# Contents

## 1 Introduction

1.1 About this User Manual ........................................................................... 21
   1.1.1 Warranty ......................................................................................... 22
   1.1.2 Registered Trademarks .................................................................. 22
   1.1.3 Safety Instructions and Hazard Warnings ...................................... 22
       1.1.3.1 Proper Use and Intended Purpose .......................................... 23
       1.1.3.2 Hazards ................................................................................... 24
       1.1.3.3 Battery Notice ......................................................................... 24
       1.1.3.4 Warning of Hot Surface ........................................................... 24
       1.1.3.5 Electrostatically Sensitive Components (ESD) ..................... 25
   1.1.4 Disclaimer ....................................................................................... 25

## 2 Overview

2.1 General Information ................................................................................ 28
2.2 Principles of the XL Driver Library .......................................................... 29
   2.2.1 General Information ...................................................................... 29
   2.2.2 Step 1: Driver Initialization ......................................................... 30
   2.2.3 Step 2: Channel Setup .................................................................. 32
   2.2.4 Step 3: On Bus/Measurement Tasks ............................................. 32
2.3 Driver Files and Examples ....................................................................... 33
2.4 System Requirements ............................................................................... 34
2.5 Additional Information ............................................................................ 35
2.6 License Management ................................................................................ 36
   2.6.1 General Information ...................................................................... 36

## 3 Common Commands

3.1 Introduction .............................................................................................. 38
3.2 Functions ................................................................................................. 39
   3.2.1 xlOpenDriver ................................................................................. 39
   3.2.2 xlCloseDriver ................................................................................. 39
   3.2.3 xlGetApplConfig .......................................................................... 39
   3.2.4 xlSetApplConfig ........................................................................... 41
   3.2.5 xlGetDriverConfig ....................................................................... 41
   3.2.6 xlGetRemoteDriverConfig ............................................................. 42
   3.2.7 xlGetChannelIndex ....................................................................... 43
   3.2.8 xlGetChannelMask ......................................................................... 43
   3.2.9 xlOpenPort ..................................................................................... 44
   3.2.10 xlClosePort .................................................................................. 46
   3.2.11 xlSetTimerRate ............................................................................ 46
3.2.12 xlSetTimerRateAndChannel ................................................................. 47
3.2.13 xlResetClock ......................................................................................... 48
3.2.14 xlSetNotification ................................................................................. 48
3.2.15 xlFlushReceiveQueue ............................................................................ 49
3.2.16 xlGetReceiveQueueLevel ...................................................................... 49
3.2.17 xlActivateChannel ................................................................................ 50
3.2.18 xlReceive .............................................................................................. 51
3.2.19 xlGetEventString ................................................................................. 52
3.2.20 xlGetErrorString .................................................................................. 53
3.2.21 xlGetSyncTime ....................................................................................... 53
3.2.22 xlGetChannelTime ................................................................................. 54
3.2.23 xlGenerateSyncPulse .............................................................................. 54
3.2.24 xlPopupHwConfig ................................................................................ 55
3.2.25 xlDeactivateChannel ............................................................................ 55
3.2.26 xlGetLicenseInfo .................................................................................. 55
3.2.27 xlSetGlobalTimeSync .......................................................................... 56
3.2.28 xlGetKeymanBoxes .............................................................................. 56
3.2.29 xlGetKeymanInfo ................................................................................ 57
3.2.30 xlCreateDriverConfig ........................................................................ 58
3.2.31 xlDestroyDriverConfig ....................................................................... 59

3.3 Structures .................................................................................................. 61
3.3.1 XldriverConfig ....................................................................................... 61
3.3.2 XlchannelConfig .................................................................................... 61
3.3.3 XlbusParams ........................................................................................ 64
3.3.4 XllicenseInfo ........................................................................................ 68
3.3.5 XlapiIDriverConfigV1 .......................................................................... 69
3.3.6 XlchannelDrvConfigV1 ......................................................................... 70
3.3.7 XldeviceDrvConfigV1 ............................................................................ 72
3.3.8 XlnetworkDrvConfigV1 ......................................................................... 73
3.3.9 XlswitchDrvConfigV1 ............................................................................ 74
3.3.10 XlvirtualportDrvConfigV1 .................................................................. 75
3.3.11 XlmeasurementpointDrvConfigV1 ....................................................... 75
3.3.12 XldllDrvConfigV1 ................................................................................ 76

3.4 Events ......................................................................................................... 77
3.4.1 Xlevent .................................................................................................. 77
3.4.2 Xl Tag Data ........................................................................................... 78
3.4.3 Xl Sync Pulse ......................................................................................... 78
3.4.4 Xl Transceiver ....................................................................................... 79
3.4.5 Xl Timer ................................................................................................. 79

4 CAN Commands ............................................................................................. 80
4.1 Introduction ........................................................................................................... 81
4.2 Flowchart ............................................................................................................... 82
4.3 Functions ................................................................................................................. 83
  4.3.1 xlCanSetChannelMode ...................................................................................... 83
  4.3.2 xlCanSetChannelOutput .................................................................................. 83
  4.3.3 xlCanSetReceiveMode ..................................................................................... 84
  4.3.4 xlCanSetChannelTransceiver ......................................................................... 84
  4.3.5 xlCanSetChannelParams ................................................................................ 86
  4.3.6 xlCanSetChannelParamsC200 ....................................................................... 87
  4.3.7 xlCanSetChannelBitrate ................................................................................. 87
  4.3.8 xlCanSetChannelAcceptance ....................................................................... 88
  4.3.9 xlCanAddAcceptanceRange .......................................................................... 89
  4.3.10 xlCanRemoveAcceptanceRange .................................................................. 90
  4.3.11 xlCanResetAcceptance ............................................................................... 91
  4.3.12 xlCanRequestChipState ............................................................................. 92
  4.3.13 xlCanTransmit .......................................................................................... 92
  4.3.14 xlCanFlushTransmitQueue ......................................................................... 93
4.4 Structs .................................................................................................................... 94
  4.4.1 XLchipParams .......................................................................................... 94
4.5 Events ..................................................................................................................... 95
  4.5.1 XL CAN Message ...................................................................................... 95
  4.5.2 XL Chip State ........................................................................................ 96
4.6 Application Examples ........................................................................................... 98
  4.6.1 xlCANdemo ............................................................................................... 98
    4.6.1.1 General Information ........................................................................... 98
    4.6.1.2 Keyboard Commands ........................................................................... 98
    4.6.1.3 Functions ........................................................................................ 99
  4.6.2 xlCANcontrol .......................................................................................... 100
    4.6.2.1 General Information ............................................................................ 100
    4.6.2.2 Classes ........................................................................................ 101
    4.6.2.3 Functions ........................................................................................ 101

5 CAN FD Commands ............................................................................................. 103
5.1 Introduction .......................................................................................................... 104
5.2 Flowchart .............................................................................................................. 105
5.3 Functions .............................................................................................................. 106
  5.3.1 xlCanFdSetConfiguration ........................................................................... 106
  5.3.2 xlCanTransmitEx ...................................................................................... 106
  5.3.3 xlCanReceive ........................................................................................ 107
  5.3.4 xlCanGetEventString ............................................................................... 107
5.4 Structs .................................................................................................................. 108
6 Ethernet Commands .......................................................... 118

6.1 Introduction ................................................................. 119
6.1.1 General Information ..................................................... 119
6.1.2 Network-Based API vs. Channel-Based API ......................... 120
6.1.3 Device Support .......................................................... 123
6.1.4 Switching Access Mode ................................................ 123

6.2 Network Based Access Mode ............................................. 125
6.2.1 Basic Concept ........................................................... 125
6.2.2 Definitions ............................................................... 127
6.2.3 General Information ..................................................... 129
   6.2.3.1 Step 1: Driver Initialization ..................................... 130
   6.2.3.2 Step 2: Network Setup .......................................... 131
   6.2.3.3 Step 3: On Network/Measurement Tasks ....................... 132
6.2.4 Flowchart ............................................................... 133
6.2.5 Functions ............................................................... 134
   6.2.5.1 xlNetActivateNetwork ........................................... 134
   6.2.5.2 xlNetAddVirtualPort ............................................ 134
   6.2.5.3 xlNetCloseNetwork .............................................. 134
   6.2.5.4 xlNetConnectMeasurementPoint ................................. 135
   6.2.5.5 xlNetDeactivateNetwork ........................................ 135
   6.2.5.6 xlNetEthOpenNetwork ......................................... 135
   6.2.5.7 xlNetEthReceive ................................................. 136
   6.2.5.8 xlNetEthRequestChannelStatus ................................. 137
   6.2.5.9 xlNetEthSend ..................................................... 137
   6.2.5.10 xlNetOpenVirtualPort ....................................... 137
   6.2.5.11 xlNetReleaseMACAddress ..................................... 138
   6.2.5.12 xlNetRequestMACAddress ..................................... 138
   6.2.5.13 xlNetSetNotification ......................................... 139
6.2.6 Structs .................................................................. 140
   6.2.6.1 T_XL_ETH_MAC_ADDRESS .................................... 140

6.2.7 Events .................................................................. 141
   6.2.7.1 T_XL_NET_ETH_CHANNEL_STATUS ............................ 141
   6.2.7.2 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX ............ 141
   6.2.7.3 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX_ERROR .... 141
### 7.3 LIN Basics

7.3 Functions ........................................................................................................... 178

7.4 Functions ........................................................................................................... 179

7.4.1 xlLinSetChannelParams ............................................................................... 179
7.4.2 xlLinSetDLC ................................................................................................ 179
7.4.3 xlLinSetChecksum ....................................................................................... 180
7.4.4 xlLinSetSlave ............................................................................................... 181
7.4.5 xlLinSwitchSlave ......................................................................................... 182
7.4.6 xlLinSendRequest ......................................................................................... 183
7.4.7 xlLinWakeUp ............................................................................................... 183
7.4.8 xlLinSetSleepMode ...................................................................................... 183

7.5 Structs ............................................................................................................... 185

7.5.1 XLlinStatPar .............................................................................................. 185

7.6 Events ............................................................................................................... 186

7.6.1 XL LIN Message API .................................................................................. 186
7.6.2 XL LIN Message ........................................................................................ 186
7.6.3 XL LIN Error Message .............................................................................. 187
7.6.4 XL LIN Sync Error ...................................................................................... 187
7.6.5 LIN No Answer ........................................................................................... 187
7.6.6 LIN Wake Up .............................................................................................. 187
7.6.7 LIN Sleep .................................................................................................. 188
7.6.8 LIN CRC Info ............................................................................................. 188

7.7 Application Examples ...................................................................................... 189

7.7.1 xlLINEExample .......................................................................................... 189

8. K-Line Commands .............................................................................................. 191

8.1 Introduction ...................................................................................................... 192

8.2 Flowchart ......................................................................................................... 193

8.3 Functions ......................................................................................................... 194

8.3.1 xlKlineFastInitTester ................................................................................ 194
8.3.2 xlKlineInit5BdEcu ..................................................................................... 194
8.3.3 xlKlineInit5BdTester ................................................................................ 195
8.3.4 xlKlineSetBaudrate ................................................................................... 195
8.3.5 xlKlineSetCommunicationTimingEcu ....................................................... 196
8.3.6 xlKlineSetCommunicationTimingTester ............................................... 196
8.3.7 xlKlineSetUartParams .............................................................................. 196
8.3.8 xlKlineSwitchHighspeedMode ................................................................. 197
8.3.9 xlKlineSwitchTesterResistor ................................................................... 197
8.3.10 xlKlineTransmit ....................................................................................... 198
## 9.5.1.5 Functions

9.5.2 xlDAIOdemo

9.5.2.1 General Information

9.5.2.2 Classes

## 10 D/A IO Commands (IOpiggy)

### 10.1 Introduction

### 10.2 Flowchart

### 10.3 Functions

10.3.1 xlIoSetTriggerMode (IOpiggy)

10.3.2 xlIoConfigurePorts

10.3.3 xlIoSetDigInThreshold

10.3.4 xlIoSetDigOutLevel

10.3.5 xlIoSetDigitalOutput

10.3.6 xlIoSetAnalogOutput

10.3.7 xlIoStartSampling

### 10.4 Structs

10.4.1 XLdaioTriggerMode

10.4.2 XLdaioSetPort

10.4.3 XLdaioDigitalParams (IOpiggy)

10.4.4 XLdaioAnalogParams

### 10.5 Events

10.5.1 XL DAIO Piggy Data

10.5.2 XL IO Analog Data

10.5.3 XL IO Digital Data

## 11 D/A IO Commands (VN1600)

### 11.1 Introduction

### 11.2 Flowchart

### 11.3 Functions

11.3.1 xlIoSetTriggerMode (VN1600)

11.3.2 xlIoSetDigitalOutput

### 11.4 Structs

11.4.1 XLdaioDigitalParams (VN1600)

### 11.5 Events

11.5.1 XL DAIO Piggy Data

11.5.2 XL IO Analog Data

11.5.3 XL IO Digital Data
12 MOST Commands .......................................................... 247
12.1 Introduction .............................................................. 248
12.2 Flowchart ................................................................. 249
12.3 Specific OS8104 Registers ............................................ 251
12.4 Functions ............................................................... 252
  12.4.1 xlMostSwitchEventSources .................................. 252
  12.4.2 xlMostSetAllBypass ............................................. 253
  12.4.3 xlMostGetAllBypass ............................................ 254
  12.4.4 xlMostSetTimingMode ......................................... 254
  12.4.5 xlMostGetTimingMode ......................................... 255
  12.4.6 xlMostSetFrequency .......................................... 256
  12.4.7 xlMostGetFrequency .......................................... 256
  12.4.8 xlMostWriteRegister ......................................... 257
  12.4.9 xlMostReadRegister .......................................... 258
  12.4.10 xlMostWriteRegisterBit .................................... 258
  12.4.11 xlMostCtrlTransmit ......................................... 259
  12.4.12 xlMostAsyncTransmit ....................................... 260
  12.4.13 xlMostSyncGetAllocTable ................................ 260
  12.4.14 xlMostCtrlSyncAudio ....................................... 261
  12.4.15 xlMostCtrlSyncAudioEx .................................... 262
  12.4.16 xlMostSyncVolume .......................................... 263
  12.4.17 xlMostSyncGetVolumeStatus ................................ 264
  12.4.18 xlMostSyncMute ............................................. 264
  12.4.19 xlMostSyncGetMuteStatus .................................. 265
  12.4.20 xlMostGetRxLight .......................................... 266
  12.4.21 xlMostSetTxLight ........................................... 266
  12.4.22 xlMostGetTxLight .......................................... 267
  12.4.23 xlMostSetLightPower ....................................... 267
  12.4.24 xlMostGetLockStatus ...................................... 268
  12.4.25 xlMostGenerateLightError .................................. 268
  12.4.26 xlMostGenerateLockError ................................... 269
  12.4.27 xlMostCtrlRxBuffer ........................................ 270
  12.4.28 xlMostCtrlConfigureBusload ................................ 271
  12.4.29 xlMostCtrlGenerateBusload .................................. 271
  12.4.30 xlMostAsyncConfigureBusload ............................. 272
  12.4.31 xlMostAsyncGenerateBusload ................................ 273
  12.4.32 xlMostReceive ............................................... 273
  12.4.33 xlMostTwinklePowerLed .................................... 274
  12.4.34 Streaming .................................................... 275
    12.4.34.1 General Information .................................. 275
    12.4.34.2 Frame Format .......................................... 276
12.4.35 xlMostStreamOpen ................................................................. 277
12.4.36 xlMostStreamClose .............................................................. 278
12.4.37 xlMostStreamStart ............................................................... 278
12.4.38 xlMostStreamStop ............................................................... 279
12.4.39 xlMostStreamBufferAllocate ............................................... 279
12.4.40 xlMostStreamBufferDeallocateAll ....................................... 280
12.4.41 xlMostStreamBufferSetNext ............................................... 281
12.4.42 xlMostStreamClearBuffers .................................................. 281
12.4.43 xlMostStreamGetInfo ........................................................... 282

12.5 Structs .................................................................................. 284
12.5.1 s_xl_most_async_busload_configuration .................................. 284
12.5.2 s_xl_most_ctrl_busload_configuration .................................... 284
12.5.3 XL_MOST_STREAM_OPEN ....................................................... 285

12.6 Events .................................................................................. 287
12.6.1 s_xl_event_most ................................................................... 287
12.6.2 s_xl_most_tag_data ............................................................... 288
12.6.3 XL_MOST_START ................................................................. 289
12.6.4 XL_MOST_STOP ................................................................. 290
12.6.5 XL_MOST_EVENT_SOURCE_EV ........................................... 290
12.6.6 XL_MOST_ALLBYPASS_EV ................................................... 290
12.6.7 XL_MOST_TIMING_MODE_EV ............................................... 290
12.6.8 XL_MOST_TIMING_MODE_SPDIF_EV ...................................... 291
12.6.9 XL_MOST_FREQUENCY_EV ................................................... 291
12.6.10 XL_MOST_REGISTER_BYTES ................................................. 291
12.6.11 XL_MOST_REGISTER_BITS_EV ............................................. 292
12.6.12 XL_MOST_SPECIAL_REGISTER_EV ...................................... 292
12.6.13 XL_MOST_CTRL_SPY_EV .................................................... 294
12.6.14 XL_MOST_CTRL_MSG_EV .................................................... 295
12.6.15 XL_MOST_CTRL_TX ............................................................ 297
12.6.16 XL_MOST_ASYNC_MSG_EV .................................................. 297
12.6.17 XL_MOST_ASYNC_TX_EV ..................................................... 297
12.6.18 XL_MOST_SYNC_ALLOC_EV .................................................. 298
12.6.19 XL_MOST_SYNC_VOLUME_STATUS_EV .................................. 298
12.6.20 XL_MOST_RX_LIGHT_EV ..................................................... 299
12.6.21 XL_MOST_TX_LIGHT_EV ..................................................... 299
12.6.22 XL_MOST_LOCK_STATUS_EV ................................................ 299
12.6.23 XL_MOST_ERROR_EV .......................................................... 300
12.6.24 XL_MOST_RX_BUFFER_EV ................................................... 300
12.6.25 XL_MOST_CTRL_SYNC_AUDIO_EV ....................................... 301
12.6.26 XL_MOST_CTRL_SYNC_AUDIO_EX ....................................... 301
12.6.27 XL_MOST_SYNC_MUTES_STATUS_EV ..................................... 302
12.6.28 XL_MOST_LIGHT_POWER_EV ................................................................. 302
12.6.29 XL_MOST_GEN_LIGHT_ERROR_EV ....................................................... 302
12.6.30 XL_MOST_GEN_LOCK_ERROR_EV .......................................................... 303
12.6.31 XL_MOST_CTRL_BUSLOAD_EV ............................................................. 303
12.6.32 XL_MOST_ASYNC_BUSLOAD_EV .......................................................... 304
12.6.33 XL_MOST_STREAM_BUFFER ................................................................. 304
12.6.34 XL_MOST_STREAM_STATE_EV ............................................................. 305
12.6.35 XL_MOST_SYNC_TX_UNDERFLOW_EV .................................................. 305
12.6.36 XL_MOST_SYNC_RX_OVERFLOW_EV ..................................................... 306

