.8.2.1 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 activate 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.

```
C:\XLAPI\exec\xlFlexDemoCmdLine.exe
-----
          xlFlexDemoCmdLine
          Vector Informatik GmbH, Jun 10 2016
-----
x1OpenDriver...
x1GetDriverConfig...
DLL Version: V9.0.34
-----
- 06 channels      Hardware Configuration -
- Ch.: 00, CM:0x 1,      UN7600 Channel 1  FRPiggy 1082cap -
- Ch.: 01, CM:0x 2,      UN7600 Channel 2  no Cab!   -
- Ch.: 02, CM:0x 4,      UN7600 Channel 3  no Cab!   -
- Ch.: 03, CM:0x 8,      UN7600 Channel 4  no Cab!   -
- Ch.: 04, CM:0x 10,     Virtual Channel 1 Unknown Transceiver 22 -
- Ch.: 05, CM:0x 20,     Virtual Channel 2 Unknown Transceiver 22 -
-----
Flexray: found 1 Flexray channels
Flexray: x1OpenPort, PH: 0, CM: 0x1, InitM: 0x1, stat: 0
FlexRay: x1SetNotificationHandle, xlHandle: 000000B0, xlStatus: 0
Flexray: active FR CM=0x1
#main>
```

12.8.2.2 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.

12.8.3 Fibex2CSharpReaderDemo

12.8.3.1 General Information

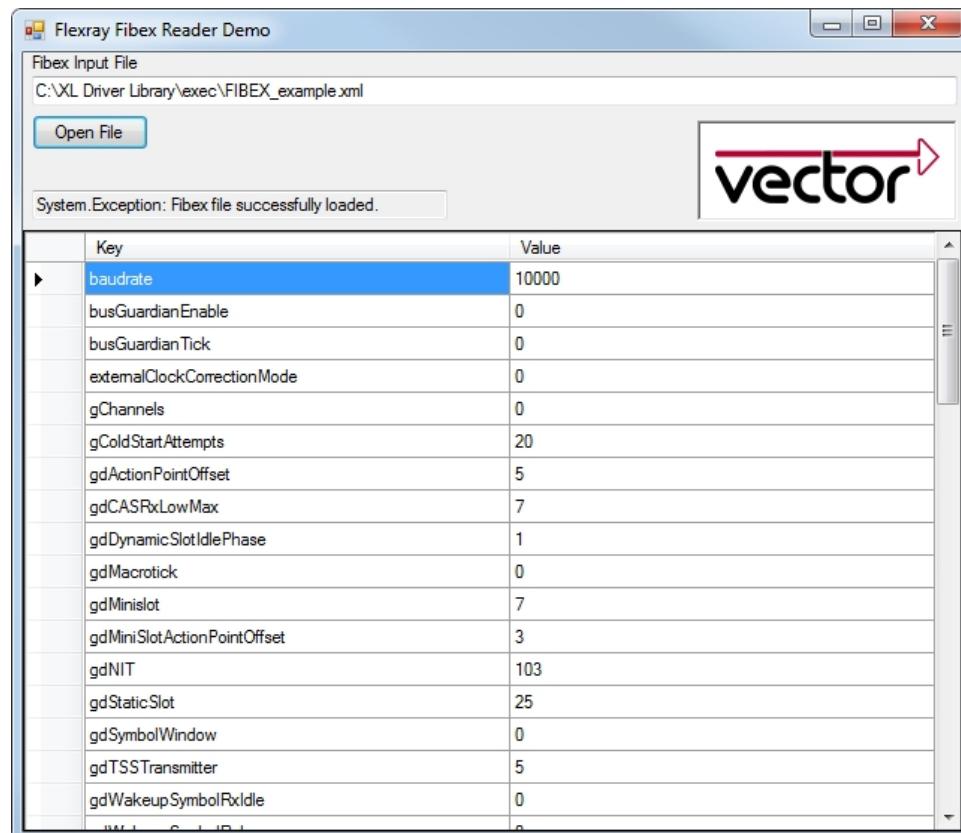
Description

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!



12.8.3.2 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 conversion of the Fibex file.

> **FibexParser.cs**

Contains the code for parsing Fibex Version 2.0.1 files.

13 Ethernet Commands

In this chapter you find the following information:

13.1 Introduction	354
13.2 Flowchart	355
13.3 Functions	356
13.4 Structs	361
13.5 Events	366
13.6 Application Examples	376

13.1 Introduction

Description

The **XL Driver Library** enables the development of Ethernet applications for supported Vector devices (see section [System Requirements](#) on page 28).

Depending on the channel property **init access** (see page 25), the application's main features are as follows:

With init access

- > channel parameters can be changed/configured
- > Ethernet frames can be received and transmitted
- > bypasses can be set and cleared

Without init access

- > Ethernet frames 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.

13.2 Flowchart

Calling sequence

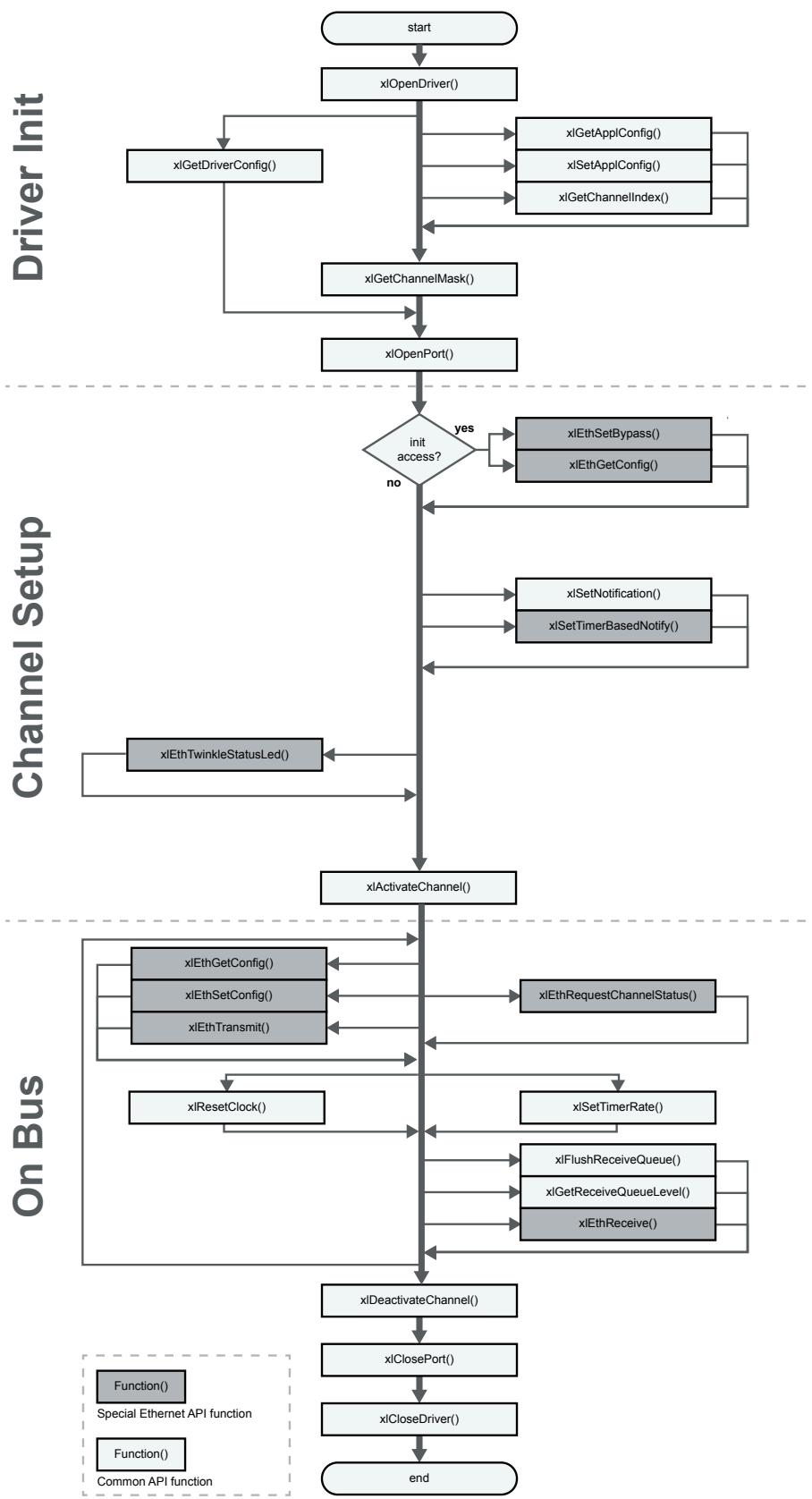


Figure 27: Function calls for Ethernet applications

13.3 Functions

13.3.1 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 [xIOpenPort\(\)](#).

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **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 363).

Return value

Returns an error code (see section [Error Codes](#) on page 423).

13.3.2 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 [xIOpenPort\(\)](#).

- > **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.

- > **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 363).

Return value Returns an error code (see section [Error Codes](#) on page 423).

13.3.3 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 frame 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.

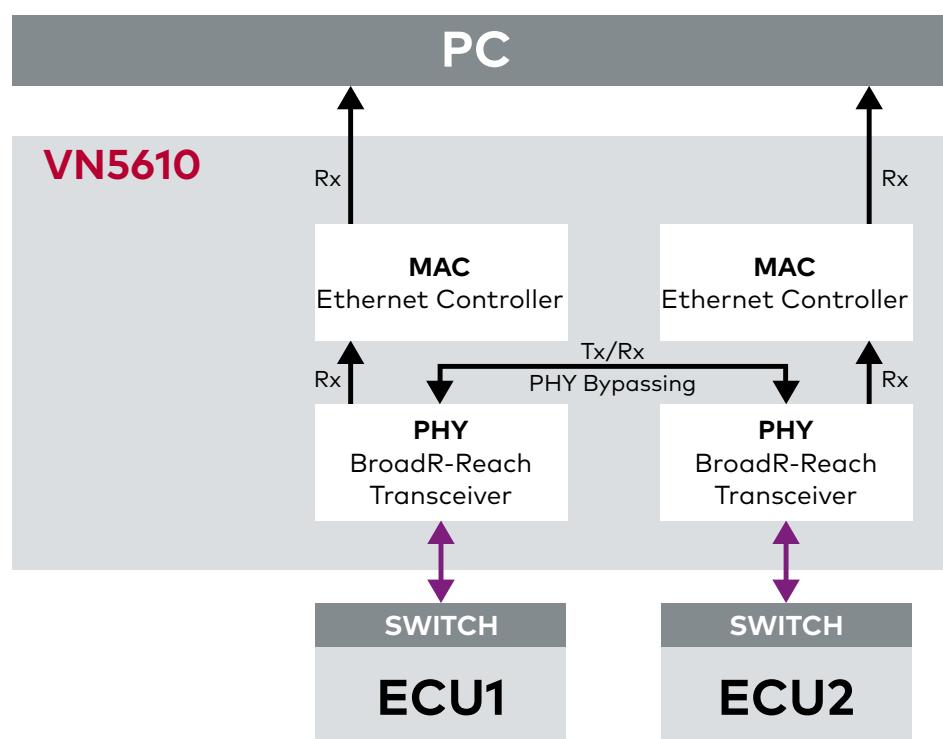


Figure 28: PHY bypassing in VN5610

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.