12.7 Application Examples .............................................................................. 307
  12.7.1 xlMOSTView ......................................................................................... 307
    12.7.1.1 General Information .................................................................... 307
    12.7.1.2 Classes ......................................................................................... 308
    12.7.1.3 Functions ...................................................................................... 308

13 MOST 150 Commands .................................................................................... 311
  13.1 Introduction .............................................................................................. 312
  13.2 Flowchart ................................................................................................ 313
  13.3 Functions .................................................................................................. 315
    13.3.1 xlMost150SwitchEventSources ......................................................... 315
    13.3.2 xlMost150SetDeviceMode ................................................................. 316
    13.3.3 xlMost150GetDeviceMode ................................................................. 317
    13.3.4 xlMost150SetSPDIFMode .................................................................. 318
    13.3.5 xlMost150GetSPDIFMode ................................................................. 318
    13.3.6 xlMost150SetSpecialNodeInfo ......................................................... 319
    13.3.7 xlMost150GetSpecialNodeInfo ......................................................... 319
    13.3.8 xlMost150SetFrequency ................................................................. 320
    13.3.9 xlMost150GetFrequency ................................................................. 321
    13.3.10 xlMost150GetSystemLockFlag ....................................................... 322
    13.3.11 xlMost150GetShutdownFlag ............................................................ 322
    13.3.12 xlMost150Shutdown ......................................................................... 323
    13.3.13 xlMost150Startup ........................................................................... 323
    13.3.14 xlMost150SetSSOResult .................................................................. 324
    13.3.15 xlMost150GetSSOResult .................................................................. 324
    13.3.16 xlMost150CtrlTransmit ................................................................. 325
    13.3.17 xlMost150AsyncTransmit ............................................................... 325
    13.3.18 xlMost150EthernetTransmit ............................................................ 326
    13.3.19 xlMost150SyncGetAllocTable ......................................................... 326
    13.3.20 xlMost150CtrlSyncAudio ............................................................... 327
    13.3.21 xlMost150SyncSetVolume ............................................................... 328
    13.3.22 xlMost150SyncGetVolume ............................................................. 328
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlMost150SyncSetMute</td>
<td>329</td>
</tr>
<tr>
<td>xlMost150SyncGetMute</td>
<td>329</td>
</tr>
<tr>
<td>xlMost150GetRxLightLockStatus</td>
<td>330</td>
</tr>
<tr>
<td>xlMost150SetTxLight</td>
<td>331</td>
</tr>
<tr>
<td>xlMost150GetTxLight</td>
<td>331</td>
</tr>
<tr>
<td>xlMost150SetTxLightPower</td>
<td>332</td>
</tr>
<tr>
<td>xlMost150GenerateLightError</td>
<td>332</td>
</tr>
<tr>
<td>xlMost150GenerateLockError</td>
<td>333</td>
</tr>
<tr>
<td>xlMost150CtrlConfigureBusload</td>
<td>334</td>
</tr>
<tr>
<td>xlMost150CtrlGenerateBusload</td>
<td>335</td>
</tr>
<tr>
<td>xlMost150AsyncConfigureBusload</td>
<td>335</td>
</tr>
<tr>
<td>xlMost150AsyncGenerateBusload</td>
<td>336</td>
</tr>
<tr>
<td>xlMost150ConfigureRxBuffer</td>
<td>337</td>
</tr>
<tr>
<td>xlMost150GenerateBypassStress</td>
<td>337</td>
</tr>
<tr>
<td>xlMost150SetECLLine</td>
<td>338</td>
</tr>
<tr>
<td>xlMost150SetECLTermination</td>
<td>339</td>
</tr>
<tr>
<td>xlMost150GetECLInfo</td>
<td>339</td>
</tr>
<tr>
<td>xlMost150ECLConfigureSeq</td>
<td>340</td>
</tr>
<tr>
<td>xlMost150ECLGenerateSeq</td>
<td>341</td>
</tr>
<tr>
<td>xlMost150SetECLGlitchFilter</td>
<td>341</td>
</tr>
<tr>
<td>Streaming</td>
<td>342</td>
</tr>
<tr>
<td>xlMost150StreamOpen</td>
<td>343</td>
</tr>
<tr>
<td>xlMost150StreamClose</td>
<td>344</td>
</tr>
<tr>
<td>xlMost150StreamStart</td>
<td>344</td>
</tr>
<tr>
<td>xlMost150StreamStop</td>
<td>347</td>
</tr>
<tr>
<td>xlMost150StreamTransmitData</td>
<td>347</td>
</tr>
<tr>
<td>xlMost150StreamClearTxFifo</td>
<td>348</td>
</tr>
<tr>
<td>xlMost150StreamInitRxFifo</td>
<td>349</td>
</tr>
<tr>
<td>xlMost150StreamReceiveData</td>
<td>349</td>
</tr>
<tr>
<td>xlMost150StreamGetInfo</td>
<td>350</td>
</tr>
<tr>
<td>xlMost150Receive</td>
<td>351</td>
</tr>
<tr>
<td>xlMost150TwinklePowerLed</td>
<td>351</td>
</tr>
<tr>
<td>Structs</td>
<td>352</td>
</tr>
<tr>
<td>XLmost150AsyncBusloadConfig</td>
<td>352</td>
</tr>
<tr>
<td>XLmost150AsyncTxMsg</td>
<td>352</td>
</tr>
<tr>
<td>XLmost150CtrlBusloadConfig</td>
<td>353</td>
</tr>
<tr>
<td>XLmost150CtrlTxMsg</td>
<td>354</td>
</tr>
<tr>
<td>XLmost150EthernetTxMsg</td>
<td>354</td>
</tr>
<tr>
<td>XLmost150SetSpecialNodeInfo</td>
<td>355</td>
</tr>
<tr>
<td>XLmost150StreamInfo</td>
<td>356</td>
</tr>
</tbody>
</table>
13.4.8 XLmost150StreamOpen .......................................................... 356
13.4.9 XLmost150SyncAudioParameter ........................................... 357

13.5 Events .................................................................................. 359
13.5.1 XLmost150event ................................................................. 359
13.5.2 XLmost150AsyncBusloadConfig ........................................... 360
13.5.3 XLmost150AsyncTxMsg ....................................................... 361
13.5.4 XLmost150EthernetTxMsg ................................................... 362
13.5.5 XL_START ........................................................................ 362
13.5.6 XL_STOP ........................................................................... 362
13.5.7 XL_MOST150_EVENT_SOURCE_EV ....................................... 363
13.5.8 XL_MOST150_DEVICE_MODE_EV .......................................... 363
13.5.9 XL_MOST150_SPDIFF_MODE_EV .......................................... 364
13.5.10 XL_MOST150_FREQUENCY_EV ........................................... 364
13.5.11 XL_MOST150_SPECIAL_NODE_INFO_EV ............................... 364
13.5.12 XL_MOST150_CTRL_SPY_EV .............................................. 366
13.5.13 XL_MOST150_CTRL_RX_EV .............................................. 368
13.5.14 XL_MOST150_CTRL_TX_ACK_EV ......................................... 368
13.5.15 XL_MOST150_ASYNC_SPY_EV ............................................ 369
13.5.16 XL_MOST150_ASYNC_RX_EV ............................................ 371
13.5.17 XL_MOST150_ASYNC_TX_ACK_EV ....................................... 372
13.5.18 XL_MOST150_CL_INFO ....................................................... 372
13.5.19 XL_MOST150_SYNC_ALLOC_INFO_EV .................................. 373
13.5.20 XL_MOST150_TX_LIGHT_EV ............................................. 373
13.5.21 XL_MOST150_RX_LIGHT_LOCKSTATUS_EV ............................. 373
13.5.22 XL_MOST150_ERROR_EV .................................................. 374
13.5.23 XL_MOST150_CTRL_SYNC_AUDIO_EV ................................... 374
13.5.24 XL_MOST150_SYNC_VOLUME_STATUS_EV ............................. 376
13.5.25 XL_MOST150_SYNC_MUTE_STATUS_EV ................................. 376
13.5.26 XL_MOST150_LIGHT_POWER_EV ......................................... 376
13.5.27 XL_MOST150_GEN_LIGHT_ERROR_EV ................................... 377
13.5.28 XL_MOST150_GEN_LOCK_ERROR_EV ..................................... 377
13.5.29 XL_MOST150_CONFIGURE_RX_BUFFER_EV ............................. 377
13.5.30 XL_MOST150_CTRL_BUSLOAD_EV ....................................... 378
13.5.31 XL_MOST150_ASYNC_BUSLOAD_EV ..................................... 378
13.5.32 XL_MOST150ETHERNET_SPY_EV ......................................... 379
13.5.33 XL_MOST150ETHERNET_RX_EV .......................................... 380
13.5.34 XL_MOST150ETHERNET_TX_ACK_EV ..................................... 381
13.5.35 XL_MOST150_SYSTEMLOCK_FLAG_EV ................................... 382
13.5.36 XL_MOST150_SHUTDOWN_FLAG_EV ..................................... 382
13.5.37 XL_MOST150_NW_STARTUP_EV .......................................... 382
13.5.38 XL_MOST150_NW_SHUTDOWN_EV ........................................ 383
14 FlexRay Commands ................................................. 395
14.1 Introduction ......................................................... 396
14.2 Flowchart ........................................................... 397
14.3 Free Library and Advanced Library ................................. 398
14.4 FlexRay Basics ...................................................... 399
  14.4.1 Introduction ................................................. 399
  14.4.2 Data Transmission Requirements ......................... 399
  14.4.3 FlexRay Communication Architecture ..................... 400
  14.4.4 Deterministic and Dynamic .............................. 402
  14.4.5 CRC-Protected Data Transmission ....................... 404
14.5 Functions ......................................................... 405
  14.5.1 xlFrSetConfiguration ...................................... 405
  14.5.2 xlFrGetChannelConfiguration ......................... 405
  14.5.3 xlFrSetMode .............................................. 406
  14.5.4 xlFrInitStartupAndSync .................................. 406
  14.5.5 xlFrSetupSymbolWindow ................................. 407
  14.5.6 xlFrActivateSpy ........................................ 407
  14.5.7 xlSetTimerBaseNotify .................................... 408
  14.5.8 xlFrReceive ............................................. 408
  14.5.9 xlFrTransmit ............................................ 409
  14.5.10 xlFrSetTransceiverMode .................................. 409
  14.5.11 xlFrSendSymbolWindow .................................. 410
  14.5.12 xlFrSetAcceptanceFilter ................................ 410
14.6 Structs ............................................................ 412
14.6.1 XLfrClusterConfig ................................................................. 412
14.6.2 XLfrChannelConfig ............................................................. 417
14.6.3 XLfrMode ............................................................................. 417
14.6.4 XLfrAcceptanceFilter ............................................................ 418

14.7 Events ................................................................................. 420
14.7.1 XLfrEvent ............................................................................. 420
14.7.2 XL_FR_START_CYCLE_EV ....................................................... 421
14.7.3 XL_FR_RX_FRAME_EV ........................................................... 422
14.7.4 XL_FR_TX_FRAME_EV ........................................................... 424
14.7.5 XL_FR_TXACK_FRAME ............................................................ 425
14.7.6 XL_FR_INVALID_FRAME .......................................................... 425
14.7.7 XL_FR_WAKEUP_EV ............................................................... 425
14.7.8 XL_FR_SYMBOL_WINDOW_EV .................................................. 426
14.7.9 XL_FR_ERROR_EV ................................................................ 427
14.7.10 XL_FR_ERROR_POC_MODE_EV ............................................. 427
14.7.11 XL_FR_ERROR_SYNC_FRAMES_BELOWMIN .................................. 428
14.7.12 XL_FR_ERROR_SYNC_FRAMES_EV ......................................... 428
14.7.13 XL_FR_ERROR_CLOCK_CORR_FAILURE_EV ............................. 428
14.7.14 XL_FR_ERROR_NIT_FAILURE_EV ............................................ 429
14.7.15 XL_FR_ERROR_CC_ERROR_EV ................................................ 429
14.7.16 XL_FR_STATUS_EV .............................................................. 430
14.7.17 XL_FR_NMVECTOR_EV .......................................................... 431
14.7.18 XL_FR_SPY_FRAME_EV ........................................................ 432
14.7.19 XL_FR_SPY_SYMBOL_EV ....................................................... 432
14.7.20 XL_APPLICATION_NOTIFICATION_EV .................................... 433

14.8 Application Examples ............................................................ 434
14.8.1 xlFlexDemo ................................................................. 434
14.8.1.1 General Information ........................................................ 434
14.8.1.2 Classes ........................................................................... 434
14.8.1.3 Functions ........................................................................ 434
14.8.1.4 Events ............................................................................ 435
14.8.2 xlFlexDemoCmdLine ............................................................ 436
14.8.2.1 General Information ........................................................ 436
14.8.2.2 Functions ........................................................................ 436
14.8.3 Fibex2CSharpReaderDemo .................................................... 437
14.8.3.1 General Information ........................................................ 437
14.8.3.2 Classes ............................................................................ 437

15 ARINC 429 Commands .............................................................. 438
15.1 Introduction ................................................................. 439
15.2 Flowchart ............................................................... 440
15.3 Functions ........................................................................... 441
15.3.1 xlA429SetChannelParams ........................................ 441
15.3.2 xlA429Transmit .................................................... 442
15.3.3 xlA429Receive ..................................................... 444

15.4 Structs .................................................................. 446
15.4.1 XL_A429_PARAMS ............................................... 446
15.4.2 XL_A429_MSG_TX ................................................. 451

15.5 Events .................................................................. 454
15.5.1 XLa429Event ......................................................... 454
15.5.2 XL_A429_EV_TX_OK ............................................. 455
15.5.3 XL_A429_EV_TAG_TX_ERR ................................. 456
15.5.4 XL_A429_EV_TAG_RX_OK .................................... 457
15.5.5 XL_A429_EV_TAG_RX_ERR ................................... 458
15.5.6 XL_A429_EV_BUS_STATISTIC ............................... 460

15.6 Application Examples .............................................. 462
15.6.1 xlA429Control ....................................................... 462
15.6.1.1 General Information ......................................... 462

16 .NET Wrapper .......................................................... 463
16.1 Overview ............................................................... 464
16.2 XLDriver - Accessing Driver ................................... 465
16.3 XLClass - Storing Data/Parameters ......................... 466
16.4 XLDefine - Using Predefined Values ....................... 467
16.5 Including the Wrapper in a New .NET Project ........ 468
16.6 Application Examples .............................................. 470
16.6.1 xlCANdemo .NET .................................................. 470
16.6.2 xlCANdemo .NET .................................................. 471
16.6.3 xlLINdemo .NET .................................................. 472
16.6.4 xlLINdemo Single .NET ......................................... 473
16.6.5 xlDAIOexample .NET ........................................... 474
16.6.5.1 General Information ....................................... 474
16.6.5.2 Setup ............................................................ 474
16.6.5.3 Keyboard commands ....................................... 475
16.6.5.4 Output Examples ........................................... 476
16.6.6 xlIOpiggyExample .NET ........................................ 478
16.6.6.1 General Information ....................................... 478
16.6.6.2 Setup ............................................................ 478
16.6.7 xlEthernetDemo .NET ........................................... 480
16.6.8 xIFRdemo .NET .................................................... 481

17 Error Codes ............................................................... 482
# 1 Introduction

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>1.1 About this User Manual</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1 Warranty</td>
<td>22</td>
</tr>
<tr>
<td>1.1.2 Registered Trademarks</td>
<td>22</td>
</tr>
<tr>
<td>1.1.3 Safety Instructions and Hazard Warnings</td>
<td>22</td>
</tr>
<tr>
<td>1.1.4 Disclaimer</td>
<td>25</td>
</tr>
</tbody>
</table>
1.1 About this User Manual

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

<table>
<thead>
<tr>
<th>Style</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bold</strong></td>
<td>Blocks, surface elements, window- and dialog names of the software. Accentuation of warnings and advices.</td>
</tr>
<tr>
<td>[OK]</td>
<td>Push buttons in brackets</td>
</tr>
<tr>
<td>File</td>
<td>Save</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>File name and source code.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hyperlink</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperlinks and references.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;CTRL&gt;+&lt;S&gt;</th>
<th>Notation for shortcuts.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>This symbol calls your attention to warnings.</td>
</tr>
<tr>
<td>i</td>
<td>Here you can obtain supplemental information.</td>
</tr>
<tr>
<td>→</td>
<td>Here you can find additional information.</td>
</tr>
<tr>
<td>↑</td>
<td>Here is an example that has been prepared for you.</td>
</tr>
<tr>
<td>⚠️</td>
<td>Step-by-step instructions provide assistance at these points.</td>
</tr>
<tr>
<td>✍️</td>
<td>Instructions on editing files are found at these points.</td>
</tr>
<tr>
<td>✘️</td>
<td>This symbol warns you not to edit the specified file.</td>
</tr>
</tbody>
</table>
1.1.1 Warranty

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

1.1.2 Registered Trademarks

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.

1.1.3 Safety Instructions and Hazard Warnings

Caution!
In order to avoid personal injuries and damage to property, you have to read and understand the following safety instructions and hazard warnings prior to installation and use of this hardware. Keep this documentation (manual) always near the hardware.

Caution!
Do not operate the device without antennas! To avoid physical damage to the device, please attach the provided antennas to the device before operation!
1.1.3.1 Proper Use and Intended Purpose

**Caution!**
The hardware is designed for analyzing, controlling and otherwise influencing control systems and electronic control units. This includes bus systems like CAN, FlexRay, Ethernet and/or BroadR-Reach.

The hardware may only be operated (i) according to the instructions and descriptions of this manual; (ii) with the electric power supply designed for the hardware, e.g. USB-powered power supply; and (iii) with accessories manufactured or approved by Vector.

The hardware is exclusively designed for use by skilled personnel as its operation may result in serious personal injuries and damage to property. Therefore, only those persons may operate the hardware who (i) have understood the possible effects of the actions which may be caused by the hardware; (ii) are specifically trained in the handling with the hardware, bus systems and the system intended to be influenced; and (iii) have sufficient experience in using the hardware safely.

The knowledge necessary for the operation of the hardware can be acquired in workshops and internal or external seminars offered by Vector. Additional and hardware specific information, such as „Known Issues“, are available in the „Vector KnowledgeBase“ on Vector’s website at www.vector.com. Please consult the „Vector KnowledgeBase“ for updated information prior to the operation of the hardware.

**Caution!**
The loggers are measuring devices which are mainly used in the automotive and commercial vehicles industries. The loggers are designed for gathering and recording data of the bus communication, for analyzing and possibly controlling electronic control units. This includes, inter alia, bus systems like CAN, LIN, MOST and FlexRay.

The loggers may only be operated in a closed state. In particular, printed circuits must not be visible. The loggers may only be operated according to the instructions and descriptions of this manual. Only suitable accessories should be used, such as the original Vector accessories or accessories approved by Vector.

The loggers are exclusively designed for use by skilled personnel as its operation may result in serious personal injuries and damage to property. Therefore, only those persons may operate the loggers who (i) have understood the possible effects of the actions which may be caused by the loggers; (ii) are specifically trained in the handling with the loggers, bus systems and the system intended to be influenced; and (iii) have sufficient experience in using the loggers safely.

The logger specific information can be acquired via the specific manuals as well as from the Vector KnowledgeBase at www.vector.com. Please consult the Vector KnowledgeBase for updated information prior to the operation of the loggers. The knowledge necessary for the bus systems used, can be acquired in workshops and internal or external seminars offered by Vector.
1.1.3.2 Hazards

Caution!
The hardware may control and/or otherwise influence the behavior of control systems and electronic control units. Serious hazards for life, body and property may arise, in particular, without limitation, by interventions in safety relevant systems (e.g. by deactivating or otherwise manipulating the engine management, steering, airbag and/or braking system) and/or if the hardware is operated in public areas (e.g. public traffic, airspace). Therefore, you must always ensure that the hardware is used in a safe manner. This includes, inter alia, the ability to put the system in which the hardware is used into a safe state at any time (e.g. by „emergency shutdown“), in particular, without limitation, in the event of errors or hazards.

Comply with all safety standards and public regulations which are relevant for the operation of the system. Before you operate the system in public areas, it should be tested on a site which is not accessible to the public and specifically prepared for performing test drives in order to reduce hazards.

Caution!
The loggers may control and/or otherwise influence the behavior of electronic control units. Serious hazards for life, body and property may arise, in particular, without limitation, by interventions in safety relevant systems (e.g. by deactivating or otherwise manipulating the engine management, steering, airbag and/or braking system) and/or if the loggers are operated in public areas (e.g. public traffic, airspace). Therefore, you must always ensure that the loggers are used in a safe manner. This includes, inter alia, the ability to put the system in which the loggers are used into a safe state at any time (e.g. by „emergency shutdown“), in particular, without limitation, in the event of errors or hazards.

Comply with all safety standards and public regulations which are relevant for the operation of the system. Before you operate the system in public areas, it should be tested on a site which is not accessible to the public and specifically prepared for performing test drives in order to reduce hazards.

1.1.3.3 Battery Notice

Caution!
There is a risk of explosion if the battery is replaced incorrectly. Replace only with the same or equivalent type recommended by the manufacturer. Discard used batteries according to the manufacturer’s instructions. Please also see the battery supplementary sheet! Valid only for devices containing a battery.

1.1.3.4 Warning of Hot Surface

Caution!
There is danger of burning! The device can become very hot during operation and should not be touched without appropriate precautions.
### 1.1.3.5 Electrostatically Sensitive Components (ESD)

**Caution!**
A sudden electrostatic discharge could destroy sensitive components. Therefore, proper packaging and grounding prescriptions must be observed. Please always observe the following safety advices:

1. Plug-in cards always have to be transported in electrostatically safe containers or bags.
2. Please leave electrostatically sensitive components in their containers until they have reached the electrostatically safe assembly site.
3. Take care that you are correctly grounded if you touch electrostatically sensitive components.
4. Make sure that electrostatically sensitive components are stored in protective packages or on antistatic mats.

**Example**
By observing the following measures, electrostatic damage to the device can be avoided:

1. Lay out antistatic mats at the work place. Wear a grounding strap which is connected to the work place and the working tools.
2. Use antistatic foot mats, foot grounding equipment or air ionizers to provide additional safety.
3. Only touch sensitive components, plug-in cards and units on the housing or at the outer edges of the plug-in cards.
4. Avoid contact with pins, wires and conductor tracks.
5. Make sure that all voltage and signal sources are switched off before establishing or disconnecting electric connections or connecting testing devices.
6. Avoid non-conductive materials such as usual mounting accessories made of plastics or polystyrene at the work place.
7. Use conductive tools (e.g. ESD screwdrivers) when working on electrostatically sensitive units and components.
8. Always put plug-in cards and drives on the antistatic mat with the component side downwards.

### 1.1.4 Disclaimer

**Caution!**
Claims based on defects and liability claims against Vector are excluded to the extent damages or errors are caused by improper use of the hardware or use not according to its intended purpose. The same applies to damages or errors arising from insufficient training or lack of experience of personnel using the hardware.

**Caution!**
Claims based on defects and liability claims against Vector are excluded to the extent damages or errors are caused by improper use of the loggers or use not according to its intended purpose. The same applies to damages or errors arising from insufficient training or lack of experience of personnel using the loggers.
Caution!
The firmware of the logger contains the copyright protected emFile module of SEGGER Microcontroller GmbH & Co. KG.

It should be noted that the Licensing Terms of the Licensor expressly prohibit the use of the SEGGER emFile module in weapons/weapons systems and/or their deployment in same. A “weapons system” is to be understood as meaning, in particular, a system whose primary or material purpose is to injure, incapacitate or kill a person or an opponent, or to destroy or damage the property of a person or an opponent, or to threaten a person or an opponent, irrespective of whether the weapon or the weapon system can be used to attack, defend, threaten or protect.
2 Overview

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 General Information</td>
<td>28</td>
</tr>
<tr>
<td>2.2 Principles of the XL Driver Library</td>
<td>29</td>
</tr>
<tr>
<td>2.3 Driver Files and Examples</td>
<td>33</td>
</tr>
<tr>
<td>2.4 System Requirements</td>
<td>34</td>
</tr>
<tr>
<td>2.5 Additional Information</td>
<td>35</td>
</tr>
<tr>
<td>2.6 License Management</td>
<td>36</td>
</tr>
</tbody>
</table>
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.

![Diagram of applications using the XL Driver Library to access Vector devices](image)

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.

![Diagram of Vector Hardware Config tool](image)

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

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

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.

**Selecting device and channels**

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 44).
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.:

<table>
<thead>
<tr>
<th>Device No.</th>
<th>Channel Index</th>
<th>Access Mask (bin)</th>
<th>Access Mask (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>0b00000001</td>
<td>0x001</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0b00000010</td>
<td>0x002</td>
</tr>
<tr>
<td>02</td>
<td>3</td>
<td>0b01000000</td>
<td>0x040</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0b10000000</td>
<td>0x080</td>
</tr>
<tr>
<td>1 + 4</td>
<td></td>
<td>0b10000001</td>
<td>0x81</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vxlapid.dll</td>
<td>32 bit DLL for Windows 7/8/10</td>
</tr>
<tr>
<td>vxlapid64.dll</td>
<td>64 bit DLL for Windows 7/8/10</td>
</tr>
<tr>
<td>vxlapih</td>
<td>C header for C/C++ based applications</td>
</tr>
<tr>
<td>vxlapi_NET.dll</td>
<td>Wrapper for .NET bases applications (requires vxlapid.dll/vxlapid64.dll)</td>
</tr>
<tr>
<td>vxlapi_NET.xml</td>
<td>Wrapper documentation, used by IntelliSense function</td>
</tr>
</tbody>
</table>

**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 x1CANcontrol and the module x1Loadlib.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
- VN1500 Interface Family
- VN1600 Interface Family
- VN2600 Interface Family
- VN5000 Interface Family
- VN7000 Interface Family
- VN8800 Interface Family
- VN8900 Interface Family
- VX0312/VX1135/VX1161.41

**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

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.

<table>
<thead>
<tr>
<th>Flags</th>
<th>Supported Bus Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400000</td>
<td>CAN, LIN, DAIO</td>
</tr>
<tr>
<td>0x2000</td>
<td>MOST</td>
</tr>
<tr>
<td>0x010000</td>
<td>FlexRay</td>
</tr>
</tbody>
</table>

Reference

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

2.6.1 General Information

**Advanced libraries** While most features of the XL Driver Library are free, some features need a license to unlock. Currently, the only feature that needs a license is the FlexRay Advanced Library (see section Free Library and Advanced Library on page 398). A function that requires a license that has not been unlocked returns XL_ERR_NO_LICENSE.

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

- **Old device licenses**
  Old device licenses are bound to a Vector network interface and listed in Vector Hardware Config. Those licenses automatically unlock features in the XL Driver Library.

- **Old Keyman licenses**
  Old Keyman licenses are bound to a Keyman hardware dongle and listed in Vector Hardware Config. To unlock a feature with an old Keyman license, first call xlGetKeymanBoxes() (see page 56) and subsequently call xlGetKeymanInfo() (see page 57).

- **New license model**
  Licenses of the new license model may be stored on a Vector network interface, a Keyman dongle or on the host PC. Those licenses are managed with the Vector License Client. All features licensed via the new model are unlocked by calling xlGetKeymanBoxes() (page 56).
3  Common Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>38</td>
</tr>
<tr>
<td>3.2 Functions</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Structs</td>
<td>61</td>
</tr>
<tr>
<td>3.4 Events</td>
<td>77</td>
</tr>
</tbody>
</table>
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 29).

**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

```c
XLstatus xlOpenDriver(void)
```

Description

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

Return value

Returns an error code (see section Error Codes on page 482).

3.2.2 xlCloseDriver

Syntax

```c
XLstatus xlCloseDriver(void)
```

Description

This function closes the driver.

Return value

Returns an error code (see section Error Codes on page 482).

3.2.3 xlGetApplConfig

Syntax

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

Description

Retrieves the hardware settings for an application which are configured in the Vector Hardware Configuration tool. The information can then be used to get the required channel mask (see section xlGetChannelMask on page 43). 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.
Common Commands

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_HWTYPE_CANCARDXL

- **pHwIndex**
  Index of same hardware types is returned (0, 1, ...), e.g. for two CANcardXL on one system:  
  - CANcardXL 01: hwIndex = 0  
  - CANcardXL 02: hwIndex = 1

- **pHwChannel**
  Channel index of same hardware types is returned (0, 1, ...), e.g. CANcardXL:  
  - Channel 1: hwChannel = 0  
  - Channel 2: hwChannel = 1

**Return value**

Returns an error code (see section Error Codes on page 482).
3.2.4 xlSetApplConfig

Syntax

```c
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_HWTYPE_CANCARDXL`

- **hwIndex**
  
  Index of same hardware types (0, 1, ...), e. g. for two CANcardXL on one system:
  
  CANcardXL 01: `hwIndex = 0`
  CANcardXL 02: `hwIndex = 1`

- **hwChannel**
  
  Channel index on one physical device (0, 1, ...)
  e. g. CANcardXL with `hwIndex=0`:
  Channel 1: `hwChannel = 0`
  Channel 2: `hwChannel = 1`

- **busType**
  
  Specifies the bus type for the application, e. g.
  
  `XL_BUS_TYPE_CAN`
  `XL_BUS_TYPE_LIN`
  `XL_BUS_TYPE_DAIO`

Return value

Returns an error code (see section Error Codes on page 482).

3.2.5 xlGetDriverConfig

Syntax

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

Description

Gets detailed information on the hardware configuration. This function can be called at any time after a successfully `xlOpenDriver()` call. The result describes the current state of the driver configuration after each call.
Note
Applications that search for channels on which they can open a port of a specific bus type should check the corresponding XL_BUS_ACTIVE_CAP_XXX (see section XLchannelConfig on page 61)

Input parameters

- **XLdriverConfig**
  Points to the information structure that is returned by the driver (see section XLDriverConfig on page 61).

Return value
Returns an error code (see section Error Codes on page 482).

### 3.2.6 xlGetRemoteDriverConfig

**Syntax**

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

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

Note
Applications that search for channels on which they can open a port of a specific bus type should check the corresponding XL_BUS_ACTIVE_CAP_XXX (see section XLchannelConfig on page 61)

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

<table>
<thead>
<tr>
<th>VN8910A</th>
<th>VN8950</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN1</td>
<td>CAN2</td>
</tr>
</tbody>
</table>

#### xlGetDriverConfig()

<table>
<thead>
<tr>
<th>channelCount</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAMING internal use</td>
<td></td>
</tr>
<tr>
<td>channelIndex</td>
<td>0</td>
</tr>
<tr>
<td>hwType</td>
<td>XL_HWTYPE_VN8900</td>
</tr>
<tr>
<td>hwChannel</td>
<td>0</td>
</tr>
<tr>
<td>hwIndex</td>
<td>0</td>
</tr>
<tr>
<td>CAN1 VN8950</td>
<td></td>
</tr>
<tr>
<td>channelIndex</td>
<td>1</td>
</tr>
<tr>
<td>hwType</td>
<td>XL_HWTYPE_VN8900</td>
</tr>
<tr>
<td>hwChannel</td>
<td>1</td>
</tr>
<tr>
<td>hwIndex</td>
<td>0</td>
</tr>
<tr>
<td>CAN2 VN8950</td>
<td></td>
</tr>
<tr>
<td>channelIndex</td>
<td>2</td>
</tr>
<tr>
<td>hwType</td>
<td>XL_HWTYPE_VN8900</td>
</tr>
<tr>
<td>hwChannel</td>
<td>2</td>
</tr>
<tr>
<td>hwIndex</td>
<td>0</td>
</tr>
<tr>
<td>CAN3 VN8950</td>
<td></td>
</tr>
<tr>
<td>channelIndex</td>
<td>3</td>
</tr>
<tr>
<td>hwType</td>
<td>XL_HWTYPE_VN8900</td>
</tr>
<tr>
<td>hwChannel</td>
<td>3</td>
</tr>
<tr>
<td>hwIndex</td>
<td>0</td>
</tr>
<tr>
<td>CAN4 VN8950</td>
<td></td>
</tr>
<tr>
<td>channelIndex</td>
<td>4</td>
</tr>
<tr>
<td>hwType</td>
<td>XL_HWTYPE_VN8900</td>
</tr>
<tr>
<td>hwChannel</td>
<td>4</td>
</tr>
<tr>
<td>hwIndex</td>
<td>0</td>
</tr>
</tbody>
</table>
### 3.2.7 `xlGetChannelIndex`

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

**Return value**
Returns the channel index.

### 3.2.8 `xlGetChannelMask`

**Syntax**
```c
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 `xlGetApplConfig()`.
Input parameters

► **hwType**
Required to distinguish the different hardware types, e.g.
-1
XL_HWTYPE_CANCARDXL
XL_HWTYPE_CANBOARDXL
...
Parameter -1 can be used if the hardware type does not matter.

► **hwIndex**
Required to distinguish between two or more devices of the same hardware type (-1, 0, 1...). Parameter -1 is used to retrieve the first available hardware. The type depends on hwType.

► **hwChannel**
Required to distinguish the hardware channel of the selected device (-1, 0, 1...). Parameter -1 can be used to retrieve the first available channel.

Return value

Returns the channel mask.

Example

**Selecting CANcardXL Channel 1**
```
m_xlChannelMask = xlGetChannelMask(XL_HWTYPE_CANCARDXL,-1, 0);
if(!m_xlChannelMask) return XL_ERR_HW_NOT_PRESENT;
xlPermissionMask = m_xlChannelMask;
xlStatus = xlOpenPort(&m_XLportHandle,
    "xICANdemo",
    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)
```
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**.

**Description**

- **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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

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

- **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, K-Line.
  - `XL_INTERFACE_VERSION_V4` for MOST, CAN FD, Ethernet, FlexRay, ARINC429
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 482).

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

Return value
Returns an error code (see section Error Codes on page 482).

### 3.2.12 `xlSetTimerRateAndChannel`

**Syntax**

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

**Description**

This call sets the rate for the port’s cyclic timer events. The resolution is 10 µs (a `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
3. Common Commands

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

**Return value**
Returns an error code (see section Error Codes on page 482).

### 3.2.13 xlResetClock

**Syntax**
```c
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 482).

### 3.2.14 xlSetNotification

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

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

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

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

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

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

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

**Input parameters**
- **portHandle**
  The port handle retrieved by `xlOpenPort()`.
- **queueLevel**
  Queue level in number of events or number of bytes to set on the queue. For LIN, this is fixed to ‘1’. For other bus types, ‘1’ is the recommended value.

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

**Return value**
Returns an error code (see section Error Codes on page 482).

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

### 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 `xlOpenPort()`.

**Return value**
Returns an error code (see section Error Codes on page 482).

### 3.2.16 `xlGetReceiveQueueLevel`

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

**Description**
This function reads the number of events or number of bytes currently in use in a
port’s receive queue. Applications can use this value to compare the actual queue usage to the allocated size requested by the \texttt{rxQueueSize} parameter of \texttt{xlOpenPort}.

**Input parameters**

- **portHandle**
  The port handle retrieved by \texttt{xlOpenPort}.

**Output parameters**

- **level**
  Pointer to an int that receives the actual count of events or bytes. The value depends on the bus type (see section \texttt{xlOpenPort} on page 44).

**Return value**

Returns an error code (see section Error Codes on page 482).

**Note**

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

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

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

### 3.2.17 \texttt{xlActivateChannel}

**Syntax**

\[
\text{XLstatus } \texttt{xlActivateChannel}( \\
\text{XLportHandle } \texttt{portHandle}, \\
\text{XLaccess } \texttt{accessMask}, \\
\text{unsigned int } \texttt{busType}, \\
\text{unsigned int } \texttt{flags})
\]

**Description**

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

**Input parameters**

- **portHandle**
  The port handle retrieved by \texttt{xlOpenPort}.

- **accessMask**
  The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the \texttt{Vector Hardware Configuration} tool if there is a prepared application setup (see section \texttt{xlGetChannelMask} on page 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **busType**
  Bus type that has also been used for \texttt{xlOpenPort}.
flags
Additional flags for activating the channels:

- `XL_ACTIVATE_RESET_CLOCK`
  Resets the internal clock after activating the channel.
- `XL_ACTIVATE_NONE`

Return value
Returns an error code (see section Error Codes on page 482).

Example
Channel Activation
```c
XLStatus xlStatus = xlActivateChannel(m_vPortHandle, 
  &m_vChannelMask[MASTER], 
  XL_BUS_TYPE_LIN, 
  XL_ACTIVATE_RESET_CLOCK);
```

3.2.18 xlReceive

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

Description
Reads the received events from the message queue. Supported bus types:
- CAN
- LIN
- K-Line
- DAIO

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

Input parameters
- `portHandle`
The port handle retrieved by `xlOpenPort()`.

Input/output parameters
- `pEventCount`
  Pointer to an event counter. On input, the variable must be set to the size (in messages) of the received buffer. On output, the variable contains the number of received messages.
- `pEventList`
  Pointer to the application allocated receive event buffer (see section `XLevent` on page 77). 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 482)
Example

Reading messages from queue

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

// Wait for event
while (g_RXThreadRun) {
    WaitForSingleObject(g_hMsgEvent, 10);
    xlStatus = XL_SUCCESS;
    while (!xlStatus) {
        msgsrx = RECEIVE_EVENT_SIZE;
        xlStatus = xlReceive(g_xlPortHandle, &msgsrx, &xlEvent);
        if (xlStatus != XL_ERR_QUEUE_IS_EMPTY) {
            if (!g_silent) {
                printf("%s\n", xlGetEventString(&xlEvent));
            }
        }
    }
}

3.2.19 xlGetEventString

Syntax

XLstringType xlGetEventString (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 77).

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
- **0000000000000000**
  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**

```c
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 482)

**Return value**

Error code as plain text string.