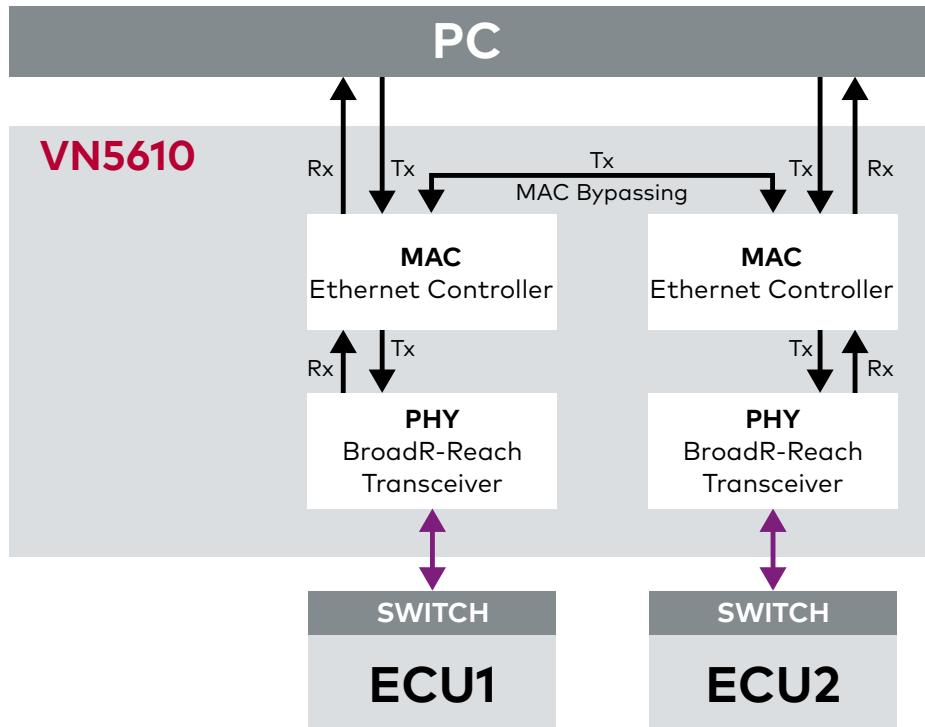


Figure 29: MAC bypassing in VN5610



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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

- > **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 423).
--------------	--

13.3.4 xlEthTransmit

Syntax

```
XLstatus xlEthTransmit (
    XLportHandle           portHandle,
    XLaccess               accessMask,
    XLuserHandle           userHandle,
    const T_XL_ETH_DATAFRAME_TX *data
)
```

Description	Transmits an Ethernet frame on the channel which is indicated in <code>accessMask</code> .
-------------	--

Input parameters

- > **portHandle**
The port handle retrieved by `xlOpenPort()`.
- > **accessMask**
The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section [xlGetChannelMask](#) on page 36). For further information on channel/access masks please also refer to section [Principles of the XL Driver Library](#) on page 23.
- > **userHandle**
The handle is created by the application and is used for the event assignment.
- > **data**
The Ethernet frame 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 423).
--------------	--

13.3.5 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 <code>xlOpenPort()</code> .
	> eventBuffer Buffer for a single Ethernet event (see section <code>T_XL_ETH_EVENT</code> on page 366).
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

13.3.6 xlEthTwinkleStatusLed

Syntax	<pre>XLstatus xlEthTwinkleStatusLed(XLportHandle portHandle, XLaccess accessMask, XLuserHandle userHandle)</pre>
Description	Twinkle the Status LED from the VN5610 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 <code>accessMask</code> .
Input parameters	> portHandle The port handle retrieved by <code>xlOpenPort()</code> . > accessMask The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the Vector Hardware Configuration tool if there is a prepared application setup (see section <code>xlGetChannelMask</code> on page 36). For further information on channel/access masks please also refer to section <code>Principles of the XL Driver Library</code> on page 23. > userHandle The handle is created by the application and is used for the event assignment.
Return value	Returns an error code (see section <code>Error Codes</code> on page 423).

13.4 Structs

13.4.1 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.

13.4.2 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, 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_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_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.

13.4.2.1 Valid Configuration Combinations

Supported 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 following tables for the supported combinations.

**RJ45
IEEE 802.3
physical layer**

Configurations for the RJ-45 connector				
speed	duplex	clockMode	mdiMode	brPairs
AUTO_100	AUTO	DON'T CARE	AUTO	DON'T CARE
AUTO_100_1000	AUTO	AUTO	AUTO	DON'T CARE
AUTO_1000	AUTO	AUTO	AUTO	DON'T CARE
FIXED_100	FULL	DON'T CARE	AUTO	DON'T CARE



Note

BroadR-Reach is not supported for the RJ-45 connector.

D-SUB9
BroadR-Reach
(LRE) mode

Configuration for the D-SUB9 connector				
speed	duplex	clockMode	mdiMode	brPairs
FIXED_100	DON'T CARE	MASTER/SLAVE	AUTO	1PAIR

Note

On the D-SUB connector, only a single cable pair is provided per channel.
IEEE 802.3 is not supported for the D-SUB9 connector.

13.5 Events

13.5.1 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

Frame data definition used inside the Rx/Tx tagData structures.

Parameters

- > **rawData**
Raw values of the Ethernet frame.
- > **etherType**
Type of protocol encapsulated within the frame.
- > **payload**
Packet payload.

13.5.2 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_BYPASSSED`
 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 367 ... page 373 for further details.

13.5.3 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 frame.

Tag XL_ETH_EVENT_TAG_FRAME_RX

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 366

13.5.4 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 frame that was received with an error.

Tag XL_ETH_EVENT_TAG_FRAME_RX_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 frame, 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 1.

13.5.5 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 frame has been sent by the application.

**Note**

The parameters destMAC, sourceMAC, etherType and payload are in ‘network byte order’.

Parameters	<ul style="list-style-type: none"> > 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 <code>xIEthTransmit()</code> for a description of allowed flags. > dataLen Combined size of <code>etherType</code> and <code>payload</code> 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 frame in nanoseconds. > reserved2 Not being used, ignore. > fcs Frame Check Sequence generated for this frame. > destMAC Destination MAC address. > sourceMAC Source MAC address. > frameData See section <code>T_XL_ETH_FRAME</code> on page 1.
-------------------	---

13.5.6 T_XL_ETH_DATAFRAME_TXACK

Syntax	<code>typedef T_XL_ETH_DATAFRAME_TX_EVENT T_XL_ETH_DATAFRAME_TXACK;</code>
Description	This event is indicated to the application each time an Ethernet frame 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	<code>XL_ETH_EVENT_TAG_FRAMETX_ACK</code>
Parameters	For a description of the structure members refer to <code>T_XL_ETH_DATAFRAME_TX_EVENT</code> .

13.5.7 T_XL_ETH_DATAFRAME_TXACK_OTHERAPP

Syntax	<code>typedef T_XL_ETH_DATAFRAME_TX_EVENT T_XL_ETH_DATAFRAME_TXACK_OTHERAPP;</code>
Description	This event indicates the successful sending of an Ethernet frame by another application.
Tag	<code>XL_ETH_EVENT_TAG_FRAMETX_ACK_OTHER_APP</code>

Parameters	For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT .
-------------------	---

13.5.8 T_XL_ETH_DATAFRAME_TXACK_SW

Syntax	<pre>typedef T_XL_ETH_DATAFRAME_TX_EVENT T_XL_ETH_DATAFRAME_TXACK_SW;</pre>
---------------	---

Description	This event is indicated to the application each time a received Ethernet frame 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_FRAME_TX_ACK_SWITCH
------------	--------------------------------------

Parameters	For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT .
-------------------	---

13.5.9 T_XL_ETH_DATAFRAME_TX_ERROR

Syntax	<pre>typedef struct s_xl_eth_dataframe_txerror { unsigned int errorType; T_XL_ETH_DATAFRAME_TX_EVENT txFrame; } T_XL_ETH_DATAFRAME_TX_ERROR;</pre>
---------------	--

Description	This event is indicated to the application each time an Ethernet frame has not been sent to the bus.
--------------------	--

Tag	XL_ETH_EVENT_TAG_FRAME_TX_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 369
---------------------	---

13.5.10 T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP

Syntax	<pre>typedef T_XL_ETH_DATAFRAME_TX_ERROR T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP;</pre>
---------------	--

Description	This event is indicated to the application each time an erroneous Ethernet frame 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](#).

13.5.11 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 frame 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](#).

13.5.12 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 `xIEthSetConfig()` was triggered.

Tag XL_ETH_EVENT_TAG_CONFIGRESULT

Parameters

- > **result**

0: Valid parameter combination set via `xIEthSetConfig()`.
 != 0: Invalid parameter combination set via `xIEthSetConfig()`.