### 3.2.21 xlGetSyncTime

**Syntax**

```c
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 482).

3.2.22 xlGetChannelTime

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

Output parameters

- pChannelTime
  64 bit PC-based card time.

Return value

Returns an error code (see section Error Codes on page 482).

3.2.23 xlGenerateSyncPulse

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

Return value

Returns an error code (see section Error Codes on page 482).
### 3.2.24 xlPopupHwConfig

**Syntax**

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

**Description**

Call this function to pop up the Vector Hardware Config tool.

**Input parameters**

- **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 482).

### 3.2.25 xlDeactivateChannel

**Syntax**

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

**Description**

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

**Input parameters**

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

- **accessMask**
  
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

**Return value**

Returns an error code (see section Error Codes on page 482).

### 3.2.26 xlGetLicenseInfo

**Syntax**

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

**Description**

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

**Input parameters**

- **channelMask**
  
  The channel mask of the Vector device containing the licenses.
3.2.27 xlSetGlobalTimeSync

**Syntax**

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

**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**

► **newValue**

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

XL_SET_TIMESYNC_ON
Enables the software synchronization in the Vector Hardware 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 482).

3.2.28 xlGetKeymanBoxes

**Syntax**

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

**Description**

Returns the number of connected Keyman license dongles. In addition, all available library relevant licenses (new license model only, e.g. advanced FlexRay support) found on any connected Vector network interface are activated. The activation is required to use the advanced features of the XL API. For further details, refer to section License Management on page 36.

**Output parameters**

► **boxCount**

Number of connected Keyman license dongles.

**Return value**

Returns XL_SUCCESS if returned `boxCount` is larger than 0 or if at least one library feature was unlocked via the new license model. Otherwise the function returns an error code (see section Error Codes on page 482).
### 3.2.29 xlGetKeymanInfo

**Syntax**

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

**Description**

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

**Input parameters**

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

**Output parameters**

- **boxMask**
  Mask of the Keyman license dongle.
- **boxSerial**
  Serial of the Keyman license dongle.
- **licInfo**
  License Info (license bits in license array).
  The structure's size is 4*64 bits (see example below).

**Return value**

Returns an error code (see section Error Codes on page 482).
Example

```c
XLstatus x1Status = XL_ERROR;
unsigned int nbrOfBoxes;
unsigned int boxMask;
unsigned int boxSerial;
unsigned int i;
XLUint64 licInfo[4], tmpLicInfo[4];
memset(LicInfo, 0, sizeof(LicInfo));
x1Status = xlGetKeymanBoxes(&nbrOfBoxes);
if (x1Status == XL_SUCCESS) {
    sprintf(tmp, "xlGetKeymanBoxes: %d Keyman License Dongle(s) found!\n", 
nbrOfBoxes);
    XLD_DEBUG(DEBUG_ADV, tmp);
    for (i = 0; i<nbrOfBoxes; i++) {
        memset(tmpLicInfo, 0, sizeof(tmpLicInfo));
        x1Status = xlGetKeymanInfo(i, &boxMask, &boxSerial, tmpLicInfo);
        if (x1Status == XL_SUCCESS) {
            sprintf(tmp, "xlGetKeymanInfo: Keyman Dongle (%d) with SerialNumber: 
%d-%d\n", i, boxMask, boxSerial); 
            XLD_DEBUG(DEBUG_ADV, tmp);
            licInfo[0] |= tmpLicInfo[0];
            licInfo[1] |= tmpLicInfo[1];
        }
    }
    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]);
    XLD_DEBUG(DEBUG_ADV, tmp);
}
```

3.2.30 xlCreateDriverConfig

**Syntax**

```c
XLstatus xlCreateDriverConfig ( 
    XLIdriverConfigVersion version, 
    struct XLIdriverConfig *pConfigInterface)
```

**Description**

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

- Has a versioned interface, which will allow later XL API versions to add additional fields and structs.
- Provides information on networks, segments, measurement points and virtual ports (see section Network Based Access Mode on page 125).
- Logically separates devices from the channels on these devices.
- Is not limited to 64 channels.
- Combines the local and remote channel information in a common structure.

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

### Input parameters
- **version**
  - Requested version of XLDriverConfig. Currently the only value is XL_IDRIVER_CONFIG_VERSION_1.

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

### Return value
- Returns an error code (see section Error Codes on page 482). If the current version of the DLL does not support the requested version of the XLDriverConfig, XL_ERR_NOT_IMPLEMENTED is returned.

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

```c
XLapiIDriverConfigV1 config;
XLdeviceDrvConfigListV1 devices;

XLstatus status = xlCreateDriverConfig(XL_IDRIVER_CONFIG_VERSION_1,
                                        (XLIDriverConfig*)&config);

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

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

if (status == XL_SUCCESS) {
    for (unsigned int i = 0; i < devices.count; ++i) {
        const XLdeviceDrvConfigV1& device = devices.item[i];
        printf("%s\n", device.name);
        for (unsigned int j = 0; j < device.channelList.count; ++j) {
            const XLchannelDrvConfigV1& channel =
                device.channelList.item[j];
            printf("- [%u] %s\n",
                   channel.channelIndex,
                   channel.transceiver.name);
        }
    }
}

xlDestroyDriverConfig(config.configHandle);
return status;
```

### 3.2.31 xlDestroyDriverConfig

#### Syntax
```
XLstatus xlDestroyDriverConfig(XLdrvConfigHandle configHandle)
```

#### Description
This function releases the memory allocated for an instance of the XLDriverConfig structure.
Input parameters

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

Return value

Returns an error code (see section Error Codes on page 482).

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

3.3.1 XLdriverConfig

**Syntax**

```c
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:
  ```c
  (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 61). XL_CONFIG_MAX_CHANNELS=64.

3.3.2 XLchannelConfig

**Syntax**

```c
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 61).
Parameters

- **name**
  The channel’s name.

- **hwType**
  Contains the hardware types (see `vxlapi.h`),
  e. g. CANcardXL: `XL_HWTYPE_CANCARDXL`

- **hwIndex**
  Index of same hardware types (0, 1, ...),
  e. g. for two CANcardXL on one system:
  CANcardXL 01: `hwIndex = 0`
  CANcardXL 02: `hwIndex = 1`

- **hwChannel**
  Channel index on one physical device (0, 1, ...)
  e. g. CANcardXL with `hwIndex=0`:
  Channel 1: `hwChannel = 0`
  Channel 2: `hwChannel = 1`

- **transceiverType**
  Contains type of Cab or Piggyback,
  e. g. 251 Highspeed Cab: `XL_TRANSCEIVER_TYPE_CAN_251`

- **transceiverState**
  State of the transceiver.

- **channelIndex**
  Global channel index (0, 1, ...).

- **channelMask**
  Global channel mask (1 << `channelIndex`).

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

The channel (hardware) supports the bus types:
- XL_BUS_COMPATIBLE_CAN
- XL_BUS_COMPATIBLE_LIN
- XL_BUS_COMPATIBLE_FLEXRAY
- XL_BUS_COMPATIBLE_MOST
- XL_BUS_COMPATIBLE_DAIO
- XL_BUS_COMPATIBLE_J1708
- XL_BUS_COMPATIBLE_Ethernet
- XL_BUS_COMPATIBLE_A429
- XL_BUS_COMPATIBLE_KLINE

The connected Cab or Piggyback supports the bus type:
- XL_BUS_ACTIVE_CAP_CAN
- XL_BUS_ACTIVE_CAP_LIN
- XL_BUS_ACTIVE_CAP_FLEXRAY
- XL_BUS_ACTIVE_CAP_MOST
- XL_BUS_ACTIVE_CAP_DAIO
- XL_BUS_ACTIVE_CAP_J1708
- XL_BUS_ACTIVE_CAP_Ethernet
- XL_BUS_ACTIVE_CAP_A429
- XL_BUS_ACTIVE_CAP_KLINE

isOnBus
The flag specifies whether the channel is on bus (1) or off bus (0).

connectedBusType
The flag specifies to which bus type the channel is connected, e.g.
- XL_BUS_TYPE_CAN
- ...

Note: The flag is only set when the channel is on bus.

busParams
Current bus parameters (see section XLbusParams on page 64).

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.
3.3.3 XLbusParams

Syntax

```c
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;
    }
}
```

- `specialCabFlags`: Only for internal use.
- `dominantTimeout`: Only for internal use.
- `reserved`: Reserved for future use.
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;
}

unsigned char raw[28];
}
}

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 NO ISO: XL_BUS_PARAMS_CANOPMODE_CANFD_NO_ISO

### 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_MODEASYNCHRONOUS

► baudrate
   FlexRay baud rate in kBaud.

Ethernet
► macAddr
   The MAC address starting with the MSB. This field is only defined on VN5610(A) and VN5640. In network-based mode, MAC addresses are available via xlNetRequestMACAddress().
► connector
The interface connector currently assigned to the MAC:
XL_ETH_STATUS_CONNECTOR_RJ45
XL_ETH_STATUS_CONNECTOR_DSUB

► phy
The currently active transmitter (physical interface):
XL_ETH_STATUS_PHY_UNKNOWN
XL_ETH_STATUS_PHY_802_3
XL_ETH_STATUS_PHY_BROADR_REACH

► link
Link state:
XL_ETH_STATUS_LINK_UNKNOWN
XL_ETH_STATUS_LINK_DOWN
XL_ETH_STATUS_LINK_UP
XL_ETH_STATUS_LINK_ERROR

► speed
Current Ethernet connection speed:

XL_ETH_STATUS_SPEED_UNKNOWN

XL_ETH_STATUS_SPEED_10
10 MBit/s operation.

XL_ETH_STATUS_SPEED_100
100 MBit/s operation.

XL_ETH_STATUS_SPEED_1000
1000 MBit/s operation.

► clockMode
Clock mode setting of the connection:

XL_ETH_STATUS_CLOCK_DONT_CARE
Reported for IEEE 802.3.

XL_ETH_STATUS_CLOCK_MASTER
XL_ETH_STATUS_CLOCK_SLAVE

► bypass
XL_ETH_BYPASS_INACTIVE (Default)
XL_ETH_BYPASS_PHY
XL_ETH_BYPASS_MACCORE

ARINC 429

► channelDirection
See XL_A429_PARAMS.

► res1
Reserved for future use.

► bitrate
See XL_A429_PARAMS.
3.3.4 XLlicenseInfo

**Syntax**

```c
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

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

3.3.5 XLapiIDriverConfigV1

Syntax

```c
typedef struct s_xlapi_driver_config_v1 {
  XDrvConfigHandle configHandle;
  TP_FCT_XLAPI_GET_DEVICE_CONFIG_V1 fctGetDeviceConfig;
  TP_FCT_XLAPI_GET_CHANNEL_CONFIG_V1 fctGetChannelConfig;
  TP_FCT_XLAPI_GET_NETWORK_CONFIG_V1 fctGetNetworkConfig;
  TP_FCT_XLAPI_GET_SWITCH_CONFIG_V1  fctGetSwitchConfig;
  TP_FCT_XLAPI_GET_VIRTUAL_PORT_CONFIG_V1 fctGetVirtualPortConfig;
  TP_FCT_XLAPI_GET_MEASUREMENT_POINT_CONFIG_V1 fctGetMeasurementPointConfig;
  TP_FCT_XLAPI_GET_DLL_CONFIG_V1 fctGetDllConfig;
} XLapiIDriverConfigV1, *pXLapiIDriverConfigV1;
```

Description

XLAPI driver configuration interface structure version 1.

Parameters

- **configHandle**
  Handle for this XLIDriverConfig instance. It is the first argument to the function pointers in this structure.

- **fctGetDeviceConfig**
  Gets the list of devices, see XLdeviceDrvConfigV1.
Common Commands

- fctGetChannelConfig
  Gets the list of channels, see XLchannelDrvConfigV1.

- fctGetNetworkConfig
  Gets the list of networks, see XLnetworkDrvConfigV1.

- fctGetSwitchConfig
  Gets the list of switches, see XLswitchDrvConfigV1.

- fctGetVirtualPortConfig
  Gets the list of virtual ports, see XLvirtualPortDrvConfigV1.

- fctGetMeasurementPointConfig
  Gets the list of measurement points, see XLmeasurementPointDrvConfigV1.

- fctGetDllConfig
  Gets information on the loaded XL API DLL, see XLdllDrvConfigV1.

3.3.6 XLchannelDrvConfigV1

Syntax

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

    struct {
        const char* name;
        unsigned int type;
        unsigned int configError;
    } transceiver;

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

Description

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

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

Parameters

- hwChannel
  Index of this channel relative to its device.

- channelIndex
  Index of the local channel in the channel list returned by fctGetChannelConfig. This index uniquely identifies a local channel in the current application. Remote channels have the same channelIndex than their local counterpart. The channelMask can be computed as (XLaccess)1 << channelIndex.
deviceIndex
Index of the channel’s device in the device list returned by fctGetDeviceConfig.

interfaceVersion
Interface version supported by this channel, currently either XLINTERFACE_VERSION_V3 or XLINTERFACE_VERSION_V4.

isOnBus
Is 1 while at least one application activated this channel (“is on bus”) and 0 otherwise.

channelCapabilities
Bitwise combination of the values below. Note the difference between the XLCHANNEL_FLAG_EX1 defines and the XLCHANNEL_FLAG defines used in the channelCapabilities field of XLchannelConfig.
- XL_CHANNEL_FLAG_EX1_TIME_SYNC_RUNNING
  The channel is synchronized with legacy software time synchronization.
- XL_CHANNEL_FLAG_EX1_HWSYNC_SUPPORT
  The channel supports hardware synchronization, see section XL Sync Pulse on page 78.
- XL_CHANNEL_FLAG_EX1_CANFD_ISO_SUPPORT
  The channel supports ISO-compliant CAN-FD operation.
- XL_CHANNEL_FLAG_EX1_SPDIF_CAPABLE
  The channel supports SPDIF, used to distinguish between VN2600 (w/o SPDIF) and VN2610 (with S/PDIF).
- XL_CHANNEL_FLAG_EX1_CANFD_BOSCH_SUPPORT
  The channel supports the non-ISO CAN-FD checksum mode, see also XLcanFdConf.
- XL_CHANNEL_FLAG_EX1_NET_ETH_SUPPORT
  The ethernet device operates in network-based instead of channel-based mode, see section Switching Access Mode on page 123.

channelCapabilities2
For future use.

channelBusCapabilities
Bitwise combination of XL_BUS_TYPE_* values that defines the set of bus types that the hardware can support on this channel if equipped with an appropriate transceiver.

channelBusActiveCapabilities
Bitwise combination of XL_BUS_TYPE_* values that the hardware can support on this channel using the currently equipped transceiver. This serves the same purpose as the XL_BUS_ACTIVE_CAP values of the channelBusCapabilities field in XLchannelConfig. Applications that search for channels on which they can open a port of a specific bus type should check channelBusActiveCapabilities.

connectedBusType
While an application in the system has opened and activated a port on this channel, connectedBusType is the XL_BUS_TYPE_* value for the bustype of the open port. Otherwise this is XL_BUS_TYPE_NONE.

currentlyAvailableTimestamps
Reserved.

busParams
Current bus parameters, see section XLbusParams on page 64.
3.3.7 XLdeviceDrvConfigV1

**Syntax**

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

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

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

**Description**

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

**Parameters**

- **name**
  Name of the device, for example “VN1630A”, as null-terminated UTF-8 string.

- **hwType**
  The XL_HWTYPE_* value for this device.

- **hwIndex**
  Index to differentiate between multiple connected devices of the same hwType.

- **serialNumber**
  Serial number of this device.

- **articleNumber**
  Article number of this device.

- **driverVersion**
  Version of the driver for this device, encoded as major << 56 | minor << 48 | revision << 32. The lower 32 bit are reserved.

- **connectionInfo**
  A 32 bit value that specifies the host connection of the device. The value is split as follows:
- XL_CONNECTION_INFO_FAMILY_MASK
  The upper 8 bit define the family.
- XL_CONNECTION_INFO_DETAIL_MASK
  The lower 24 bit provide additional information. Interpretation depends on the family.

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

The detail information for the USB family reports the speed of the active USB connection:
- XL_CONNECTION_INFO_USB_UNKNOWN
- XL_CONNECTION_INFO_USB_FULLSPEED
- XL_CONNECTION_INFO_USB_HIGHSPEED
- XL_CONNECTION_INFO_USB_SUPERSPEED

► isRemoteDevice
  1 if this is a remote device, 0 if it is a local device.
► remoteDeviceList
  List of remote devices connected to this device.
► channelList
  List of channels that this device provides, see XLchannelDrvConfigV1.

Note
The host connection of VN8900 devices can be detected via the hwType XL_HWTYPE_VN8900 (USB) versus XL_HWTYPE_IPCLIENT (network).
The device list contains the Vector Timesync Service (XL_HWTYPE_VTSSERVICE) but it is not possible to interact with this pseudo device using the XL API.

### 3.3.8 XLnetworkDrvConfigV1

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

**Description**
This structure contains information on a network that is configured on at least one connected device (see section Network Based Access Mode on page 125).

**Parameters**
- **networkName**
  Name of the network as null-terminated UTF-8 string.
Common Commands

A code that describes a configuration error.

- **XL_NET_CFG_STAT_OK**
  The network is OK and the application can open it.

- **XL_NET_CFG_DUPLICATE_SEGMENT_NAME**
  The network contains two segments with the same name.

- **XL_NET_CFG_DUPLICATE_VP_NAME**
  The network contains two virtual ports with the same name.

- **XL_NET_CFG_DUPLICATE_MP_NAME**
  The network contains two measurement points with the same name.

**statusErrorString**

A null-terminated UTF-8 string that contains an English error message that explains the statusCode. In case of a name-duplication, the message contains the duplicate name. If no error message is available, the string is empty.

**networkType**

Currently always XL_ETH_NETWORK.

**switchList**

List of switches that are part of this network, see XLswitchDrvConfigV1.

### 3.3.9 XLswitchDrvConfigV1

**Syntax**

```c
typedef struct s_xl_switch_drv_config_v1 {
    const char *switchName;
    XLswitchId switchId;
    unsigned int networkIdx;
    const XldeviceDrvConfigV1 *device;
    unsigned int switchCapability;
    XLvirtualportDrvConfigListV1 vpList;
    XlmeasurementpointDrvConfigListV1 mpList;
} XLswitchDrvConfigV1, *pXLswitchDrvConfigV1;
```

**Description**

This structure contains information on a switch that is part of a network. Note that the term “switch” in the driver configuration refers to any kind of segment. For the definition of segment, see section Network Based Access Mode on page 125.

**Parameters**

- **switchName**
  Name of the switch as null-terminated UTF-8 string.

- **switchId**
  ID of the switch in the network. Switches in different networks may have the same switch ID.

- **networkIdx**
  Index of the network in the network list returned by fctGetNetworkConfig.

- **device**
  Pointer to the device that this switch resides on.

- **switchCapability**
  Defines what kind of segment this “switch” is:

  - **XL_NET_ETH_SWITCH_CAP_REALSWITCH**
    This is a switch segment (MAC address learning is on).

  - **XL_NET_ETH_SWITCH_CAP_DIRECTCONN**
    This segment is a direct connection.
3.3.10  XLvirtualportDrvConfigV1

**Syntax**

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

**Description**

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

**Parameters**

- **virtualPortName**
  Name of the virtual port as null-terminated UTF-8 string.

- **networkIdx**
  Index of the network in the network list returned by `fctGetNetworkConfig`.

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

3.3.11  XLmeasurementpointDrvConfigV1

**Syntax**

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

**Description**

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

**Parameters**

- **measurementPointName**
  Name of the measurement point as null-terminated UTF-8 string.

- **networkIdx**
  Index of the network in the network list returned by `fctGetNetworkConfig`.

- **switchId**
  ID of the switch that this measurement point is connected to.

- **channel**
  Pointer to the channel associated with the measurement point or NULL if no such channel exists.
3.3.12 XLdllDrvConfigV1

**Syntax**

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

**Description**

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

**Parameters**

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

3.4.1 XEvent

Syntax

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

Parameters

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

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

  All K-Line events
  - XL_KLINE_MSG

  Special DAIO events
  - XL_RECEIVE_DAIO_DATA

- **chanIndex**
  Channel on which the event occurs.

- **transId**
  Internal use only.

- **portHandle**
  Internal use only.

- **flags**
  e.g. XL_EVENT_FLAG_OVERRUN

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

- **time stamp**
  Actual time stamp generated by the hardware with 8 μs resolution. Value is in nanoseconds.
3.4.2 XL Tag Data

**Syntax**

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

**Parameters**

- **msg**
  Union for all CAN events.

- **chipState**
  Structure for all CHIPSTATE events.

- **linMsgApi**
  Union for all LIN events.

- **syncPulse**
  Structure for all SYNC_PULSE events.

- **transceiver**
  Structure for all TRANSCEIVER events.

- **daioData**
  Structure for all DAIO data.

- **daioPiggyData**
  Structure for all DAIO Piggy data.

- **klineData**
  Structure for all K-Line events.

3.4.3 XL Sync Pulse

**Syntax**

```c
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 77).
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

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

Tag

XL_TRANSCEIVER (see section XLevent on page 77).

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 XLevent on page 77).
4 CAN Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>81</td>
</tr>
<tr>
<td>4.2 Flowchart</td>
<td>82</td>
</tr>
<tr>
<td>4.3 Functions</td>
<td>83</td>
</tr>
<tr>
<td>4.4 Structs</td>
<td>94</td>
</tr>
<tr>
<td>4.5 Events</td>
<td>95</td>
</tr>
<tr>
<td>4.6 Application Examples</td>
<td>98</td>
</tr>
</tbody>
</table>
4.1 Introduction

**Description**

The XL Driver Library enables the development of CAN applications for supported Vector devices (see section System Requirements on page 34). Multiple CAN applications can use a common physical CAN channel at the same time.

Depending on the channel property *init access* (see page 31), 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

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 xlOpenPort on page 44) for the specified channels, and the channels must be deactivated.

4.3.3 xlCanSetReceiveMode

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

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

Input parameters
► portHandle
The port handle retrieved by xlOpenPort().

► ErrorFrame
Suppresses error frames.

► ChipState
Suppresses chipstate events.

Return value
Returns an error code (see section Error Codes on page 482).

4.3.4 xlCanSetChannelTransceiver

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

**Description**
This function is used to set the transceiver modes. The possible transceiver modes depend on the transceiver type connected to the hardware. The port must have *init access* (see section `xlOpenPort` on page 44) to the 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 43). For further information on channel/access masks please also refer to section *Principles of the XL Driver Library* on page 29.
- **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_FB_CAN_TT_OPTO`

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

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

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

- **XL_TRANSCEIVER_LINEMODE_NORMAL**
  - Enables normal operation.

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

- **XL_TRANSCEIVER_LINEMODE_SWC_SLEEP**
  - Switches to sleep mode.

- **XL_TRANSCEIVER_LINEMODE_SWC_NORMAL**
  - Switches to normal operation.

- **XL_TRANSCEIVER_LINEMODE_SWC_FAST**
  - Switches transceiver to fast mode.

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

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

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

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

**Return value**

Returns an error code (see section Error Codes on page 482).

### 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 44), and the selected channels must be deactivated.
Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section *Principles of the XL Driver Library* on page 29.

- **pChipParams**
  Pointer to an array of chip parameters (see section `XLchipParams` on page 94).

Return value

Returns an error code (see section *Error Codes* on page 482).

### 4.3.6 `xlCanSetChannelParamsC200`

#### Syntax

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

#### Description

This function initializes the channels defined by `accessMask` with the given parameters. In order to call this function, the port must have `init access` (see section `xlOpenPort` on page 44), and the selected channels must be deactivated.

#### Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section *Principles of the XL Driver Library* on page 29.

- **btr0**
  BTRO value for a C200 or 527 compatible controllers.

- **btr1**
  BTR1 value for a C200 or 527 compatible controllers.

#### Return value

Returns an error code (see section *Error Codes* on page 482).

### 4.3.7 `xlCanSetChannelBitrate`

#### Syntax

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

#### Description

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

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

► **accessMask**
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► **bitrate**
Bit rate in BPS. May be in the range 15000 … 1000000.

Return value

Returns an error code (see section Error Codes on page 482).

4.3.8 xlCanSetChannelAcceptance

Syntax

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

Description

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

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

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

**Note**

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

Input parameters

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

► **accessMask**
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► **code**
The acceptance code for id filtering.

► **mask**
The acceptance mask for id filtering, bit = 1 means relevant.
idRange
To distinguish whether the filter is for standard or extended identifiers:

- XL_CAN_STD
- XL_CAN_EXT

Return value
Returns an error code (see section Error Codes on page 482).

Example
Several acceptance filter settings.

<table>
<thead>
<tr>
<th>IDs</th>
<th>mask</th>
<th>code</th>
<th>idRange</th>
</tr>
</thead>
<tbody>
<tr>
<td>Std. Open for all IDs</td>
<td>0x000</td>
<td>0x000</td>
<td>XL_CAN_STD</td>
</tr>
<tr>
<td>Open for ID 1, ID=0x001</td>
<td>0x7FF</td>
<td>0x001</td>
<td>XL_CAN_STD</td>
</tr>
<tr>
<td>Close for all IDs</td>
<td>0xFFF</td>
<td>0xFFF</td>
<td>XL_CAN_STD</td>
</tr>
<tr>
<td>Ext. Open for all IDs</td>
<td>0x000</td>
<td>0x000</td>
<td>XL_CAN_EXT</td>
</tr>
<tr>
<td>Open for ID 1, ID=0x80000001</td>
<td>0xFFFFFFFF</td>
<td>0x001</td>
<td>XL_CAN_EXT</td>
</tr>
<tr>
<td>Close for all IDs</td>
<td>0xFFFFFFFF</td>
<td>0xFFFFFFFF</td>
<td>XL_CAN_EXT</td>
</tr>
</tbody>
</table>

Example
Open filter for all standard message IDs

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

4.3.9 xlCanAddAcceptanceRange

Syntax

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

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

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

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

Input parameters
- `portHandle` The port handle retrieved by `xlOpenPort()`.
accessMask

The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

first_id

First ID to pass acceptance filter.

last_id

Last ID to pass acceptance filter.

Returns an error code (see section Error Codes on page 482).

### Example

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

```c
xlStatus = xlCanAddAcceptanceRange(XLportHandle, xlChannelMask, 10, 17);
xlStatus = xlCanAddAcceptanceRange(XLportHandle, xlChannelMask, 22, 33);
```

### 4.3.10 xlCanRemoveAcceptanceRange

#### Syntax

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

#### Description

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

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

**Note**

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

#### Input parameters

- **portHandle**

  The port handle retrieved by xlOpenPort().
accessMask
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

first_id
First ID to remove.

last_id
Last ID to remove (inclusive).

Return value
Returns an error code (see section Error Codes on page 482).

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

Example

Opening filter for all messages with extended IDs
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 43).
  For further information on channel/access masks please also refer to section
  Principles of the XL Driver Library on page 29.

Return value

Returns an error code (see section Error Codes on page 482).

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 43).
  For further information on channel/access masks please also refer to section
  Principles of the XL Driver Library on page 29.

► 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 482).

#### Example
**Transmitting 100 CAN messages with the ID = 4**
```c
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
```c
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 482).
4.4 Structs

4.4.1 XLchipParams

**Syntax**

```c
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**

\[
\text{Baudrate} = \frac{f}{2^{\text{presc}}(1 + \text{tseg1} + \text{tseg2})}
\]

<table>
<thead>
<tr>
<th>Presc</th>
<th>sjw</th>
<th>tseg1</th>
<th>tseg2</th>
<th>sam</th>
<th>Baudrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1 MBd</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>500 kBd</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>12</td>
<td>7</td>
<td>3</td>
<td>100 kBd</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>16</td>
<td>8</td>
<td>3</td>
<td>10 kBd</td>
</tr>
</tbody>
</table>
4.5 Events

4.5.1 XL CAN Message

Syntax

```c
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 77.
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**

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

**Description**

This event occurs after calling `xlCanRequestChipState()`.
XL_CHIP_STATE (see section XLevent on page 77).

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 xlCANdemo

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:

```
xlCANdemo <Baudrate> <ApplicationName> <Identifier>
```

Figure 6: Running xlCANdemo

4.6.1.2 Keyboard Commands

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;t&gt;</td>
<td>Transmit a message</td>
</tr>
<tr>
<td>&lt;B&gt;</td>
<td>Transmit a message burst</td>
</tr>
<tr>
<td>&lt;M&gt;</td>
<td>Transmit a remote message</td>
</tr>
<tr>
<td>&lt;G&gt;</td>
<td>Request chip state</td>
</tr>
<tr>
<td>&lt;S&gt;</td>
<td>Start/stop</td>
</tr>
<tr>
<td>&lt;R&gt;</td>
<td>Reset clock</td>
</tr>
<tr>
<td>&lt;+&gt;</td>
<td>Select channel (up)</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>Select channel (down)</td>
</tr>
<tr>
<td>&lt;i&gt;</td>
<td>Select transmit Id (up)</td>
</tr>
<tr>
<td>&lt;l&gt;</td>
<td>Select transmit Id (down)</td>
</tr>
<tr>
<td>&lt;X&gt;</td>
<td>Toggle extended/standard Id</td>
</tr>
<tr>
<td>&lt;O&gt;</td>
<td>Toggle output mode</td>
</tr>
<tr>
<td>&lt;A&gt;</td>
<td>Toggle timer</td>
</tr>
<tr>
<td>&lt;V&gt;</td>
<td>Toggle logging to screen</td>
</tr>
<tr>
<td>&lt;P&gt;</td>
<td>Show hardware configuration</td>
</tr>
<tr>
<td>&lt;H&gt;</td>
<td>Help</td>
</tr>
</tbody>
</table>
### 4.6.1.3 Functions

<table>
<thead>
<tr>
<th>Description</th>
<th>The source file <code>xlCANdemo.c</code> contains all needed functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>► <strong>demoInitDriver()</strong></td>
</tr>
<tr>
<td></td>
<td>This function opens the driver and reads the actual hardware configuration. A valid <code>channelMask</code> is calculated and one port is opened afterwards.</td>
</tr>
<tr>
<td></td>
<td>► <strong>demoInitDriver()</strong></td>
</tr>
<tr>
<td></td>
<td>In order to read the driver message queue a thread is generated.</td>
</tr>
</tbody>
</table>
4.6.2 xlCANcontrol

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 - xlCANcontrol accesses VN1630A (CH1/CH2)](image)

**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.
4.6.2.2 Classes

Description

- **CaboutDlg**
  About box.

- **CXLCANcontrolApp**
  Main MFC class ➔ xlCANcontrol.cpp

- **CXLCANcontrolDlg**
  The 'main' dialog box ➔ xlCANcontrollDlg.cpp

- **CCANFunctions**
  Contains all functions for the LIN access ➔ xlCANFunctions.cpp

4.6.2.3 Functions

Description

- **CANInit**
  This function is called on application start to get the valid channel masks (access masks). Afterwards, one port is opened for the two channels and a thread is created to read the message queue.

- **CANGoOnBus**
  After pressing the [Go OnBus] button, the CAN parameters are set and both channels are activated.

- **CANGoOffBus**
  After pressing the [Go OffBus] button, the channels will be deactivated.
- **CANSend**
  Transmits the CAN message with `xlCANtransmit()`.

- **CANResetFilter**
  Resets (open) the acceptance filter.

- **CANSetFilter**
  Sets the acceptance filter range. It is needed to close the acceptance filter for every ID before.

- **canGetChannelMask**
  This function looks for assigned channels in the Vector Hardware Config tool with `xlGetApplConfig()`. If there is no application registered, the application searches for available CAN channels and assigns them in the Vector Hardware Config tool with `xlSetApplConfig()`. The function fails if there are no valid channels found.

- **canInit**
  Opens one port with both channels (see section `xlOpenPort` on page 44).

- **canCreateRxThread**
  In order to read out the driver message queue, the application uses a thread (RxThread). An event is created and set up with `xlSetNotification()` to notify the thread.
# 5 CAN FD Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>104</td>
</tr>
<tr>
<td>5.2 Flowchart</td>
<td>105</td>
</tr>
<tr>
<td>5.3 Functions</td>
<td>106</td>
</tr>
<tr>
<td>5.4 Structs</td>
<td>108</td>
</tr>
<tr>
<td>5.5 Events</td>
<td>111</td>
</tr>
</tbody>
</table>
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 34). Multiple CAN applications can use a common physical CAN FD channel at the same time.

Depending on the channel property init access (see page 31), 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

Driver Init

start

xOpenDriver()

xGetDriverConfig()

xGetChannelMask()

xOpenPort()

Channel Setup

access?

yes

xCanFdSetConfiguration()

xCanSetChannelMode()

xCanSetReceiveMode()

xCanAddAcceptanceRange()

xCanRemoveAcceptanceRange()

xCanResetAcceptance()

xActivateChannel()

xResetClock()

xSetTimerRate()

On Bus

xCanReceive()

xCanEventString()

xCanRequestChipState()

xCanFlushReceiveQueue()

xCanTransmitEx()

xCanFlushTransmitQueue()

xCanReceiveQueueLevel()

xCanErrorString()

xDeactivateChannel()

xClosePort()

xCloseDriver()

end

Figure 9: Function calls for CAN FD applications
5.3 Functions

5.3.1 xlCanFdSetConfiguration

Syntax

```c
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 44) 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **pCanFdConf**
Points to the CAN FD configuration structure to set up a CAN FD channel (see section XLcanFdConf on page 108).

Return value
Returns an error code (see section Error Codes on page 482).

5.3.2 xlCanTransmitEx

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
5 CAN FD Commands

- **msgCnt**
  Amount of messages to be transmitted by the user.

- **pMsgCntSent**
  Amount of messages which were transmitted.

- **pXLCanTxEvt**
  Points to a user buffer with messages to be transmitted (see section XLcanTxEvent on page 111).
  At least the buffer must have the size of msgCnt.

**Return value**
Returns XL_SUCCESS if all requested messages have been successfully transmitted. If no message or not all requested messages have been transmitted because the internal transmit queue is full, XL_ERR_QUEUE_IS_FULL is returned (see section Error Codes on page 482).

### 5.3.3 xlCanReceive

**Syntax**
```c
XLstatus xlCanReceive (  
    XLportHandle portHandle,  
    XLcanRxEvent *pXLCanRxEvt)
```

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

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

**Input/output parameters**
- **pXLCanRxEvt**
  Pointer to the application allocated receive event buffer (see section XLcanRxEvent on page 113).

**Return value**
XL_ERR_QUEUE_IS_EMPTY: No event is available (see section Error Codes on page 482).

### 5.3.4 xlCanGetEventString

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

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

**Input parameters**
- **pEv**
  Points the CAN Rx event buffer to be parsed (see section XLcanRxEvent on page 113).

**Return value**
Returns an error code (see section Error Codes on page 482).
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 char reserved;
  unsigned char options;
  unsigned char reserved1[2];
  unsigned int  reserved2;
} XLcanFdConf;

Parameters

► arbitrationBitRate
  Arbitration CAN bus timing for nominal / arbitration bit rate in bit/s.

► sjwAbr
  Arbitration CAN bus timing value (sample jump width).
  Range: 0 < sjwAbr <= min(tseg2Abr, 128).

► tseg1Abr
  Arbitration CAN bus timing tseg1.
  Range: 1 < tseg1Abr < 255.

► tseg2Abr
  Arbitration CAN bus timing tseg2.
  Range: 1 < tseg2Abr < 255.

► dataBitRate
  CAN bus timing for data bit rate in bit/s.
  Range: dataBitRate >= max(arbitrationBitRate, 25000).

► sjwDbr
  Data phase CAN bus timing value (sample jump width).
  Range: 0 < sjwDbr <= min(tseg2Dbr, 64).

► tseg1Dbr
  Data phase CAN bus timing for data tseg1.
  Range: 1 < tseg1Dbr < 127.

► tseg2Dbr
  Data phase CAN bus timing for data tseg2.
  Range: 1 < tseg2Dbr < 127.

► reserved
  Reserved for future use. Set to 0.

► options
  CANFD-BOSCH
  CANFD_CONFOPT_NO_ISO

► reserved1[2]
  Reserved for future use. Set to 0.
reserved2
Reserved for future use. Set to 0.
Example
Deriving tseg1 and tseg2 for a given bitrate and sample point.

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

<table>
<thead>
<tr>
<th>Arbitration Phase</th>
<th>Data Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitrate</td>
<td>Sample Point</td>
</tr>
<tr>
<td>20k</td>
<td>75%</td>
</tr>
<tr>
<td>125k</td>
<td>67.5%</td>
</tr>
<tr>
<td>500k</td>
<td>87.5%</td>
</tr>
<tr>
<td>1M</td>
<td>75%</td>
</tr>
</tbody>
</table>

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

5.5.1 XLcanTxEvent

**Syntax**
```c
typedef struct {
    unsigned short tag;
    unsigned short transId;
    unsigned char channelIndex;
    unsigned char reserved[3];
    union {
        XL_CAN_TX_MSG canMsg;
    } tagData;
} XLcanTxEvent;
```

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

**Parameters**
- **tag**
  - Event type. Set to XL_CAN_EV_TAG_TX_MSG.
- **transId**
  - Internal use.
- **channelIndex**
  - Internal use. The `accessMask` parameter of `xlCanTransmitEx()` specifies which channels send the message.
- **reserved**
  - Internal use.
- **tagData**
  - Tag Data (see section XL_CAN_TX_MSG on page 111).

5.5.2 XL_CAN_TX_MSG

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

**Tag**
XL_CAN_EV_TAG_TX_MSG

**Parameters**
- **canId**
  - CAN ID (11 or 29 bits).
  - For extended IDs: `canID = (XL_CAN_EXT_MSG_ID | id)`.
► **msgFlags**
Set to 0 to transmit a CAN 2.0 frame.

`XL_CAN_TXMSG_FLAG_BRS`  
Baudrate switch.

`XL_CAN_TXMSG_FLAG_HIGHTPRI0`  
High priority message. Clears all send buffers then transmits.

`XL_CAN_TXMSG_FLAG_WAKEUP`  
Generates a wake up message.

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

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

► **dlc**
4-bit data length code.

<table>
<thead>
<tr>
<th>DLC</th>
<th>Number of Data Bytes CAN 2.0</th>
<th>Number of Data Bytes CAN FD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>

► **reserved**
Internal use.

► **data**
Data to be transmitted.
5.5.3 XLcanRxEvent

Syntax

```c
typedef struct {
    unsigned int   size;
    unsigned short tag;
    unsigned char  channelIndex;
    unsigned char  reserved;
    unsigned int   userHandle;
    unsigned short  flagsChip;
    unsigned short  reserved0;
    XLuint64        reserved1;
    XLuint64        timeStampSync;

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

Description

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

Parameters

- **size**
  Overall size of the complete event.