13.5.13 T_XL_ETH_LOSTEVENT

Syntax

```
typedef struct s_xl_eth_lostevent {
    XLeventTag 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.

13.5.14 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.

> 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_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

> **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

13.5.15 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 frame that can be sent to one or more network links.

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

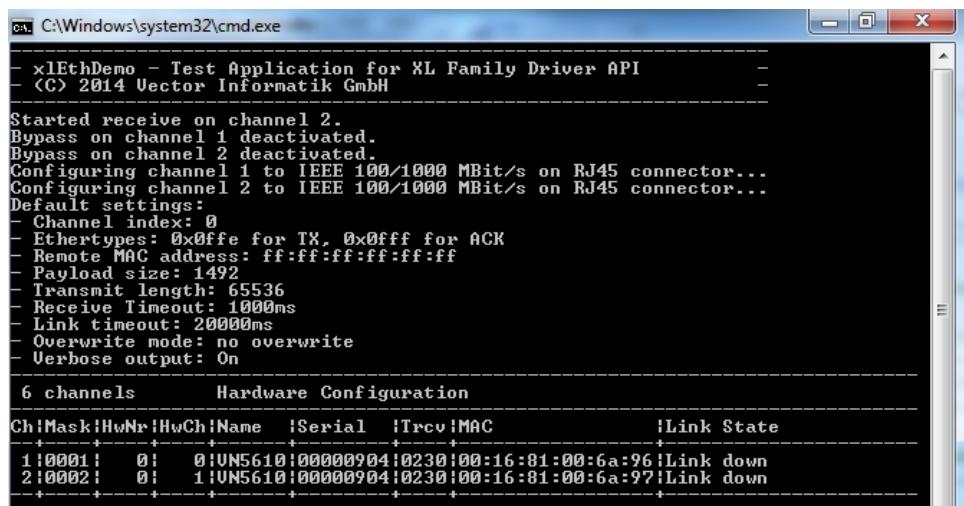
section [T_XL_ETH_FRAME](#) on page 366

13.6 Application Examples

13.6.1 xIEthDemo

13.6.1.1 General Information

This example demonstrates how to transmit/receive Ethernet frames. 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 frames in a loop e. g. by pressing <t>. It is also possible to send frames in a burst mode by pressing . To transmit a complete file via Ethernet use the command line options to start the example (/t).



13.6.1.2 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.
	Start burst transmission (needs active receiver).
<s>	Stop burst transmission.
<r>	Receive.
<e>	Set Ether type to use.
<p>	Set packet payload size.

Key	Command
< >	Set burst data length.
<m>	Set receiver MAC address.
<k>	Twinkle status LED of device.
<w>	Show driver configuration.
<v>	Toggle verbose output.

13.6.1.3 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.
/n\"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.

13.6.2 xIEthBypassDemo

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.
-------------	--

14 ARINC 429 Commands

In this chapter you find the following information:

14.1 Introduction	380
14.2 Flowchart	381
14.3 Functions	382
14.4 Structs	387
14.5 Events	395
14.6 Application Examples	403

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 28).

Depending on the channel property **init access** (see page 25), 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.



Reference

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

14.2 Flowchart

Calling sequence

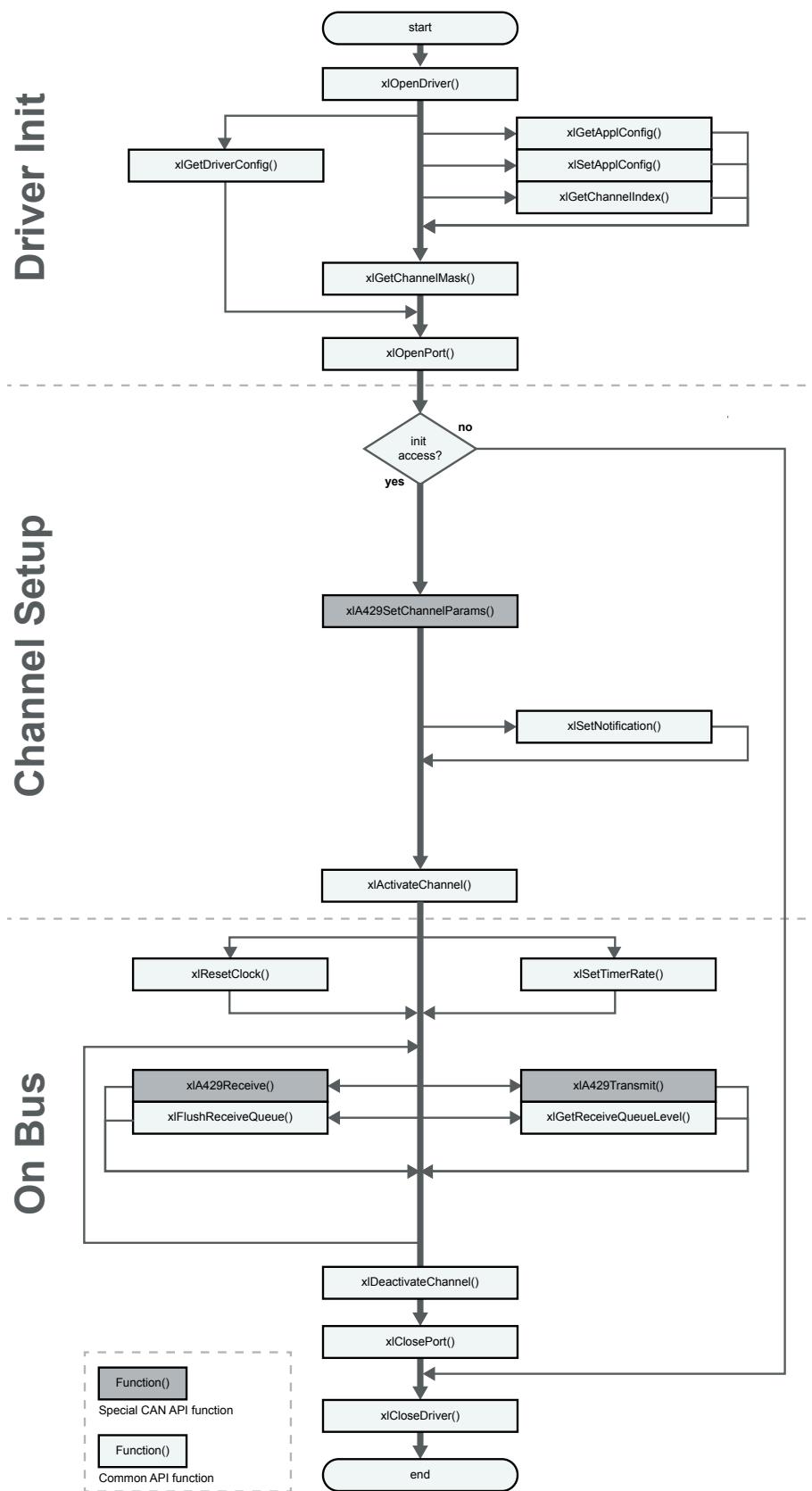


Figure 30: Function calls for ARINC 429 applications

14.3 Functions

14.3.1 xIA429SetChannelParams

Syntax

```
XLstatus xIA429SetChannelParams (
    XLportHandle    portHandle,
    XLaccess        accessMask,
    XL_A429_PARAMS* pXIA429Params
)
```

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 `xIOpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xIGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **pXIA429Params**

ARINC 429 configuration structure (see section `XL_A429_PARAMS` on page 387).



Note

Each `xIA429SetChannelParams()` call has to be called before `xIActivateChannel()` function call. Parameter changes after `xIActivateChannel()` calls (e. g. bitrate) are not supported. After `xIDeactivateChannel()`, `xIA429SetChannelParams()` can be called again.



Example

Configures an A429 channel in Tx channel direction with a bit rate of 100000 bit/s, parity calculation disabled and a default `minGap` of 4 bit.

```
XL_A429_PARAMS xIA429Params;

memset(&xIA429Params, 0, sizeof(XL_A429_PARAMS));
xIA429Params.channelDirection = XL_A429_MSG_CHANNEL_DIR_TX;
xIA429Params.data.tx.bitrate = 100000;
xIA429Params.data.tx.minGap = XL_A429_MSG_GAP_4BIT;
xIA429Params.data.tx.parity = XL_A429_MSG_PARITY_DISABLED;

XLstatus = xIA429SetChannelParams(xlPortHandle,
                                    xlChannelMask,
                                    &xIA429Params);
```

**Example**

Configures an A429 channel in Rx channel direction with enabled bit rate detection (expected bit rate 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 `xA429Transmit()` 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. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `xlGetChannelMask` on page 36). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 23.

> **msgCnt**

Amount of messages to be transmitted.

> **pXIA429MsgTx**

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 387).

Output parameters

> **pMsgCntSent**

Number of messages successfully transferred to the transmit queue.

Return value

Returns an error code (see section [Error Codes](#) on page 423).

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 x1A429MsgTx;
unsigned int msgCnt      = 1;
unsigned int msgCntSent = 0;
memset(&x1A429MsgTx, 0, sizeof(XL_A429_MSG_TX));
x1A429MsgTx.userHandle = 0;
x1A429MsgTx.flags     = XL_A429_MSG_FLAG_ON_REQUEST;
x1A429MsgTx.label     = 0x04;
x1A429MsgTx.gap        = 32;
x1A429MsgTx.parity    = XL_A429_MSG_PARITY_DEFAULT;
x1A429MsgTx.data       = 0xAABBCC;
x1Status = x1A429Transmit(portHandle,
                           accessMask,
                           msgCnt,
                           &msgCntSent,
                           &x1A429MsgTx);
```

**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 x1A429MsgTx;
unsigned int msgCnt      = 1;
unsigned int msgCntSent = 0;
memset(&x1A429MsgTx, 0, sizeof(XL_A429_MSG_TX));
x1A429MsgTx.userHandle = 0;
x1A429MsgTx.cycleTime  = 100000;
x1A429MsgTx.flags     = XL_A429_MSG_FLAG_CYCLIC;
x1A429MsgTx.label     = 0x04;
x1A429MsgTx.gap        = 64;
x1A429MsgTx.parity    = XL_A429_MSG_PARITY_DEFAULT;
x1A429MsgTx.data       = 0xAABBCC;
x1Status = x1A429Transmit(portHandle,
                           accessMask,
                           msgCnt,
                           &msgCntSent,
                           &x1A429MsgTx);
```

**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 x1A429MsgTx[100];
unsigned int msgCnt      = 100;
unsigned int msgCntSent = 0;
memset(x1A429MsgTx, 0, sizeof(XL_A429_MSG_TX));
for (i=0; i<nMsgCnt;i++) {
    x1A429MsgTx[i].userHandle = 0;
    x1A429MsgTx[i].flags     = XL_A429_MSG_FLAG_ON_REQUEST;
    x1A429MsgTx[i].label     = 0x04;
    x1A429MsgTx[i].gap       = XL_A429_MSG_GAP_DEFAULT;
    x1A429MsgTx[i].parity    = XL_A429_MSG_PARITY_ODD;
    x1A429MsgTx[i].data      = 0xAABBCC;
}
x1Status = x1A429Transmit(portHandle,
                           accessMask,
                           msgCnt,
                           &msgCntSent,
                           x1A429MsgTx);
```

14.3.3 x1A429Receive

Syntax

```
XLstatus x1A429Receive (
    XLportHandle portHandle,
    XLa429Event* pX1A429Event
)
```

Description

Retrieves one event from the event queue. This operation is synchronous.

Input parameters

> **portHandle**

The port handle retrieved by `x1OpenPort()`.

> **accessMask**

The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the **Vector Hardware Configuration** tool if there is a prepared application setup (see section `x1GetChannelMask` on page 36). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 23.

> **pX1A429Event**

Pointer to the application allocated receive event buffer (see section `XLa429Event` on page 395).

Return value

Returns an error code (see section `Error Codes` on page 423).