- **tag**
  - XL CAN_EV_TAG RX_OK
  - XL CAN_EV_TAG RX_ERROR
  - XL CAN_EV_TAG TX_ERROR
  - XL CAN_EV_TAG TX_REQUEST
  - XL CAN_EV_TAG TX_OK
  - XL CAN_EV_TAG CHIP_STATE
  - XL SYNC_PULSE

- **channelIndex**
  Channel index of the hardware (see section `xlGetChannelIndex` on page 43).

- **reserved**
  Internal use.

- **userHandle**
  Internal use.

- **flagsChip**
  Queue overflow (upper 8bit), XL CAN_QUEUE_OVERFLOW.

- **reserved0**
  Internal use.

- **reserved1**
  Internal use.

- **timeStampSync**
  Timestamp which is synchronized by the driver.

- **tagData**
  Tag Data. See the following sections for further details.
5.5.4 XL_CAN_EV_RX_MSG

**Syntax**

```c
typedef struct {
    unsigned int canId;
    unsigned int msgFlags;
    unsigned int crc;
    unsigned char reserved1[12];
    unsigned short totalBitCnt;
    unsigned char dlc;
    unsigned char reserved[5];
    unsigned char data[XL_CAN_MAX_DATA_LEN];
} XL_CAN_EV_RX_MSG;
```

**Tag**

XL_CAN_EV_TAG_RX_OK, XL_CAN_EV_TAG_TX_OK

**Parameters**

- **canId**
  CAN ID.

- **msgFlags**

  - XL_CAN_RXMSG_FLAG_EDL
    Extended data length.

  - XL_CAN_RXMSG_FLAG_BRS
    Baud rate switch.

  - XL_CAN_RXMSG_FLAG_ESI
    Error state indicator.

  - XL_CAN_RXMSG_FLAG_EF
    Error frame.

  - XL_CAN_RXMSG_FLAG_ARB LOST
    Arbitration lost.

  - XL_CAN_RXMSG_FLAG_RTR
    Remote frame.

  - XL_CAN_RXMSG_FLAG_WAKEUP
    High voltage message on single wire CAN.

  - XL_CAN_RXMSG_FLAG_TE
    1: transceiver error detected.

- **crc**
  Crc of the CAN message.

- **totalBitCnt**
  Number of received bits including stuff bit.

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

- **reserved**
  Internal use.

- **data**
  Data that was received.
### 5.5.5 XL_CAN_EV_ERROR

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

**Tag**
`XL_CAN_EV_TAG_RX_ERROR, XL_CAN_EV_TAG_TX_ERROR`

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

### 5.5.6 XL_CAN_EV_CHIP_STATE

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

**Tag**
`XL_CAN_EV_TAG_CHIP_STATE`

**Parameters**
- **busStatus**
  - Returns the state of the CAN controller. The following codes are possible:
    - `XL_CHIPSTAT_BUSOFF`
      - The bus is offline.
    - `XL_CHIPSTAT_ERROR_PASSIVE`
      - One of the error counters has reached the error level.
    - `XL_CHIPSTAT_ERROR_WARNING`
      - One of the error counters has reached the warning level.
    - `XL_CHIPSTAT_ERROR_ACTIVE`
      - The bus is online.
- **txErrorCounter**
  - Error counter for the transmit section of the CAN controller.
- **rxErrorCounter**
  - Error counter for the receive section of the CAN controller.
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;
```

**Tag**

XL_CAN_EV_TAG_TX_REQUEST

**Parameters**

- **canId**
  CAN ID.

- **msgFlags**
  XL_CAN_RXMSG_FLAG_EDL
  Extended data length.

  XL_CAN_RXMSG_FLAG_BRS
  Baud rate switch.

  XL_CAN_RXMSG_FLAG_esi
  Error state indicator.

  XL_CAN_RXMSG_FLAG_EF
  Error frame.

  XL_CAN_RXMSG_FLAG_ARB_LOST
  Arbitration lost.

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

- **txAttemptConf**
  Reserved.

- **reserved**
  Internal use.

- **data**
  Data that was receive.

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

Tag

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 Ethernet Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>119</td>
</tr>
<tr>
<td>6.2</td>
<td>Network Based Access Mode</td>
<td>125</td>
</tr>
<tr>
<td>6.3</td>
<td>Channel Based Access Mode</td>
<td>150</td>
</tr>
</tbody>
</table>
6.1 Introduction

6.1.1 General Information

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

► **channel-based access**, which was introduced with the VN5610(A) and VN5640 interface generation, and

► **network-based access**, which was introduced with the VN5620 and VN5430 interface generation.

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

Products that supported channel-based access until then also supports network-based access (see section Device Support on page 123). However, the new interface generation VN5620 and VN5430 and following only support network-based access to Ethernet networks.

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

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

The access mode can be switched for supporting device drivers (see section Switching Access Mode on page 123).

The channel-based API will continue to be supported as a legacy API on devices that previously supported it.
6.1.2 Network-Based API vs. Channel-Based API

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

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

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

![Example Topology](Figure 10: Example topology)
With the channel-based access method, the hardware port is the focus of the API. Messages are received or sent on a specific channel.

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

Figure 11: Example of channel-based access
For network-based access, the configuration of the interface hardware defines how all participants communicate with each other and what the communication paths are. The configuration can be used to decide how close it should be to the original topology.

By defining which ports (see definition of terms in Definitions on page 127) are located within a network, the application does not have to care about which physical port the target hardware is connected to.

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

![Diagram of network-based access](image)

Figure 12: Example of network-based access
6.1.3 Device Support

<table>
<thead>
<tr>
<th>Device</th>
<th>channel-based access mode</th>
<th>network-based access mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN5610A</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VN5640</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VN5620</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>VN5430</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VN7640</td>
<td>X</td>
<td>planned</td>
</tr>
<tr>
<td>VX0312</td>
<td>X</td>
<td>planned</td>
</tr>
</tbody>
</table>

6.1.4 Switching Access Mode

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

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

**Modes**

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

**Note**
Only the VN5610(A) and VN5640 support both modes. The new interfaces only support network-based Ethernet configuration. Accordingly, this step-by-step guide applies to VN5610A/VN5640 only.

**Step by Step Procedure**

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

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

4. Wait until mode programming has finished.
6.2 Network Based Access Mode

6.2.1 Basic Concept

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

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

Channel-based access mode

![Diagram showing network setup](image)

Figure 13: Simulation setup of the device firmware before version 11.2
The device firmware from version 11.2 allows free segmentation. Therefore, two switch segments with the associated ports can be defined with this version. Both segments are assigned to the same network **Network 1**. In CANoe, in this setup only one Ethernet network is required in the Simulation setup (network name = **Network 1**).

![Simulation setup of the device firmware from version 11.2](image)

**Figure 14:** Simulation setup of the device firmware from version 11.2
6.2.2 Definitions

Terms

Port
A port is an access point for an application like CANoe, CANape or a real device like an ECU. A distinction is made between a physical port and a virtual port. Each port has a unique name and is assigned to exactly one segment.

Physical Port
Each Vector Ethernet network interface provides a defined number of physical connections. Exactly one physical port is assigned to each physical connection at one point in time. An application can configure physical layer properties by means of the physical port. An application has only read access to a physical port, apart from stress sending.

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

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

Figure 15: Simulation setup of the device driver from version 11.2
Note
Simulation nodes (e. g. CAPL program or interactive generator block) are connected to the Ethernet network by means of virtual ports. The number of virtual ports that can be created is limited by the hardware (e. g. a VN5640 supports up to 32 virtual ports). In most cases, the virtual ports are automatically created by the application and no user interaction is necessary.

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

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

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

► Link Segment
A link segment always connects exactly two ports transparently to each other. The link segment is used to transparently forward Ethernet packets and the states of the physical layer (e. g. link up/down, OPEN Alliance TC10 wake/sleep*).

* only with VN5640 Interface Option 100BASE-T1 (TJA1101)

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

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

- Direct Connection
  Connection of one physical port to one virtual port.

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

Note
The network name is also used to connect applications to the devices. For example, in CANoe the network name can be specified in the System View in the Simulation Setup.
Uplink
An uplink connects the device to a host. Filters can be configured to reduce the data transfer on an uplink.

- Host: Vector Application
  From device driver version 11.2, a USB or Ethernet can be used as uplink to the Vector application.

- Host: Mirroring
  Ethernet packets can be mirrored via a mirror uplink. For example, a data logger can be connected to a mirror port.

6.2.3 General Information

Configuring networks
The XL Driver Library can only open networks that have been previously configured on a Vector device. Therefore, before an XL API application can perform measurement or simulation on a device, you must write an appropriate network configuration to the device using the Vector Hardware Configuration tool. Usage instructions for the tool are available in the tool's integrated help and the VN5000 Ethernet Interface Family Manual.

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

► Step 1: Driver initialization
  - Open a network of a certain type (for example Ethernet). A network can span over multiple devices.

► Step 2: Network setup
  Configuration of the opened network:
  - Open or add virtual ports.
  - Connect to physical ports with measurement points.

► Step 3: On network/measurement tasks
  - Definition of main tasks for Tx and Rx frames (messages).

Note
The network based access mode and the channel based access mode are different XL Driver Library APIs. Do not mix the xlNet*() function calls with xlEth*().
6.2.3.1 Step 1: Driver Initialization

Open a network

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

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

![Diagram of network applications and devices]

**Figure 16: Principle structure of network applications**

Multiple applications

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

**Note**

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

Network handle

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

**Note**

The XLnetworkHandle is different from the XLportHandle used in channel based access mode and has a different meaning.
6.2.3.2 Step 2: Network Setup

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

To transmit a frame directly from a physical port or to retrieve frames from it, a measurement point with the name of the physical port must be connected with the function `xlNetConnectMeasurementPoint()`.

To send a frame into a switch or direct segment, a virtual port must be added with the function `xlNetAddVirtualPort()` or opened with `xlNetOpenVirtualPort()` if the port was already configured with the Vector Hardware Configuration tool.

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

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

All port names must be unique in a network.

**Reference**
For further information on the network setup, please refer to the flowchart on page 133.

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

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

**Receive handle**
If a measurement point is connected or a virtual port is added or opened, the application must assign a receive handle `XLrxHandle` that identifies events originating from this measurement point or virtual port. The receive handles should be assigned in a way that allow the application to uniquely identify the source of an event.
6.2.3.3 Step 3: On Network/Measurement Tasks

Transmitting frames

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

Receiving frames

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

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

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

Calling sequence

Driver Init

- \texttt{xlCreateDriverConfig()}
- \texttt{xlNetEthOpenNetwork()}

Network Setup

- \texttt{xlNetAddVirtualPort()}
- \texttt{xlNetOpenVirtualPort()}
- \texttt{xlNetConnectMeasurementPoint()}
- \texttt{xlNetSetNotification()}
- \texttt{xlNetActivateNetwork()}

On Network

- \texttt{xlNetDeactivateNetwork()}
- \texttt{xlNetCloseNetwork()}

Function calls for Ethernet applications

Figure 17: Function calls for Ethernet applications
6.2.5 Functions

6.2.5.1 xlNetActivateNetwork

Syntax

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

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

Input parameters
- `networkHandle`: Handle to access the network retrieved by `xlNetEthOpenNetwork()`.

Return value
Returns an error code (see section Error Codes on page 482).

6.2.5.2 xlNetAddVirtualPort

Syntax

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

Description
Adds a temporary virtual port to a switch segment with the given switch name on the network, specified by the `networkHandle`. The port name must be unique in the network.

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

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

Return value
Returns an error code (see section Error Codes on page 482).

6.2.5.3 xlNetCloseNetwork

Syntax

```c
XLstatus xlNetCloseNetwork ( 
    XLnetworkHandle networkHandle
)
```
6.2.5.4 xlNetConnectMeasurementPoint

**Syntax**

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

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

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

**Return value**
Returns an error code (see section **Error Codes** on page 482).

6.2.5.5 xlNetDeactivateNetwork

**Syntax**

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

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

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

**Return value**
Returns an error code (see section **Error Codes** on page 482).

6.2.5.6 xlNetEthOpenNetwork

**Syntax**

```c
XLstatus xlNetEthOpenNetwork (  
    const char *pNetworkName,  
    XLNetworkHandle *pNetworkHandle,  
    const char *pAppName,  
    unsigned int accessType,  
    unsigned int queueSize  
)
```
6.2.5.7 xlNetEthReceive

Syntax

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

Description

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

Input parameters

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

- **pEventBuffer**
  Buffer for a single Ethernet event (see section `T_XL_NET_ETH_EVENT` on page 144).

- **pRxHandleCount**
  - Input direction
    Maximum number of receive handles in the `pRxHandle` list.
  - Output direction:
    Actual number of receive handles within the `pRxHandle` list.

- **pRxHandle**
  List of the application specific receive handles set by `xlNetAddVirtualPort()`, `xlNetOpenVirtualPort()` or `xlNetConnectMeasurementPoint()`.
**Return value**

Returns XL_ERR_QUEUE_IS_EMPTY if receive queue is empty.

### 6.2.5.8 xlNetEthRequestChannelStatus

**Syntax**

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

**Description**

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

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

Response event: XL_ETH_CHANNEL_STATUS.

**Input parameters**

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

**Return value**

Returns an error code (see section Error Codes on page 482).

### 6.2.5.9 xlNetEthSend

**Syntax**

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

**Description**

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

**Input parameters**

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

- **ethPortHandle**
  
  Handle to access the virtual port or measurement point.

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

- **pEthTxFrame**
  
  The Ethernet frame to be sent.

**Return value**

Returns XL_ERR_QUEUE_IS_FULL if transmit queue is full.

### 6.2.5.10 xlNetOpenVirtualPort

**Syntax**

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

**Description**

Open a pre-defined virtual port on the network, specified by the networkHandle.
Input parameters
► networkHandle
   Handle to access the network retrieved by xlNetEthOpenNetwork().
► pVPortName
   String for the virtual port name.
► pEthPortHandle
   Return value to access the virtual port - for later function calls.
► rxHandle
   Application specific handle to identify the different events received.

Return value
Returns an error code (see section Error Codes on page 482).

6.2.5.11 xlNetReleaseMACAddress

Syntax

```c
XLstatus xlNetReleaseMACAddress ( 
    XLnetworkHandle networkHandle, 
    const T_XL_ETH_MAC_ADDRESS *pMACAddress 
)
```

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

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

Return value
Returns an error code (see section Error Codes on page 482).

6.2.5.12 xlNetRequestMACAddress

Syntax

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

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

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

Input parameters
► networkHandle
   Handle to access the network retrieved by xlNetEthOpenNetwork().
► pMACAddress
   Pointer structure to retrieve ETH MAC address.

Return value
Returns an error code (see section Error Codes on page 482).
6.2.5.13 xlNetSetNotification

Syntax

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

Description

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

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

Input parameters

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

- **pHandle**
  Generated handle.

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

Return value

Returns an error code (see section Error Codes on page 482).
6.2.6 Structs

6.2.6.1 T_XL_ETH_MAC_ADDRESS

**Syntax**

```c
#define XL_ETH_MACADDR_OCTETS 6

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

**Description**

This struct defines the composition of a MAC address.

**Parameters**

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

6.2.7.1 T_XL_NET_ETH_CHANNEL_STATUS

Syntax

```c
typedef T_XL_ETH_CHANNEL_STATUS
    T_XL_NET_ETH_CHANNEL_STATUS;
```

Description

This event is generated each time the link information changes.

Parameters

For a description of the structure members refer to `T_XL_ETH_CHANNEL_STATUS`.

6.2.7.2 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX

Syntax

```c
typedef T_XL_NET_ETH_DATAFRAME_RX
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX;
```

Description

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

Parameters

For a description of the structure members refer to `T_XL_NET_ETH_DATAFRAME_RX`.

6.2.7.3 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX_ERROR

Syntax

```c
typedef T_XL_NET_ETH_DATAFRAME_RX_ERROR
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX_ERROR;
```

Description

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

Parameters

For a description of the structure members refer to `T_XL_NET_ETH_DATAFRAME_RX_ERROR`.

6.2.7.4 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX

Syntax

```c
typedef T_XL_NET_ETH_DATAFRAME_RX
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX;
```

Description

This event is indicated to the application each time an Ethernet frame has been successfully sent to the network. It is neither a delivery confirmation from the receiver, nor a guarantee that the intended recipient will receive that frame. It currently has an identical layout to the Rx packet; the different name is merely for a better understanding.

Parameters

For a description of the structure members refer to `T_XL_NET_ETH_DATAFRAME_RX`.

6.2.7.5 T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX_ERROR

Syntax

```c
typedef T_XL_NET_ETH_DATAFRAME_RX_ERROR
    T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX_ERROR;
```

Description

This event is indicated to the application each time an Ethernet frame has not been
successfully sent to the network from a measurement point.

Parameters
For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX_ERROR.

6.2.7.6 T_XL_NET_ETH_DATAFRAME_RX

Syntax
typedef struct sXL_net_eth_dataframe_rx {
    unsigned int frameDuration;
    unsigned short dataLen;
    unsigned char reserved1;
    unsigned char reserved2;
    unsigned int errorFlags;
    unsigned int reserved3;
    unsigned int fcs;
    unsigned char destMAC[XL_ETH_MACADDR_OCTETS];
    unsigned char sourceMAC[XL_ETH_MACADDR_OCTETS];
    T_XL_ETH_FRAMEDATA frameData;
} T_XL_NET_ETH_DATAFRAME_RX;

Description
Structure describing the network Ethernet frames that can be received (including Tx frames).

Parameters
► frameDuration
Transmit duration of the frame, given in nanoseconds.

► dataLen
Combined size of etherType and payload in bytes. This specifies the size actually used, not the maximum size of the struct.

► reserved1
Not being used, ignore.

► reserved2
Not being used, ignore.

► errorFlags
In an Rx event the bits indicate following errors:
- XL_ETH_NETWORK_RX_ERROR_INVALID_LENGTH
  Bit 0: Invalid length error. Set when the receive frame has an invalid length as defined by IEEE802.3
- XL_ETH_NETWORK_RX_ERROR_INVALID_CRC
  Bit 1: CRC error. Set when frame is received with CRC-32 error but valid length
- XL_ETH_NETWORK_RX_ERROR_PHY_ERROR
  Bit 2: Corrupted receive frame caused by a PHY error
- XL_ETH_NETWORK_RX_ERROR_MACADDR_ERROR
  Bit 3: Invalid source or destination MAC address

In an Tx event the bits indicate following errors:
- XL_ETH_NETWORK_TX_ERROR_NO_LINK
  Bit 0: No Link
- XL_ETH_NETWORK_TX_ERROR_PHY_NOT_CONFIGURED
  Bit 1: PHY not yet configured
- XL_ETH_NETWORK_TX_ERROR_PHY_BRIDGE_ENABLED
  Bit 2: PHY bypass activated
- XL_ETH_NETWORK_TX_ERROR_CONVERTER_RESET
  Bit 3: RGMII converter in reset
- XL_ETH_NETWORK_TX_ERROR_INVALID_LENGTH
  Bit 4: Invalid length error. Set when the frame has an invalid length as defined by IEEE802.3
- XL_ETH_NETWORK_TX_ERROR_INVALID_CRC
  Bit 5: CRC error. Set when frame is transmitted with CRC-32 error but valid length
- XL_ETH_NETWORK_TX_ERROR_MACADDR_ERROR
  Bit 6: Invalid source or destination MAC address

► reserved3
Not being used, ignore.

► fcs
Frame Check Sequence as received from network.

► destMAC
Destination MAC address.

► sourceMAC
Source MAC address.

► frameData
Section T_XL_ETH_FRAME on page 162

6.2.7.7 T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ACK

Syntax
typedef T_XL_NET_ETH_DATAFRAME_RX
T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ACK;

Description
This event is indicated to the application each time an Ethernet frame has been successfully sent to the network from a virtual port. It is neither a delivery confirmation from the receiver, nor a guarantee that the intended recipient will receive that frame. It currently has an identical layout to the Rx packet; the different name is merely for a better understanding.

Parameters
For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX.

6.2.7.8 T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ERROR

Syntax
typedef T_XL_NET_ETH_DATAFRAME_RX_ERROR
T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ERROR;

Description
This event is indicated to the application each time an Ethernet frame has not been successfully sent to the network from a virtual port.

Parameters
For a description of the structure members refer to T_XL_NET_ETH_DATAFRAME_RX_ERROR.
6.2.7.9 T_XL_NET_ETH_EVENT

Syntax

```c
typedef unsigned short XLethEventTag;

typedef struct s_xl_net_eth_event {
    unsigned int size;
    XLethEventTag tag;
    unsigned short channelIndex;
    unsigned int userHandle;
    unsigned short flagsChip;
    unsigned short reserved;
    XLuint64 reserved1;
    XLuint64 timestampSync;

    union s_xl_eth_net_tag_data {
        unsigned char rawData[XL_ETH_EVENT_SIZE_MAX];
        T_XL_NET_ETH_DATAFRAME_RX frameSimRx;
        T_XL_NET_ETH_DATAFRAME_RX_ERROR frameSimRxError;
        T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ACK frameSimTxAck;
        T_XL_NET_ETH_DATAFRAME_SIMULATION_TX_ERROR frameSimTxError;
        T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX frameMeasureRx;
        T_XL_NET_ETH_DATAFRAME_MEASUREMENT_RX_ERROR frameMeasureRxError;
        T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX frameMeasureTx;
        T_XL_NET_ETH_DATAFRAME_MEASUREMENT_TX_ERROR frameMeasureTxError;
        T_XL_NET_ETH_CHANNEL_STATUS channelStatus;
    } tagData;
} T_XL_NET_ETH_EVENT;
```

Description

Structure describing the network Ethernet events that can be received (including Tx events).

Parameters

- **size**
  Size of the complete Ethernet event, including header and payload data.

- **tag**
  Specifies the structure that is applied to `tagData`, e. g. XL_ETH_EVENT_TAG_FRAMERX_MEASUREMENT.

- **channelIndex**
  Logical channel number where this event originated or is target to.

- **userHandle**
  Application-specific handle that may be used to link associated events, e. g. a transmit confirmation to the original send request. Not used (set to 0) for indications not related to a request.
**flagsChip**

The lower 8 bit contain chip information:
- Bit 0: XL_ETH_CONNECTOR_RJ45
- Bit 1: XL_ETH_CONNECTOR_DSUB
- Bit 2: XL_ETH_PHY_IEEE
- Bit 3: XL_ETH_PHY_BROADR
- Bit 4: XL_ETH_FRAME_BYPASSED
- Bit 5..7: unused

The upper 8 bit contain special flags:
- Bit 8: XL_ETH_QUEUE_OVERFLOW

Not all events generated by the device could be indicated to the application.

- Bit 9..14: unused
- Bit 15: XL_ETH_BYPASS_QUEUE_OVERFLOW

Indicates that one or more received packets could not be sent to the opposite bus in MAC bypass mode.

**reserved**

Not being used, ignore.

**reserved1**

Not being used, ignore.

**timestampSync**

Synchronized time stamp with 1 ns resolution (PC → device) and an accuracy of 8 µs. Time synchronization is applied if enabled in Vector Hardware Control Panel.

**tagData**

See structures on page 142 ... page 141 for further details.
6.2.8 Application Examples

6.2.8.1 xlNetEthDemo

General Information

Description
The example xlNetEthDemo (further noted as app) demonstrates how to transmit/receive Ethernet frames using network-based mode. The app structure contains a small command line interface controlled by keyboard commands.

Starting the example
Before running the app, an Ethernet device must be connected that operates in network-based mode. Additionally, at least one network must be configured on the device with the Vector Hardware Configuration tool.

Example Test Case

Startup
At execution, the app searches for all attached devices, whose drivers have been installed. Therefore, information shown in Vector Hardware Configuration application, will be available through this Demo too (see xlCreateDriverConfig()).

As soon as the app obtains hardware information from Vector Driver, you will see following printout:

```
- xlNetEthDemo - Test Application for XL Family Driver API -
(C) 2020 Vector Informatik GmbH

Found 4 devices
[0] Found UN5640:1<000002> device
[1] Found UN5640:2<005247> device
[3] Found ISService:1<000000> device
Choose Device for Showing Hardware Configuration:
```

Figure 18: Found Vector hardware

Configurations of all devices with Ethernet ports can be printed out in console (see command <w>)- as is shown in the following picture. Hardware information (transceiver, hardware ports etc.) is related to the information about a device. However, this set of information about devices does not contain direct information about available networks on the selected device, but network information is a separate set of information.

```
<table>
<thead>
<tr>
<th>Device: UN5640</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Hardware Configuration</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UN5640:1&lt;000002&gt;</td>
<td>Transceiver</td>
<td>MNC</td>
<td>Link State</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>BCM97611</td>
<td>00:16:81:48:01:10:1000000</td>
<td>Link down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 19: Example of device hardware configuration
As depicted below, the list of all networks, from all connected devices, contains information about each network: name, segment(s), segment name, segment type, measurement points (MPs) and virtual ports (VPs) connected to corresponding segment.

Adding a VP

![Network Configuration Table]

Figure 20: Network and segment configuration information

The app will at this point, for demonstration purposes, manually add a virtual port with the name VP_manually_added. In order to do so, the user firstly needs to choose a network from the list by typing the ID value shown in the first column (see figure above). In the example case, ID = 1 is chosen (Test Example Network). After choosing the network, only the configuration of the chosen network will be displayed — showing segments (ID, name, type) that are within the chosen network. At this point the user needs to choose the segment to which the VP will be manually added.

If the segment is of type DIRECT and it already contains a VP, or of type TAP, the app shows an error that the VP has not been added, as the chosen segment is already full. If the user selects a segment that is not full, manually adding a VP will succeed. Subsequently, the app displays the change in the configuration of the chosen network. In order to detect the added VP, the app calls xlCreateDriverConfig() again after adding the VP, so it gets new up-to-date snapshot of the network configuration.

Main menu

At this point, the initialization phase is over and the interaction menu is shown. The user is prompted by the main menu about further actions (see section Keyboard Commands on page 149). The <t> and <T> keys trigger the transmission of Ethernet frames via MP or VP with a dummy IP and UDP header. The user can set the destination MAC and IP with the <m> and <i> keys. By pressing <G>, the user can acquire a unique source MAC from the pool of MAC addresses available in the device. The source IP of the dummy frame is 192.168.0.x, where x is the port ID. MP port as a sender is chosen by typing the port ID or by <+> or <->. VP as a sender is chosen by typing <+> or </>. After sending the Ethernet frame, received events will be processed and a corresponding message is displayed (see example function handleEthernetEvent).
Events

The testbed setup for this example is depicted below. At this point, the user has chosen to send an Ethernet frame from VP_manually_added. The resulting frame path is marked with arrows starting with number 1.

Figure 21: Testbed used for the example and Frame Path from VP_manually_added port

The app prints the following event output with the rxHandle that the app assigned in parenthesis.

► Received TX Ack on VP_manuallyAdded (4)
  Confirmation that frame was sent (step 1 in figure above).

► Received TX on MP01 (0)
  MP01 forwards the frame (step 2).

► Received RX on MP02 (2)
  MP01 and MP11 are connected by cable (violett connection) and thus MP11 receives the frame (step 3) and MP12 forwards the frame (step 4) to MP02 (step 5). We only receive an event form MP02 but not from MP11 and MP12, because the app only opened the Test_Example_Network and not the MirrorLink network.

► Received RX on VP02 (1)
  As MP02, MP03 and VP02 belong to the same segment, the switch forwards the frame from MP02 to VP02 (step 6.1).
Received TX on MP03 (3)
The switch also forwards the frame to MP03 (step 6.2), which transmits it to the other end of cable (step 7).

Keyboard Commands

The running app can be controlled by the following keyboard commands:

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1&gt;…(max MPs)</td>
<td>Select specific MP in the chosen network by giving its ID (1 &lt; MP_ID &lt; 9).</td>
</tr>
<tr>
<td>&lt;+&gt;</td>
<td>Select next MP in the chosen network.</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>Select previous MP in the chosen network.</td>
</tr>
<tr>
<td>&lt;t&gt;</td>
<td>Transmit Ethernet frame from currently chosen MP.</td>
</tr>
<tr>
<td>&lt;*&gt;</td>
<td>Select next VP in the chosen network.</td>
</tr>
<tr>
<td>&lt;/&gt;</td>
<td>Select previous VP in the chosen network.</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
<td>Transmit Ethernet frame from currently chosen VP.</td>
</tr>
<tr>
<td>&lt;s&gt;</td>
<td>Check current link status on all channels in the chosen network.</td>
</tr>
<tr>
<td>&lt;p&gt;</td>
<td>Print basic information from/about last received Ethernet frame in the chosen network Info: {SrcMAC, DstMAC, SrcIP, DstIP, Payload}.</td>
</tr>
<tr>
<td>&lt;G&gt;</td>
<td>Request to obtain MAC address from the device MAC address pool.</td>
</tr>
<tr>
<td>&lt;R&gt;</td>
<td>Release previously requested MAC address from the device MAC address pool.</td>
</tr>
<tr>
<td>&lt;m&gt;</td>
<td>Set destination MAC address.</td>
</tr>
<tr>
<td>&lt;i&gt;</td>
<td>Set destination IP address.</td>
</tr>
<tr>
<td>w&gt;</td>
<td>Show hardware configuration of the chosen device (not related to chosen network and segment).</td>
</tr>
<tr>
<td>&lt;h&gt; / &lt;?&gt;</td>
<td>Show/print main menu.</td>
</tr>
<tr>
<td>&lt;q&gt;</td>
<td>Quit xlNetEthDemo app.</td>
</tr>
</tbody>
</table>
6.3 Channel Based Access Mode

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

For new projects, we recommend the network-based access mode API to access Ethernet networks (see section Network Based Access Mode on page 125).

**Description**
The XL Driver Library enables the development of Ethernet applications for supported Vector devices see section System Requirements on page 34 and also section Device Support on page 123.

Depending on the channel property **init access** (see page 31), 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.
6.3.1 Flowchart

Calling sequence

Driver Init

Channel Setup

On Bus

Figure 22: Function calls for Ethernet applications
6.3.2 Functions

6.3.2.1 xlEthSetConfig

Syntax

```c
XLstatus xlEthSetConfig ( 
  XLportHandle portHandle, 
  XLaccess accessMask, 
  XLuserHandle userHandle, 
  const T_XL_ETH_CONFIG *config 
)
```

Description

Configures basic Ethernet settings. The result of the operation is reported via a T_XL_ETH_CONFIG_RESULT event. This function needs init access.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 159).

Return value

Returns an error code (see section Error Codes on page 482).

6.3.2.2 xlEthGetConfig

Syntax

```c
XLstatus xlEthGetConfig ( 
  XLportHandle portHandle, 
  XLaccess accessMask, 
  XLuserHandle userHandle, 
  const T_XL_ETH_CONFIG *config 
)
```

Description

Reads the basic Ethernet settings from the device that was configured last. Note that the device does not keep those settings after a restart. This is a synchronous operation.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **userHandle**
  The handle is created by the application and is used for the event assignment.
6.3.2.3 xlEthSetBypass

Syntax

XLstatus xlEthSetBypass (  
  XLportHandle portHandle,  
  XLaccess accessMask,  
  XLuserHandle userHandle,  
  unsigned int mode  
)

Description

The function sets the bypass mode for the channel specified in `accessMask`. For the given channel `init access` is required; for the bypass partner channel `init access` is required whenever the channel is currently used by any application.

When the PHY bypass mode is set, two Ethernet channels are internally hard-wired. This requires compatible settings (i.e. same speed, same duplex mode). Sending on either channel is not supported in this mode. The main purpose of this mode is to convert the physical layer from IEEE802.3 to BroadR-Reach and vice versa, with minimal impact on latency. The PHY bypass mode can also be used for monitoring with low latencies.

When in MAC bypass mode, the device connects two channels on 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.

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 24: 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **userHandle**
The handle is created by the application and is used for the event assignment.
6 Ethernet Commands

- **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 482).

### 6.3.2.4 xlEthTransmit

**Syntax**

```c
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 `accessMask`.

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

### 6.3.2.5 xlEthReceive

**Syntax**

```c
XLstatus xlEthReceive (  
    XLportHandle portHandle,  
    T_XL_ETH_EVENT *eventBuffer
)
```

**Description**
Retrieves one event from the event queue. This operation is synchronous.

**Input parameters**

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

- **eventBuffer**
  Buffer for a single Ethernet event (see section `T_XL_ETH_EVENT` on page 162).
6 Ethernet Commands

Return value

Returns an error code (see section Error Codes on page 482).

6.3.2.6 xlEthTwinkleStatusLed

Syntax

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

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 accessMask.