Example

Read each message from the message queue

```
XL429Event xlA429Event;

x1Status = xlA429Receive(portHandle, &xlA429Event);

if (x1Status != 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_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**

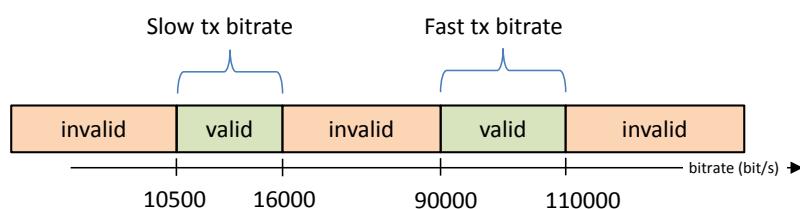
Reserved for future use.

> **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

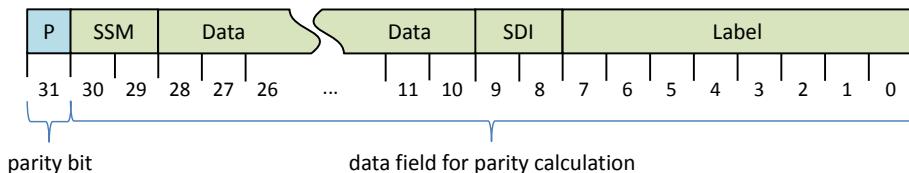
Disables the parity calculation. For each transmitted Tx message the parity information has to be passed in the data field parameter separately.

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

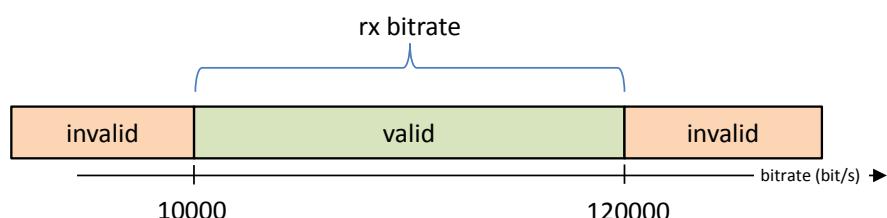
Specifies the global minimum gap time between two consecutive messages. The configured gap time is inserted before a message is transmitted. Minimum Gap between two messages is defined in 1/8 bit time steps. This value is limited to 2047 (a min gap time of 255 bit). At a bitrate of 100000 bit/s the bit 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)

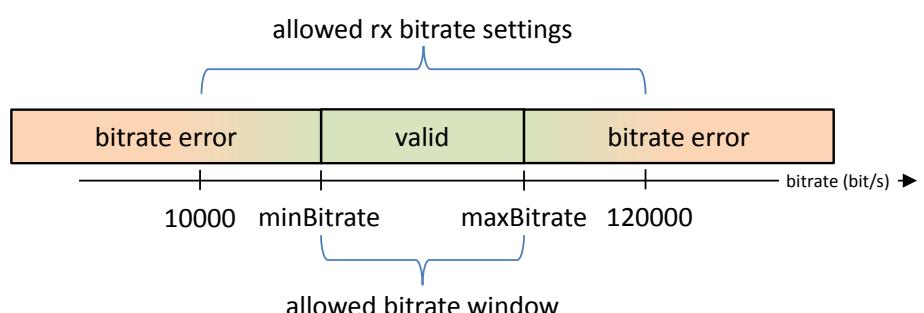


> **rx.minBitrate**

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_MAX). 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_MAX`). 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 `xIA429SetChannelParams()` and not the measured/configured values of the hardware interface.



Example

Configures an A429 channel in Rx channel direction with enabled bit rate detection (expected bit rate 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 xIA429Params;
memset(&xIA429Params, 0, sizeof(XL_A429_PARAMS));
xIA429Params.channelDirection
= XL_A429_MSG_CHANNEL_DIR_RX;
xIA429Params.data.rx.autoBaudrate
= XL_A429_MSG_AUTO_BAUDRATE_ENABLED;
xIA429Params.data.rx.minBitrate = 97500;
xIA429Params.data.rx.maxBitrate = 102500;
xIA429Params.data.rx.parity      = XL_A429_MSG_PARITY_ODD;
xIA429Params.data.rx.minGap     = XL_A429_MSG_GAP_4BIT;
xIStatus = xIA429SetChannelParams(xlPortHandle, xlChannelMask,
&xIA429Params);
```

**Example**

Configures an A429 channel in x channel direction with disabled bit rate detection (bit rate 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;
xIStatus = xlA429SetChannelParams(xlPortHandle, xlChannelMask,
&xlA429Params);
```

14.4.2 XL_A429_MSG_TX

Syntax

```
typedef struct s_xl_a429_msg_tx {
    unsigned short userHandle;
    unsigned short 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 immediately 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 `xIA429Transmit()` 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 Control Panel. Offset correction is possible with `xlResetClock()`.

14.5.2 XL_A429_EV_TX_OK

Syntax

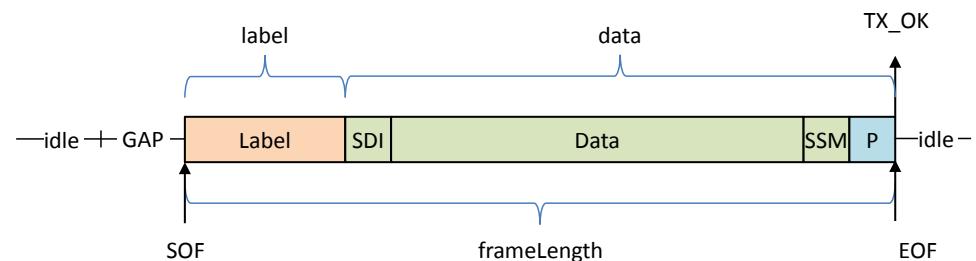
```
typedef struct s_xl_a429_ev_tx_ok {
    unsigned int    frameLength;
    unsigned int    bitrate;
    unsigned char   label;
    unsigned char   msgCtrl;
    unsigned short  res1;
    unsigned int    data;
} XL_A429_EV_TX_OK;
```

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).

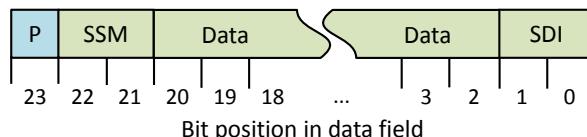
XL_A429_MSG_CTRL_ON_REQUEST
XL_A429_MSG_CTRL_CYCLIC

> **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

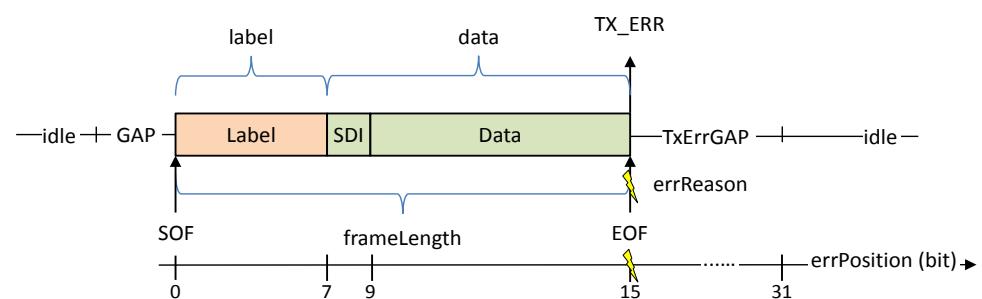
```
typedef struct s_xl_a429_ev_tx_err {
    unsigned int frameLength;
    unsigned int bitrate;
    unsigned char errorPosition;
    unsigned char errorReason;
    unsigned char label;
    unsigned char res1;
    unsigned int data;
} XL_A429_EV_TX_ERR;
```

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

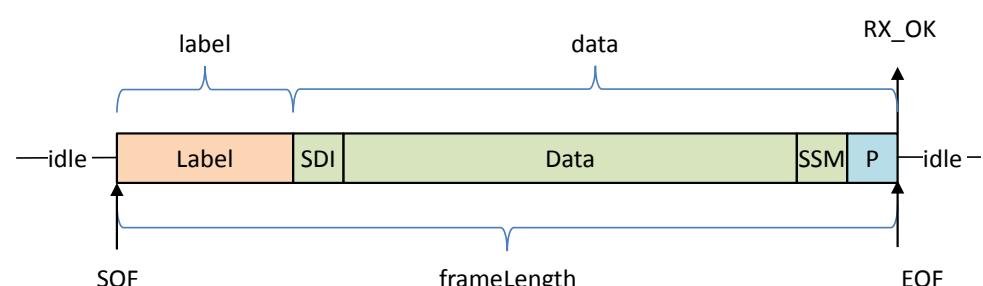
```
typedef struct s_xl_a429_ev_rx_ok {
    unsigned int frameLength;
    unsigned int bitrate;
    unsigned char label;
    unsigned char res1[3];
    unsigned int data;
} XL_A429_EV_RX_OK;
```

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**

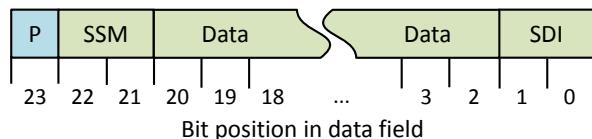
label of ARINC word.

> **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

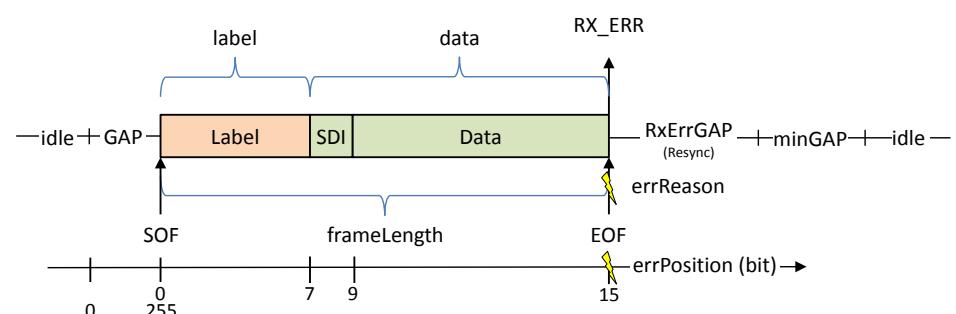
```
typedef struct s_xl_a429_ev_rx_err {
    unsigned int framelength;
    unsigned int bitrate;
    unsigned int bitLengthOfLastBit;
    unsigned char errorPosition;
    unsigned char errorReason;
    unsigned char label;
    unsigned char res1;
    unsigned int data;
} XL_A429_EV_RX_ERR;
```

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 nanoseconds.
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:

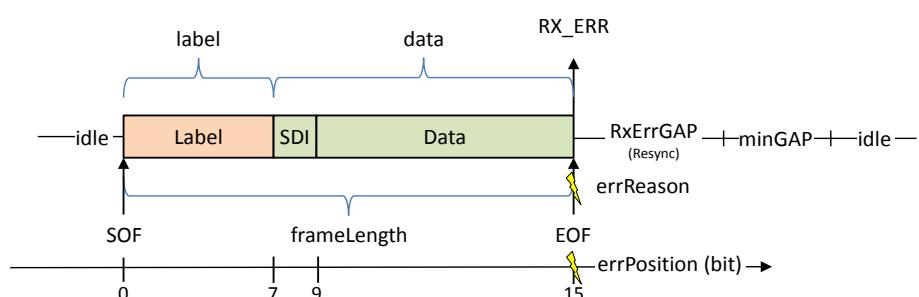
`XL_A429_EV_RX_ERROR_PARITY`

`XL_A429_EV_RX_ERROR_DUTY_FACTOR`

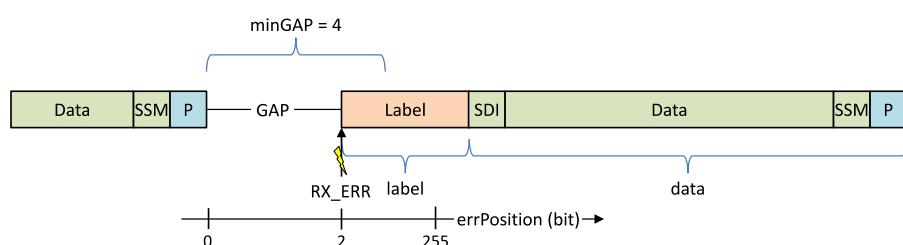
`XL_A429_EV_RX_ERROR_AVG_BIT_LENGTH`

> **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 exceeded 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 -10V to 10V 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 ($\pm 1.0\%$).

> **label**

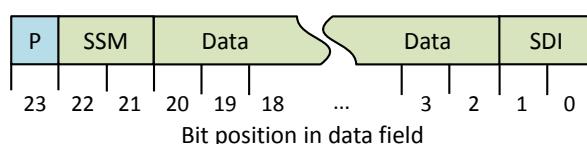
Label of ARINC word. If error position > 7 the value is valid (except `minGap` violation error). Label field does not contain valid values for `minGap` violation error.

> **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;
```

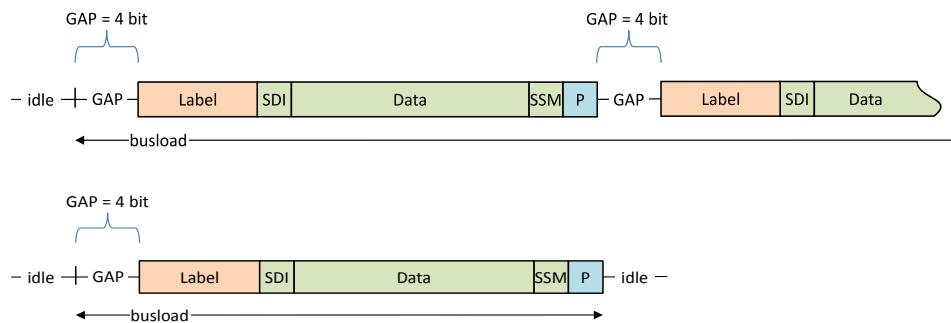
```
unsigned int res1[3];
} XL_A429_EV_BUS_STATISTIC;
```

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 xIA429Control

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.

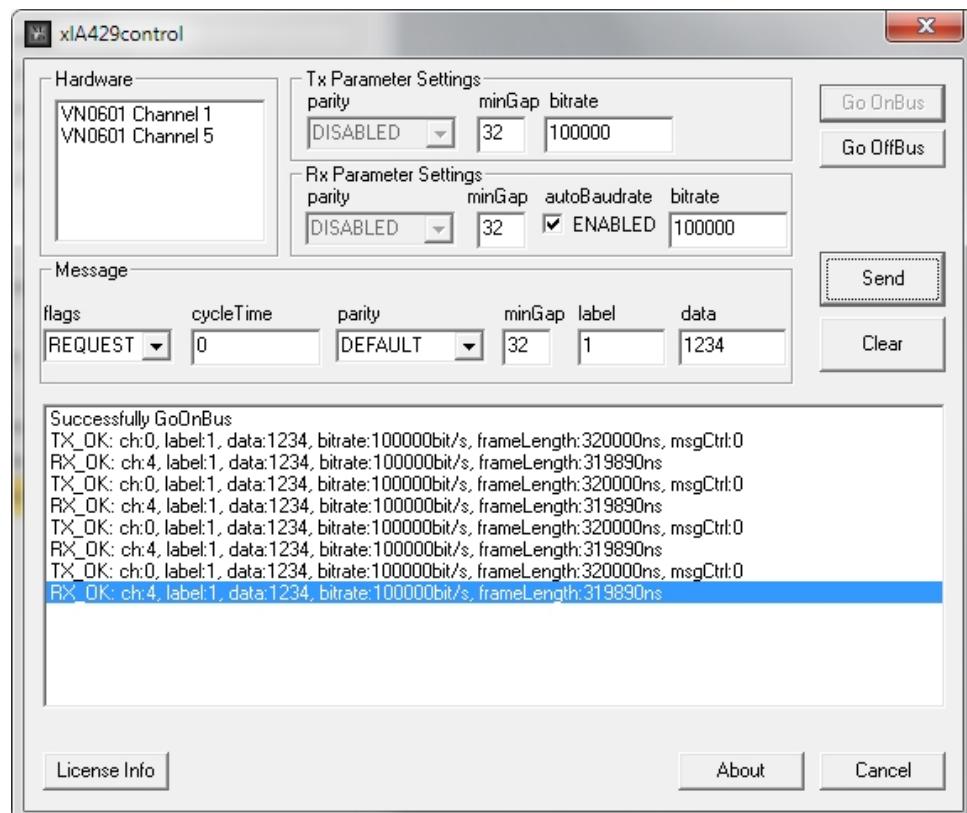


Figure 31: Running xIA429Control

15 .NET Wrapper

In this chapter you find the following information:

15.1 Overview	405
15.2 XLDriver - Accessing Driver	406
15.3 XLClass - Storing Data/Parameters	407
15.4 XLDefine - Using Predefined Values	408
15.5 Including the Wrapper in a New .NET Project	409
15.6 Application Examples	411

15.1 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 is 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:	<code>XL_OpenDriver()</code>
Native XL API:	<code>xlOpenDriver()</code>

The required parameters of the .NET methods can be looked up by using the Intel-iSense feature of the IDE, for example:

```
// Activate channel
status = CANDemo.XL_ActivateChannel(portHandle, accessMask, busTypeCAN, XLDefine.XL_AC_Flags.XL_
Console.WXLDefine.XL_Status XLDriver.XL_ActivateChannel(int portHandle, ulong accessMask, XLDefine.XL_BusTypes busType, XLDefine.XL_AC_Flags flags)
if (status != portHandle: The port handle returned by xlOpenPort().
```

Examples

The **XL Driver Library** setup also contains a few examples in different .NET languages that explain the usage in each environment.



Caution!

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 on the Vector Driver Disk in \Drivers\XL Driver Library\bin.

In order to run the .NET wrapper with your application, the general libraries `vxlap-i.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.

15.2 XLDriver - Accessing Driver

Accessing XL API

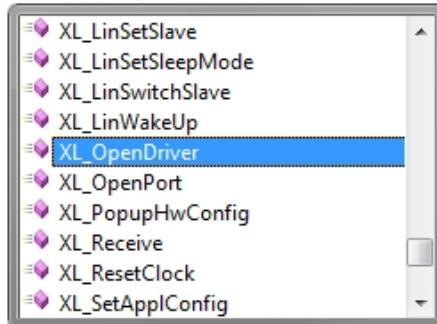
In .NET, the native XL API can be accessed by the major class `XLDriver` which supports most of the native functions.



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`.

```
XLDriver myApp = new XLDriver();  
myApp.XL_Op|
```



XLDefines.XL_Status XLDriver.XL_OpenDriver()
Opens the XL Driver.

15.3 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.

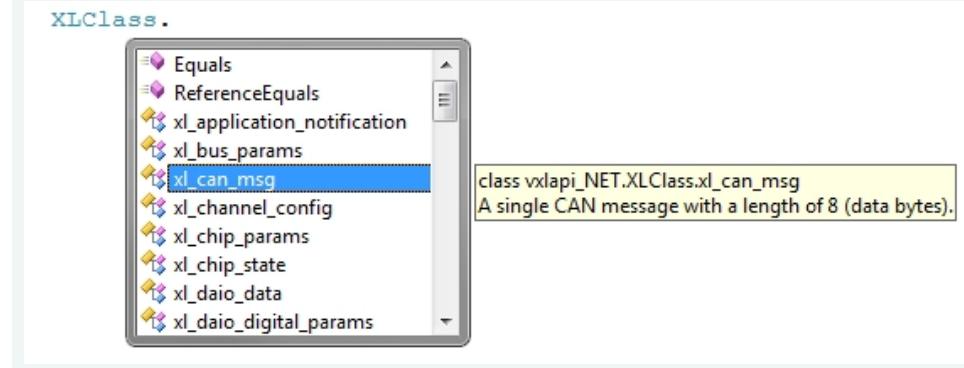
Here are some examples of these predefined classes:

- > **`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_SetChannelParams()`.
- > **`xl_linStatPar`**
Used by method `XL_LinSetChannelParams()`.



Note

More predefined classes in `XLClass` can be found via the IntelliSense feature in your IDE.



15.4 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.

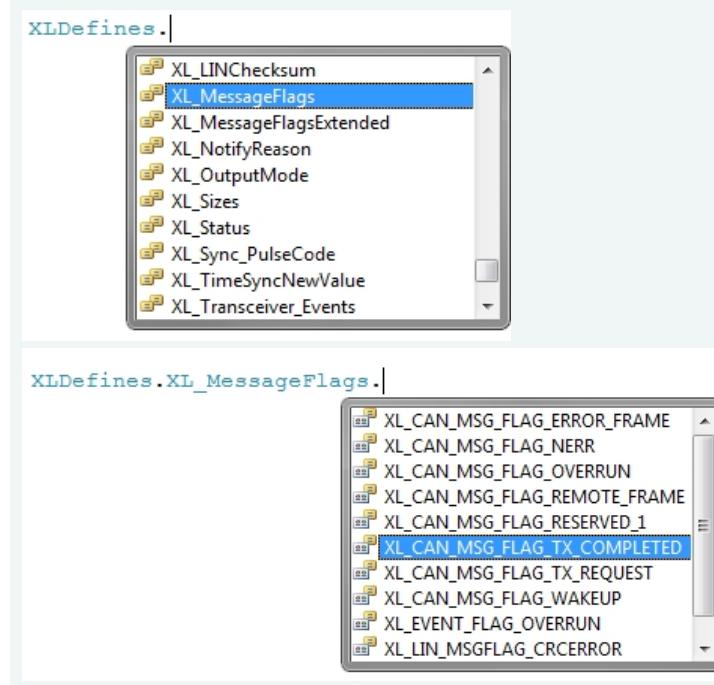
Here are some examples of these predefined definitions:

- > **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`.



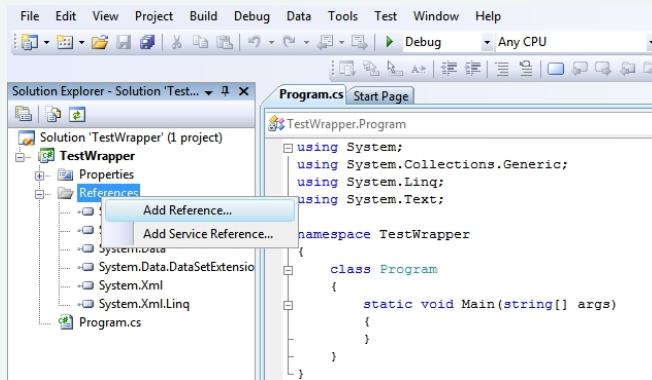
15.5 Including the Wrapper in a New .NET Project



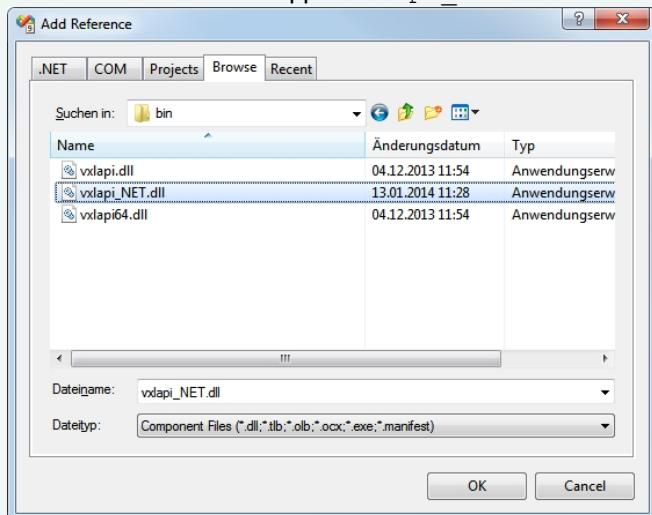
Step by Step Procedure

1. Copy the general XL Driver Library vxlapi.dll/vxlapi64.dll to your execution folder of your project (\Debug or \Release).

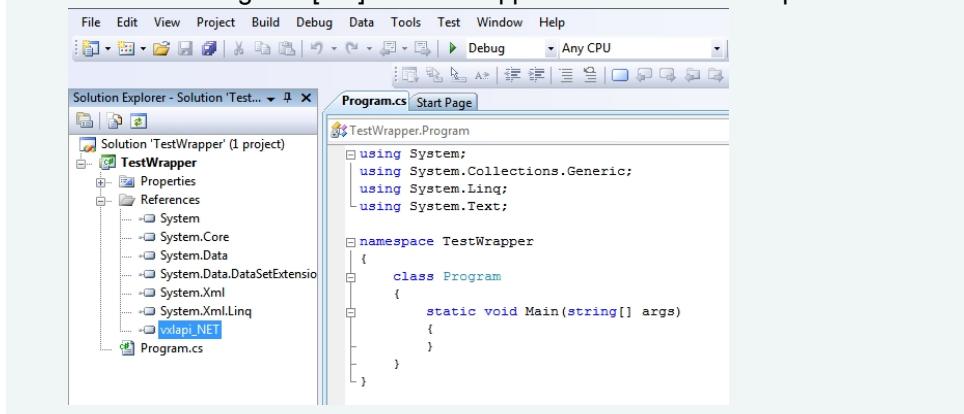
2. In VS2008, right-click on **References** (Solution Explorer) and select **Add Reference...**



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:

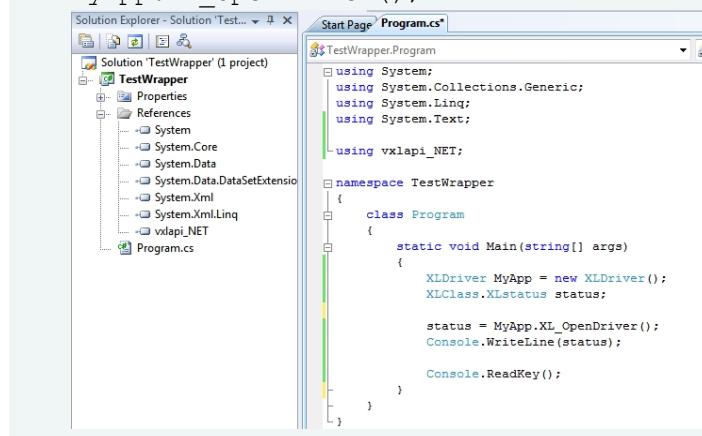
```
using vxlapi_.NET;
```

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**

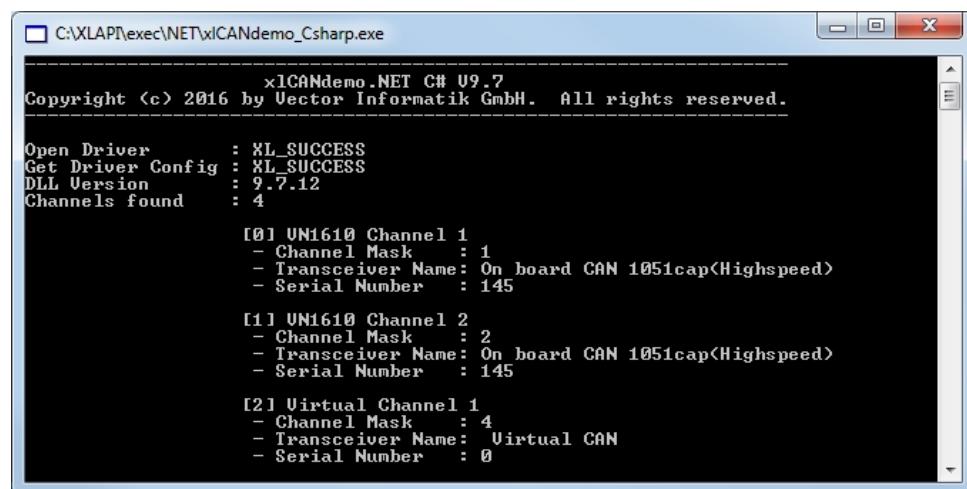
Take a look at our examples (source code) on the Vector Driver Disk for further information.

15.6 Application Examples

15.6.1 xICANdemo .NET

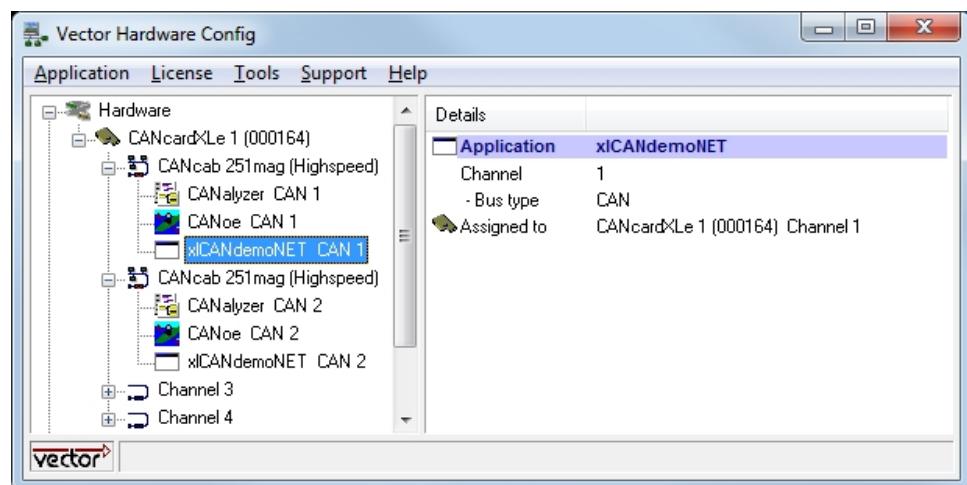
Description

This example shows how to access a Vector CAN interface.



Starting the example

When the example starts, it looks for the application **xICANdemoNET** in **Vector Hardware Config**. 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 CAN-cardXLe or the VN1630A. Both channels have also to be physically connected, e. g. via CANcable1.



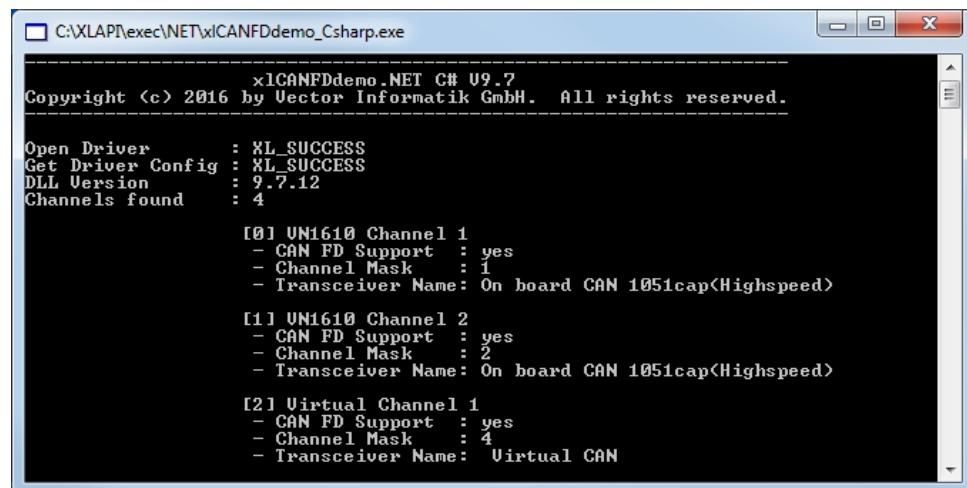
Send and receive messages

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.

15.6.2 xICANdemo .NET

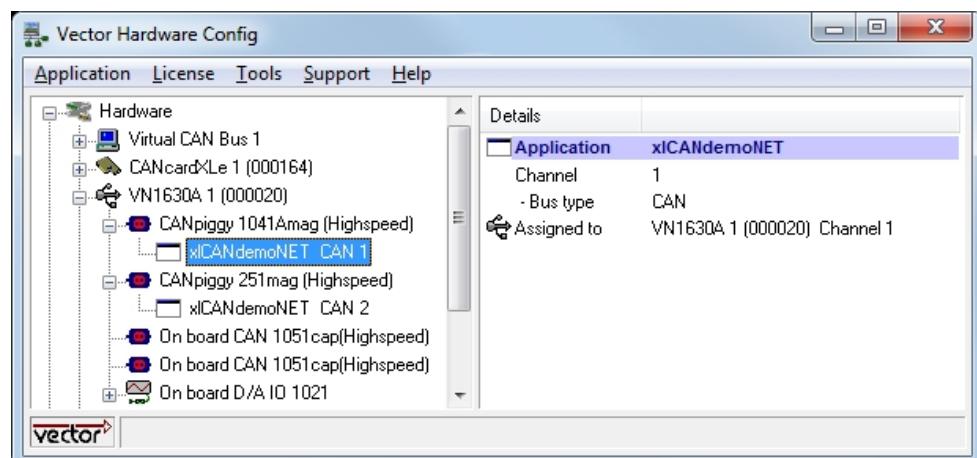
Description

This example shows how to access the XL API for CAN FD.



Starting the example

When the example starts, it looks for the application **xICANdemoNET** in **Vector Hardware Config**. 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.

15.6.3 xLINdemo .NET

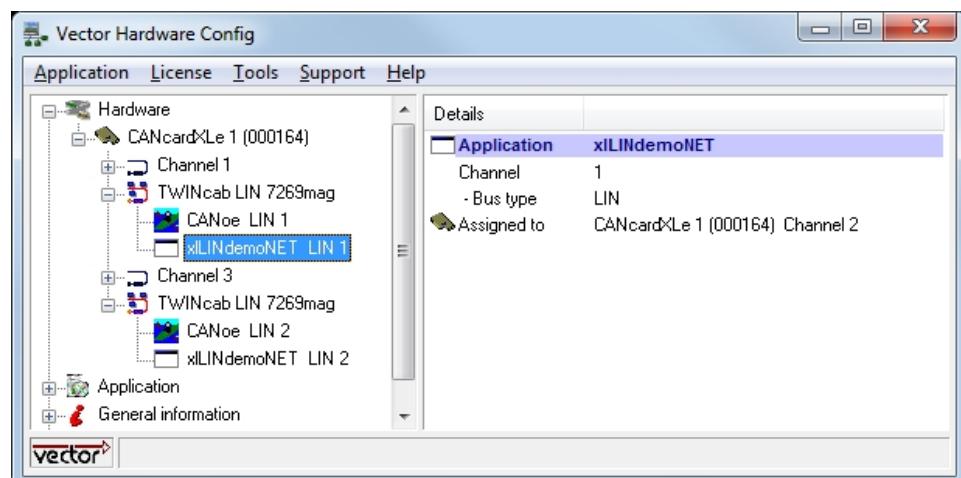
Description

This example shows how to access a Vector LIN interface.



Starting the example

When the example starts, it looks for the application **xLINdemoNET** in the Vector Hardware Config. 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 CAN-cardXLe or 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.

15.6.4 xLINdemo Single .NET

Description

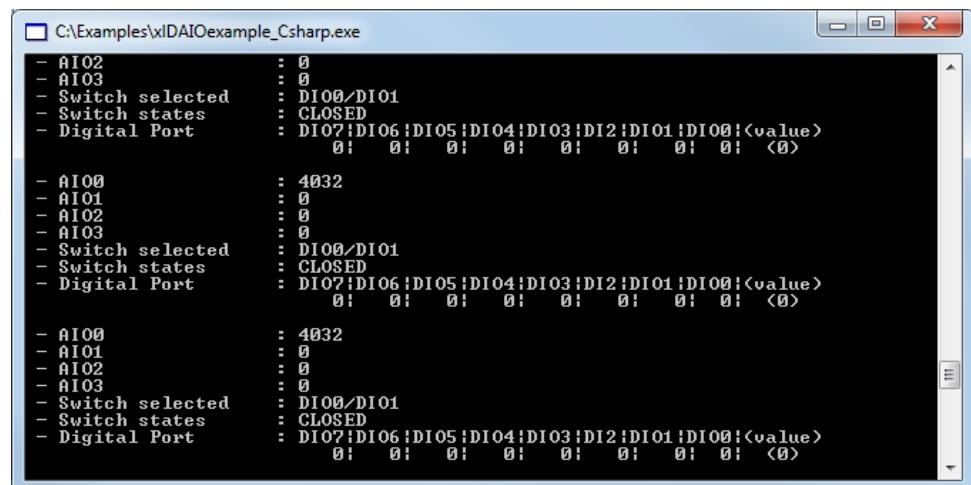
This example is similar to [xLINdemo .NET](#) on page 413, but uses only one LIN channel.

15.6.5 xIDAOexample .NET

15.6.5.1 General Information

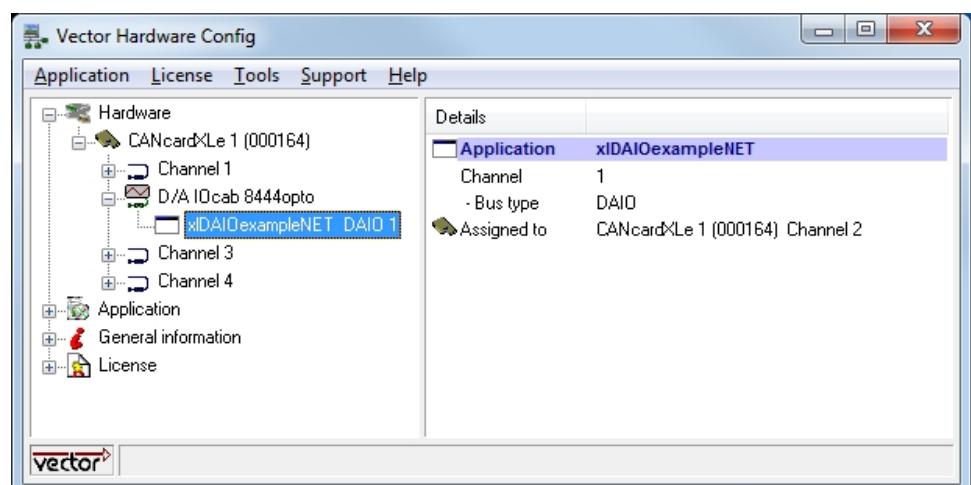
Description

This example demonstrates how to access a IOcab 8444opto for cyclical measurement.



Starting the example

When the example starts, it looks for the application **xIDAOexampleNET** in **Vector Hardware Config**. 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.



15.6.5.2 Setup

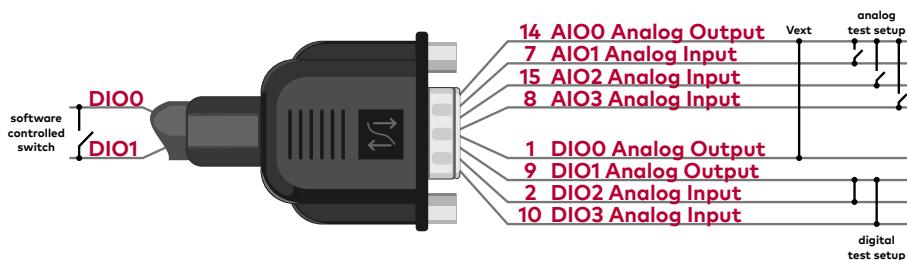
Pin definition

The following pins of the IOcab 8444opto are used in this example:

Signal	Pin	Type
AIO0	14	Analog output

Signal	Pin	Type
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 `x1DAIOSetDigitalOutput()`. If the switch is closed, the applied voltage at DIO0 can be measured at DIO1.

15.6.5.3 Keyboard commands

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

Key	Command
<ENTER>	Toggle digital output.
<x>	Closes application.

15.6.5.4 Output Examples

	Example
AIO0:	4032mV
AIO1:	0mV
AIO2:	0mV
AIO3:	0mV
Switch selected:	DIO0/DIO1
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 to 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/DIO1
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 to 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.)



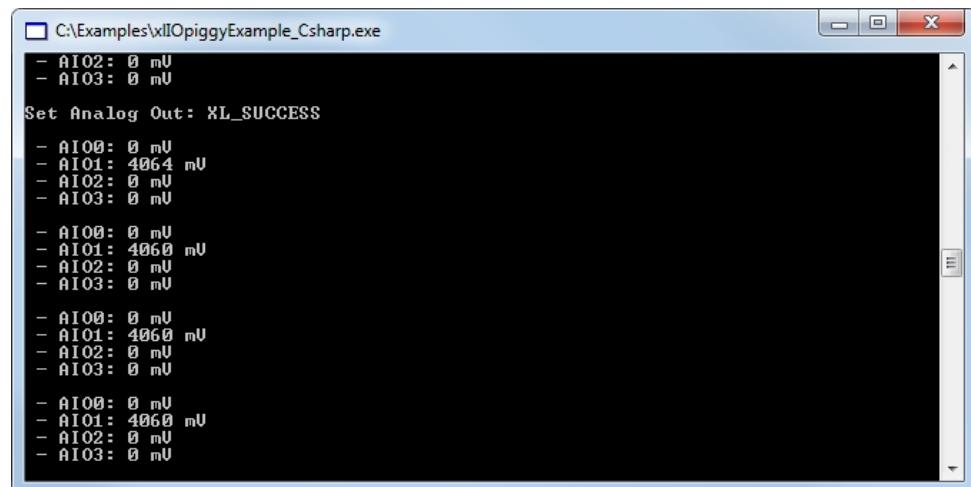
Note

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 4.096V. Due to different thresholds, the inputs DIO4...DIO7 needs higher voltages ($\geq 4.7V$) to toggle from '0' to '1'.

15.6.6 xIIOpiggyExample .NET

15.6.6.1 General Information

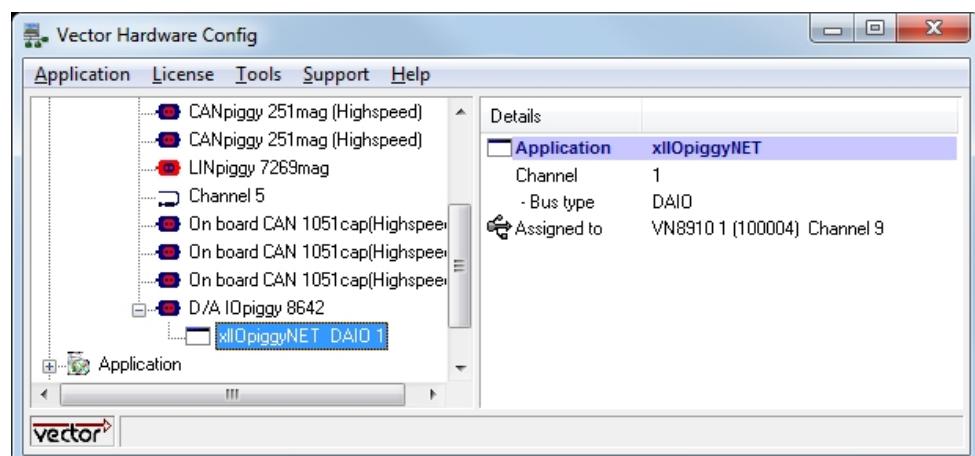
Description This example shows how to access the IOpiggy 8642 for analog measurement.s



```
C:\Examples\xIIOpiggyExample_Csharp.exe
- AIO2: 0 mV
- AIO3: 0 mV
Set Analog Out: XL_SUCCESS
- AIO0: 0 mV
- AIO1: 4064 mV
- AIO2: 0 mV
- AIO3: 0 mV
- AIO0: 0 mV
- AIO1: 4060 mV
- AIO2: 0 mV
- AIO3: 0 mV
- AIO0: 0 mV
- AIO1: 4060 mV
- AIO2: 0 mV
- AIO3: 0 mV
- AIO0: 0 mV
- AIO1: 4060 mV
- AIO2: 0 mV
- AIO3: 0 mV
```

Starting the example

When the example starts, it looks for the application **xIIOpiggyNET** in **Vector Hardware Config**. 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).



15.6.6.2 Setup

Pin definition

The following pins of the IOpiggy 8642 are used in this example:

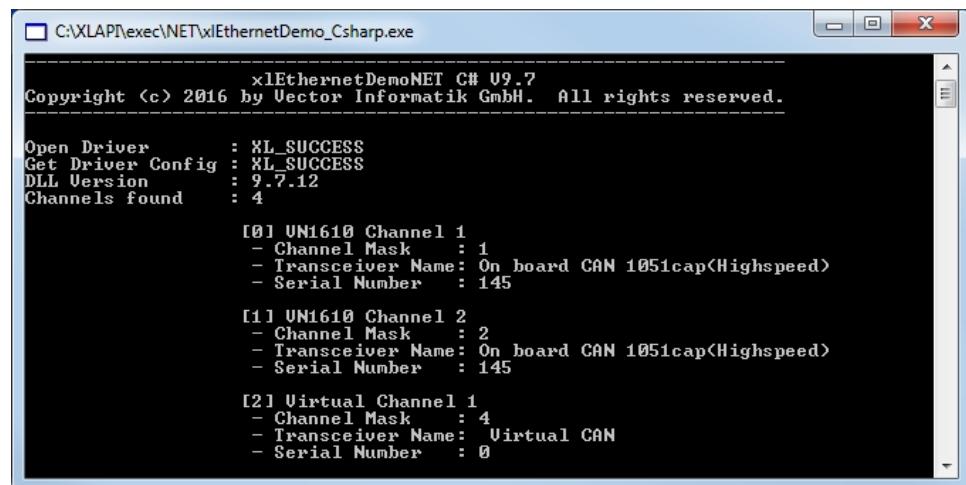
Signal	Pin	Type
AIO0	14	Analog output
AIO1	7	Analog input
AIO2	15	Analog input
AIO3	8	Analog input

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.

15.6.7 xlEthernetDemo .NET

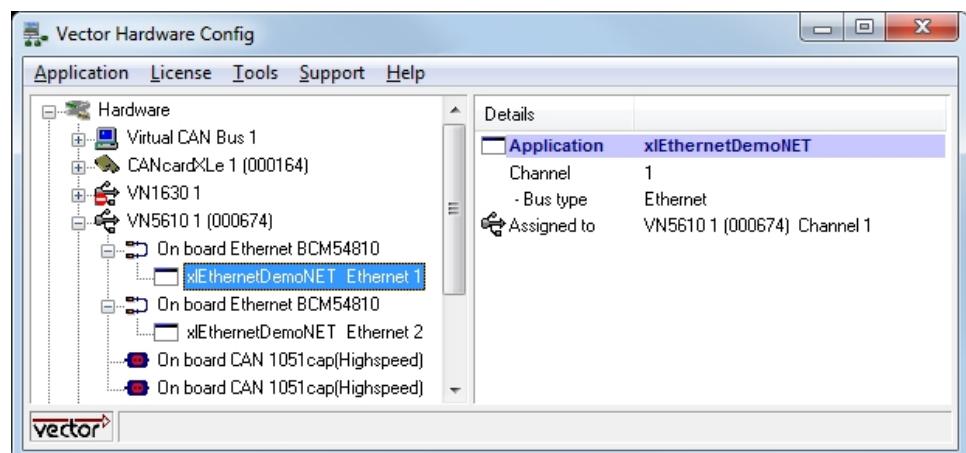
Description

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 **Vector Hardware Config**. 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).



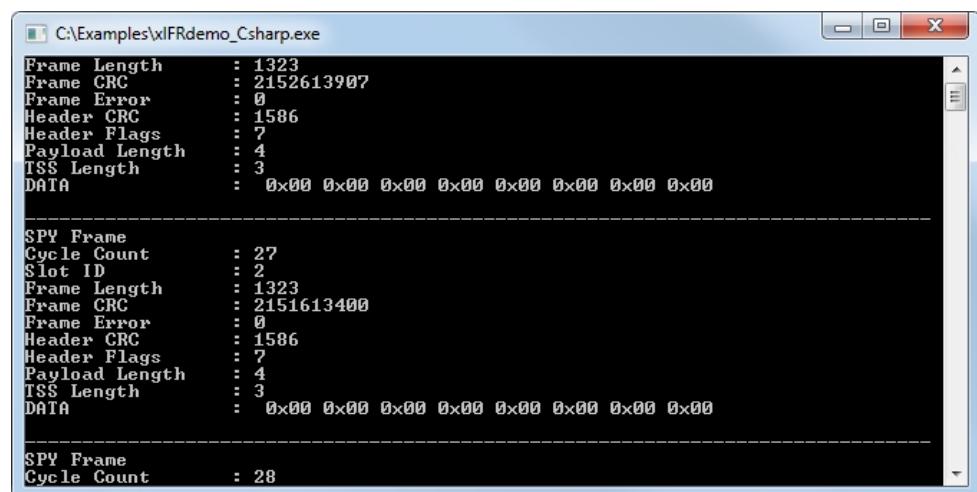
Send and receive messages

By pressing the **[ENTER]** key, the example sends and receives Ethernet frames. The message is sent over the first configured channel and is received by the second one.

15.6.8 4.8 xIFRdemo .NET

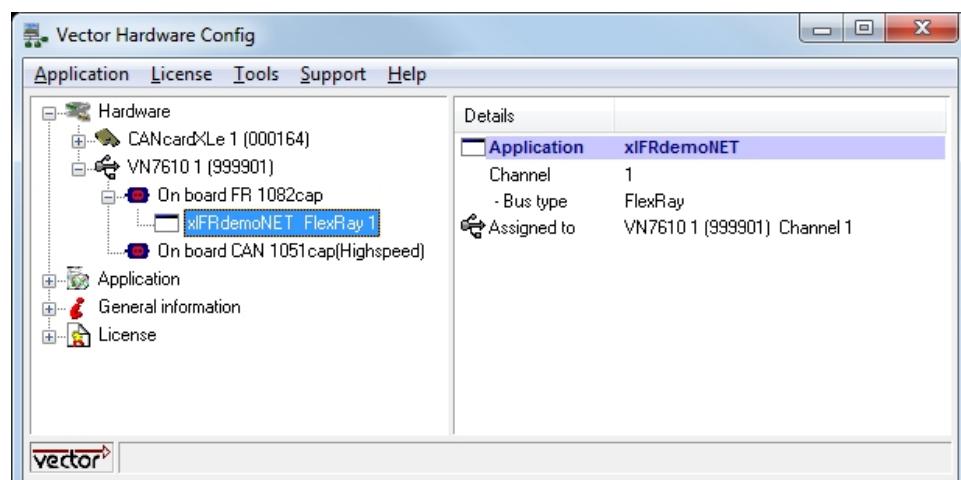
Description

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 **xIFRdemoNET** in **Vector Hardware Config**. 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 XL Status Error Codes	424
16.2 MOST150 Error Codes	426
16.3 Ethernet Error Codes	427

16.1 XL Status Error Codes

Code	Error	Description
0	XL_SUCCESS	The driver call was successful.
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	Only used in the MOST option to differ between the free- and 'MOST Analyses' library.
101	XL_ERR_WRONG_PARAMETER	At least one parameter passed to the driver was wrong or invalid.
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.
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.
158	XL_ERR_INIT_ACCESS_MISSING	Function call requires init access.
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.

Code	Error	Description
204	XL_ERR_INVALID_CHANNEL_MASK	Invalid channel mask.
205	XL_ERR_NOT_SUPPORTED	Function call not supported.
255	XL_ERROR	An unspecified error occurred.

16.2 MOST150 Error Codes

Code	Description
4224	Invalid parameter <code>deviceMode</code> set in <code>xIMost150SetDeviceMode()</code> .
4225	Invalid parameter <code>nodeAddress</code> set in <code>xIMost150SetSpecialNodeInfo()</code> .
4226	Invalid parameter <code>groupAddress</code> set in <code>xIMost150SetSpecialNodeInfo()</code> .
4227	Invalid parameter <code>sbc</code> set in <code>xIMost150SetSpecialNodeInfo()</code> .
4228	Invalid parameter <code>CtrlRetryTime</code> or <code>ctrlSendAttemps</code> or <code>asyncRetryTime</code> or <code>asyncSendAttemps</code> set in <code>xIMost150SetSpecialNodeInfo()</code> .
4234	Invalid parameter <code>device</code> set in <code>xIMost150CtrlSyncAudio()</code> , <code>xIMost150SyncSetXXX()</code> or <code>xIMost150SyncGetXXX()</code> .
4235	Invalid parameter <code>label</code> set in <code>xIMost150CtrlSyncAudio()</code> .
4236	Invalid parameter <code>width</code> set in <code>xIMost150CtrlSyncAudio()</code> .
4237	Invalid parameter <code>volume</code> set in <code>xIMost150SyncSetVolume()</code> .
4238	Invalid parameter <code>mute</code> set in <code>xIMost150SyncSetMute()</code> .
4239	Invalid parameter <code>mode</code> set in <code>xIMost150CtrlSyncAudio()</code> .
4240	Invalid parameter <code>sourceMask</code> set in <code>xIMost150SwitchEventSources()</code> .
4242	Invalid parameter <code>attenuation</code> set in <code>xIMost150SetTxLightPower()</code> .
4243	Invalid parameter <code>txLightset</code> in <code>xIMost150SetTxLight()</code> .
4244	Invalid parameter <code>requestMask</code> set in <code>xIMost150GetSpecialNodeInfo()</code> .
4245	Invalid parameter <code>frequency</code> set in <code>xIMost150SetFrequency()</code> .
4246	Invalid parameter <code>targetAddress</code> set in <code>xIMost150CtrlConfigureBusload()</code> or <code>xIMost150AsyncConfigureBusload()</code> .
4247	Invalid parameter <code>telLen</code> or <code>length</code> set in <code>xIMost150CtrlConfigureBusload()</code> or <code>xIMost150AsyncConfigureBusload()</code> .
4248	Invalid parameter <code>counterType</code> set in <code>xIMost150CtrlConfigureBusload()</code> or <code>xIMost150AsyncConfigureBusload()</code> .
4249	Invalid parameter <code>counterPosition</code> set in <code>xIMost150CtrlConfigureBusload()</code> or <code>xIMost150AsyncConfigureBusload()</code> .
4250	Invalid parameter <code>telLen</code> set in <code>xIMost150CtrlTransmit()</code> .
4251	Invalid parameter <code>length</code> set in <code>xIMost150AsyncTransmit()</code> .
4252	Invalid parameter <code>length</code> set in <code>xIMost150EthernetTransmit()</code> .
4253	Invalid parameter <code>busloadType</code> set in <code>xIMost150AsyncConfigureBusload()</code> .
4254	Invalid parameter <code>numBytesPerFrame</code> set in <code>xIMost150StreamOpen()</code> .
4255	Invalid parameter <code>latency</code> set in <code>xIMost150StreamOpen()</code> .
4256	Invalid parameter <code>direction</code> set in <code>xIMost150StreamOpen()</code> .
4257	Invalid parameter <code>streamHandle</code> set in <code>xIMost150StreamXXX()</code> .
4258	Invalid parameter <code>pConnLabels</code> set in <code>xIMost150StreamStart</code> (invalid CL).
4259	Invalid parameter <code>pConnLabels</code> set in <code>xIMost150StreamStart</code> (no CL provided).
4260	Invalid parameter <code>pConnLabels</code> set in <code>xIMost150StreamStart</code> (duplicate CL).
4261	Invalid stream state. Rx or Tx stream state does not allow the call of <code>xIMost150StreamXXX()</code> .
4262	Rx stream FIFO not initialized. This error can occur when calling <code>xIMost150StreamStart</code> without calling <code>xIMost150StreamInitRxFifo</code> before.
4263	Invalid parameter <code>bypassCloseTime</code> or <code>bypassOpenTime</code> set in <code>xIMost150GenerateBypassStress()</code> .
4264	Invalid parameter <code>numStates</code> or <code>pEclStates</code> or <code>pEclStatesDuration</code> set in

Code	Description
	xiMost150ECLConfigureSeq.
4265	ECL sequence contains too many entries set in xiMost150ECLConfigureSeq.

16.3 Ethernet Error Codes

Code	Error
0x1100	XL_ERR_ETH_PHY_ACTIVATION_FAILED
0x1101	XL_ERR_ETH_MAC_RESET_FAILED
0x1102	XL_ERR_ETH_MAC_NOT_READY
0x1103	XL_ERR_ETH_PHY_CONFIG_ABORTED
0x1104	XL_ERR_ETH_RESET_FAILED
0x1107	XL_ERR_ETH_MAC_ACTIVATION_FAILED



Get More Information

Visit our website for:

- > News
- > Products
- > Demo software
- > Support
- > Training classes
- > Addresses

www.vector.com