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

Return value

Returns an error code (see section Error Codes on page 482).
6.3.3 Structs

6.3.3.1 XLdriverConfig

**Syntax**

```c
typedef struct {
    unsigned int busType;
    union {
        struct {
            [...]  // can, ...
        } 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 {
    [...]  // busParams, ...
    XLbusParams busParams;
    [...]  // channelConfig
} XLchannelConfig;

typedef struct s_xl_driver_config {
    [...]  // channelConfig, 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.
6.3.3.2 T_XL_ETH_CONFIG

**Syntax**

```c
typedef struct {
    unsigned int speed;
    unsigned int duplex;
    unsigned int connector;
    unsigned int phy;
    unsigned int clockMode;
    unsigned int mdiMode;
    unsigned int brPairs;
} T_XL_ETH_CONFIG;
```

**Parameters**

- **speed**
  Specifies the desired channel bandwidth:
  - `XL_ETH_MODE_SPEED_AUTO_100`  
    100Base-TX only, enable auto-negotiation.
  - `XL_ETH_MODE_SPEED_AUTO_1000`  
    1000Base-T only, enable auto-negotiation.
  - `XL_ETH_MODE_SPEED_AUTO_100_1000`  
    100Base-TX or 1000Base-T, enable auto-negotiation.
  - `XL_ETH_MODE_SPEED_FIXED_100`  
    100Base-TX or 100Base-T1, no auto-negotiation.
  - `XL_ETH_MODE_SPEED_FIXED_1000`  
    1000Base-T1, no auto-negotiation.

- **duplex**
  Specifies the duplex mode for this channel:
  - `XL_ETH_MODE_DUPLEX_DONT_CARE`  
    Used for BroadR-Reach, since only full duplex available for BR!
  - `XL_ETH_MODE_DUPLEX_AUTO`  
    Requires auto-negotiation; only full duplex supported!
  - `XL_ETH_MODE_DUPLEX_FULL`

- **connector**
  Selects the connector to use for this channel:
  - `XL_ETH_MODE_CONNECTOR_DONT_CARE`  
    Should be used on devices other than VN5610(A).
  - `XL_ETH_MODE_CONNECTOR_DSUB`
  - `XL_ETH_MODE_CONNECTOR_RJ45`
Two different physical layers are supported on the VN5600 Interface Family: IEEE802.3 ("standard" Ethernet) and BroadR-Reach.

- XL_ETH_MODE_PHY_DONT_CARE
- XL_ETH_MODE_PHY_IEEE_802_3
  Only available on RJ-45 connector.
- XL_ETH_MODE_PHY_BROADR_REACH

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.

Medium-dependent interface mode (i.e. the assignment of transmit/receive wires on the connector):

- XL_ETH_MODE_MDI_AUTO
  Auto-MDIX detection.

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.

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.

### Valid configuration combinations

<table>
<thead>
<tr>
<th>Rj45</th>
<th>Configurations for the RJ-45 connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed</td>
<td>duplex</td>
</tr>
<tr>
<td>AUTO_100</td>
<td>AUTO</td>
</tr>
<tr>
<td>AUTO_100_1000</td>
<td>AUTO</td>
</tr>
<tr>
<td>AUTO_1000</td>
<td>AUTO</td>
</tr>
<tr>
<td>FIXED_100</td>
<td>FULL</td>
</tr>
</tbody>
</table>

**Note**
Connector should be set to XL_ETH_MODE_CONNECTOR_DONT_CARE on devices other than VN5610(A).

<table>
<thead>
<tr>
<th>D-SUB9</th>
<th>Configurations for the RJ-45 connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed</td>
<td>duplex</td>
</tr>
<tr>
<td>FIXED_100</td>
<td>DON'T CARE</td>
</tr>
</tbody>
</table>
Note
On the D-SUB connector, only a single cable pair is provided per channel. Connector should be set to XL_ETH_MODE_CONNECTOR_DONT_CARE on devices other than VN5610(A).

<table>
<thead>
<tr>
<th>speed</th>
<th>duplex</th>
<th>phy</th>
<th>clockMode</th>
<th>mdiMode</th>
<th>brPairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIXED_100</td>
<td>DON'T CARE</td>
<td>DON'T CARE</td>
<td>MASTER/SLAVE</td>
<td>AUTO</td>
<td>1PAIR</td>
</tr>
<tr>
<td>FIXED_1000</td>
<td>DON'T CARE</td>
<td>DON'T CARE</td>
<td>MASTER/SLAVE</td>
<td>AUTO</td>
<td>1PAIR</td>
</tr>
</tbody>
</table>

Note
Connector should be set to XL_ETH_MODE_CONNECTOR_DONT_CARE on devices other than VN5610(A).
6.3.4 Events

6.3.4.1 T_XL_ETH_FRAME

**Syntax**

typedef union s_xl_eth_framedata {
    unsigned char  rawData[XL_ETH_RAW_FRAME_SIZE_MAX];
    T_XL_ETH_FRAME  ethFrame;
} T_XL_ETH_FRAMEDATA;

typedef struct s_xl_eth_frame {
    unsigned short  etherType;
    unsigned char  payload[XL_ETH_PAYLOAD_SIZE_MAX];
} T_XL_ETH_FRAME;

**Description**
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.

6.3.4.2 T_XL_ETH_EVENT

**Syntax**

typedef unsigned short XLibEventTag;

typedef struct s_xl_eth_event {
    unsigned int  size;
    XLibEventTag  tag;
    unsigned short channelIndex;
    unsigned int  userHandle;
    unsigned short flagsChip;
    unsigned short reserved;
    XLibuint64   reserved1;
    XLibuint64   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_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_ERROR_OTHERAPP frameTxErrorOtherApp;
        T_XL_ETH_DATAFRAME_TX_ERROR_SW frameTxErrorSw;
        T_XL_ETH_CONFIG_RESULT configResult;
        T_XL_ETH_LOSTEVENT  lostEvent;
        T_XL_ETH_CHANNEL_STATUS channelStatus;
        XL_SYNC_PULSE_EV syncPulse;
    } tagData;
} T_XL_ETH_EVENT;

**Description**
Structures describing the Ethernet events that can be received (including Tx events).

**Parameters**

- **size**
  Size of the complete Ethernet event, including header and payload data.
tag
Specifies the structure that is applied to tagData, e. g. XL_ETH_EVENT_TAG_FRAMERX.

channelIndex
Logical channel number where this event originated or is target to.

userHandle
Application-specific handle that may be used to link associated events, e. g. a transmit confirmation to the original send request. Not used (set to 0) for indications not related to a request.

flagsChip
The lower 8 bit contain chip information:
Bit 0: XL_ETH_CONNECTOR_RJ45
Bit 1: XL_ETH_CONNECTOR_DSUB
Bit 2: XL_ETH_PHY_IEEE
Bit 3: XL_ETH_PHY_BROADR
Bit 4: XL_ETH_FRAME_BYPASSED
Bit 5..7: unused

The upper 8 bit contain special flags:
Bit 8: XL_ETH_QUEUE_OVERFLOW

Not all events generated by the device could be indicated to the application.

Bit 9..14: unused
Bit 15: XL_ETH_BYPASS_QUEUE_OVERFLOW

Indicates that one or more received packets could not be sent to the opposite bus in MAC bypass mode.

reserved
Not being used, ignore.

reserved1
Not being used, ignore.

timestampSync
Synchronized time stamp with 1 ns resolution (PC → device) and an accuracy of 8 µs. Time synchronization is applied if enabled in Vector Hardware Control Panel. Offset correction is possible with xlResetClock.

tagData
See structures on page 163 … page 169 for further details.

6.3.4.3 T_XL_ETH_DATAFRAME_RX

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;
}
6.3.4.4 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_FRAMERX_ERROR

Parameters
- **frameldentifier**
  Unique identifier assigned during receive. Used to correlate a later Tx event (in case of MAC bypass) to the Rx event.

```c
typedef struct s_xl_eth_dataframe_rxerror {
  unsigned int frameldentifier;
  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;
```
► **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.

### 6.3.4.5 T_XL_ETH_DATAFRAME_TX_EVENT

#### Syntax
```c
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
- **frameIdentifier**
Unique identifier assigned by the device. For packets sent by the Bypass feature this matches the respective element of the original Rx event.

- **flags**
Transmit flags requested by the application and processed by the device. See `xlEthTransmit()` for a description of allowed flags.
6.3.4.6 T_XL_ETH_DATAFRAME_TXACK

**Syntax**
```c
typedef T_XL_ETH_DATAFRAME_TX_EVENT T_XL_ETH_DATAFRAME_TXACK;
```

**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**
XL_ETH_EVENT_TAG_FRAMETX_ACK

**Parameters**
For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT.

6.3.4.7 T_XL_ETH_DATAFRAME_TXACK_OTHERAPP

**Syntax**
```c
typedef T_XL_ETH_DATAFRAME_TX_EVENT T_XL_ETH_DATAFRAME_TXACK_OTHERAPP;
```

**Description**
This event indicates the successful sending of an Ethernet frame by another application.

**Tag**
XL_ETH_EVENT_TAG_FRAMETX_ACK_OTHER_APP

**Parameters**
For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT.

6.3.4.8 T_XL_ETH_DATAFRAME_TXACK_SW

**Syntax**
```c
typedef T_XL_ETH_DATAFRAME_TX_EVENT T_XL_ETH_DATAFRAME_TXACK_SW;
```
Description

This event is indicated to the application each time a received Ethernet 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_FRAMETX_ACK_SWITCH

Parameters

For a description of the structure members refer to T_XL_ETH_DATAFRAME_TX_EVENT.

6.3.4.9 T_XL_ETH_DATAFRAME_TX_ERROR

Syntax

typedef struct s_xl_eth_dataframe_txerror {
    unsigned int  errorType;
    T_XL_ETH_DATAFRAME_TX_EVENT txFrame;
} T_XL_ETH_DATAFRAME_TX_ERROR;

Description

This event is indicated to the application each time an Ethernet frame has not been sent to the bus.

Tag

XL_ETH_EVENT_TAG_FRAMETX_ERROR

Parameters

► errorType
  Indicates the kind of transmission error and can be one of the following values:
  
  XL_ETH_TX_ERROR_BYPASS_ENABLED
  Bypass enabled.

  XL_ETH_TX_ERROR_NO_LINK
  No link established.

  XL_ETH_TX_ERROR_PHY_NOT_CONFIGURED
  PHY not yet configured.

► txFrame
  section T_XL_ETH_DATAFRAME_TX_EVENT on page 165

6.3.4.10 T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP

Syntax

typedef T_XL_ETH_DATAFRAME_TX_ERROR
T_XL_ETH_DATAFRAME_TX_ERR_OTHERAPP;

Description

This event is indicated to the application each time an erroneous Ethernet 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.
6.3.4.11 T_XL_ETH_DATAFRAME_TX_ERR_SW

**Syntax**

```c
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`.

6.3.4.12 T_XL_ETH_CONFIG_RESULT

**Syntax**

```c
struct s_xl_eth_config_result {
    unsigned int result;
} T_XL_ETH_CONFIG_RESULT;
```

**Description**

This event is generated when a configuration change via `xlEthSetConfig()` was triggered.

**Tag**

`XL_ETH_EVENT_TAG_CONFIGRESULT`

**Parameters**

- `result`
  - 0: Valid parameter combination set via `xlEthSetConfig()`.
  - `!= 0`: Invalid parameter combination set via `xlEthSetConfig()`.

6.3.4.13 T_XL_ETH_LOSTEVENT

**Syntax**

```c
typedef struct s_xl_eth_lostevent {
    XLethEventTag eventTypeLost;
    unsigned short reserved;
    unsigned int reason;
    union {
        struct {
            unsigned int frameIdentifier;
            unsigned int fcs;
            unsigned char sourceMAC[XL_ETH_MACADDR_OCTETS];
            unsigned char reserved[2];
        } txAck, txAckSw;
        struct {
            unsigned int errorType;
            unsigned int frameIdentifier;
            unsigned int fcs;
            unsigned char sourceMAC[XL_ETH_MACADDR_OCTETS];
            unsigned char reserved[2];
        } txError, txErrorSw;
        unsigned int reserved[20];
    } eventInfo;
} T_XL_ETH_LOSTEVENT;
```

**Description**

This event is generated when the driver detects a regular event that could not be indic-
ated to the application (e. g. not all data available).

**Tag**

XL_ETH_EVENT_TAG_LOSTEVENT

**Parameters**

See respective regular events.

### 6.3.4.14 T_XL_ETH_CHANNEL_STATUS

**Syntax**

```c
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_DUPLEXUNKNOWN
  XL_ETH_STATUS_DUPEXFULL

- **mdiMode**
  
  The active MDI state:
  
  XL_ETH_STATUS_MDI_UNKNOWN
  XL_ETH_STATUS_MDI_STRAIGHT MDI.
  XL_ETH_STATUS_MDI_CROSSOVER MDI-X.


**activeConnector**
The interface connector currently assigned to the MAC:
- XL_ETH_STATUS_CONNECTOR_DEFAULT
- XL_ETH_STATUS_CONNECTOR_RJ45
- XL_ETH_STATUS_CONNECTOR_DSUB

**activePhy**
The currently active transmitter (physical interface):
- XL_ETH_STATUS_PHY_UNKNOWN
- XL_ETH_STATUS_PHY_802_3
- XL_ETH_STATUS_PHY_BROADR_REACH
- XL_ETH_STATUS_PHY_100BASE_T1
- XL_ETH_STATUS_PHY_1000BASE_T1

**clockMode**
Clock mode setting of the connection:
- XL_ETH_STATUS_CLOCK_DONT_CARE
  Reported for IEEE 802.3.
- XL_ETH_STATUS_CLOCK_MASTER
- XL_ETH_STATUS_CLOCK_SLAVE

**brPairs**
The number of cable pairs used for the link:
- XL_ETH_STATUS_BR_PAIR_DONT_CARE
  Reported for IEEE 802.3.
- XL_ETH_STATUS_BR_PAIR_1PAIR

---

### 6.3.4.15 T_XL_ETH_DATAFRAME_TX

**Syntax**
```c
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**

- **framIdentifier**
  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 162
6.3.5 Application Examples

6.3.5.1 xlEthDemo

General Information

Description
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 <b>. To transmit a complete file via Ethernet use the command line options to start the example (/t).

Keyboard Commands

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1&gt;…(max eth channels)</td>
<td>Select an Ethernet channel.</td>
</tr>
<tr>
<td>&lt;+&gt;</td>
<td>Select next Ethernet channel.</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>Select previous Ethernet channel.</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>Activate current channel.</td>
</tr>
<tr>
<td>&lt;d&gt;</td>
<td>Deactivate current channel.</td>
</tr>
<tr>
<td>&lt;c&gt;</td>
<td>Set channel configuration.</td>
</tr>
<tr>
<td>&lt;t&gt;</td>
<td>Transmit single packet.</td>
</tr>
<tr>
<td>&lt;b&gt;</td>
<td>Start burst transmission (needs active receiver).</td>
</tr>
<tr>
<td>&lt;s&gt;</td>
<td>Stop burst transmission.</td>
</tr>
<tr>
<td>&lt;r&gt;</td>
<td>Receive.</td>
</tr>
<tr>
<td>&lt;e&gt;</td>
<td>Set Ether type to use.</td>
</tr>
<tr>
<td>&lt;p&gt;</td>
<td>Set packet payload size.</td>
</tr>
<tr>
<td>&lt;l&gt;</td>
<td>Set burst data length.</td>
</tr>
</tbody>
</table>
### Ethernet Commands

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;m&gt;</td>
<td>Set receiver MAC address.</td>
</tr>
<tr>
<td>&lt;k&gt;</td>
<td>Twinkle status LED of device.</td>
</tr>
<tr>
<td>&lt;w&gt;</td>
<td>Show driver configuration.</td>
</tr>
<tr>
<td>&lt;v&gt;</td>
<td>Toggle verbose output.</td>
</tr>
</tbody>
</table>

### Command Line Interface

The following command line options are available:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/h or /?</td>
<td>This help.</td>
</tr>
<tr>
<td>/dn</td>
<td>XLAPI Ethernet device channel n to use (n = 1,2,...).</td>
</tr>
<tr>
<td>/cX</td>
<td>Use channel configuration mode X.</td>
</tr>
<tr>
<td>/t</td>
<td>Transmit test pattern.</td>
</tr>
<tr>
<td>/t&quot;Name&quot;</td>
<td>Transmit content of file.</td>
</tr>
<tr>
<td>/r</td>
<td>Receive data.</td>
</tr>
<tr>
<td>/r&quot;Name&quot;</td>
<td>Receive data and write to file; the file must not exist.</td>
</tr>
<tr>
<td>/eX,Y</td>
<td>Use Ether type X for transmission, Y for acknowledge.</td>
</tr>
<tr>
<td>/pX</td>
<td>Maximum transmit packet payload in bytes (42...1500).</td>
</tr>
<tr>
<td>/lX</td>
<td>Maximum transmit length (0=no limit/file size).</td>
</tr>
<tr>
<td>/mX</td>
<td>Receiver MAC address X (format: aa:bb:cc:dd:ee:ff).</td>
</tr>
<tr>
<td>/oX</td>
<td>Transmit/receive timeout in milliseconds (0=Disable timeout).</td>
</tr>
<tr>
<td>/v</td>
<td>Verbose output.</td>
</tr>
<tr>
<td>/w</td>
<td>Twinkle status LED of device owning the given XLAPI channel and exit.</td>
</tr>
<tr>
<td>/q</td>
<td>Quit after transmit/receive.</td>
</tr>
</tbody>
</table>
6.3.5.2 xlEthBypassDemo

Description

The bypass demo is a small command-line tool that shows how to configure a Vector Ethernet device, activate a channel bypass and how to receive data indications. The device can be started without arguments; in this case, a default operation is being used. For a list of the possible command-line arguments, run the tool with a “/?” argument.
7 LIN Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>176</td>
</tr>
<tr>
<td>7.2 Flowchart</td>
<td>177</td>
</tr>
<tr>
<td>7.3 LIN Basics</td>
<td>178</td>
</tr>
<tr>
<td>7.4 Functions</td>
<td>179</td>
</tr>
<tr>
<td>7.5 Structs</td>
<td>185</td>
</tr>
<tr>
<td>7.6 Events</td>
<td>186</td>
</tr>
<tr>
<td>7.7 Application Examples</td>
<td>189</td>
</tr>
</tbody>
</table>
7.1 Introduction

The XL Driver Library enables the development of LIN applications for supported Vector devices (see section System Requirements on page 34). A LIN application always requires init access (see section xlOpenPort on page 44). Multiple LIN applications cannot use a common physical LIN channel at the same time.

Depending on the channel property init access (see page 31), 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.
### 7.2 Flowchart

**Calling sequence**

**Driver Init**
- `xlOpenDriver()`
- `xlGetDriverConfig()`
- `xlOpenPort()`

**Channel Setup**
- `xlGetChannelMask()`
- `xlOpenPort()`
- `xlGetDriverConfig()`
- `xlGetApplConfig()`
- `xlGetChannelIndex()`

**On Bus**
- `xlActivateChannel()`
- `xlDeactivateChannel()`
- `xlFlushReceiveQueue()`
- `xlGetReceiveQueueLevel()`
- `xlLinSwitchSlave()`
- `xlLinSetChecksum()`
- `xlLinSetDLC()`
- `xlLinSendRequest()`
- `xlLinSetSlave()`

**Function calls**
- `Function()` (Special LIN API function)
- `Function()` (Common API function)

**Flowchart Description**
- Start
- `xlOpenDriver()`
- `xlGetAppConfig()`
- `xlGetChannelMask()`
- `xlOpenPort()`
- `xlGetDriverConfig()`

**Setup LIN Master**
- `xlLinSetSlave` for each slave

**Setup LIN Slave**
- `xlLinSetSlave`
- `xlLinSetDLC`

**Figure 25:** Function calls for LIN applications
7.3 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.

**Functional principle**
The LIN network is based on a master-slave architecture where the LIN master is one privileged node of the LIN network. The master consists of a master task as well as a slave task, while the slaves only comprise a slave task. The LIN master task controls slave tasks by sending special patterns called headers on the bus at times defined within a so called schedule table. Such a header contains a message address and can be viewed as a request to be responded to by one LIN slave task. The total of header plus slave task response is called a LIN message. All other slaves can either receive the LIN message or ignore it.

**LIN messages**
Generally, there are 62 identifiers i.e. LIN messages possible within a LIN2.x network, two of which (60 and 61) are dedicated to diagnostics on LIN (see `xlLinSetDLC()`). A response can contain up to eight data bytes (defined for each slave, see `xlLinSetSlave()`).

**XL API**
The XL API comprises functions for the LIN master as well as the LIN slaves, allowing sending and receiving messages on the LIN bus with any Vector XL Interface. If using the XL API for the master, be sure to have it defined via `xlLinSetChannelParams()` with Master flag. Furthermore, the XL API can be simultaneously used for LIN slaves, which must be configured separately via `xlLinSetChannelParams()` (Slave flag), `xlLinSetDLC()`, `xlLinSetChecksum()` and `xlLinSetSlave()`. See the LIN flowchart and the provided LIN examples for further details.
7.4 Functions

7.4.1 xlLinSetChannelParams

**Syntax**

```c
XLstatus xlLinSetChannelParams ( 
    XLportHandle portHandle, 
    XLaccess accessMask, 
    XLlinStatPar statPar)
```

**Description**
Sets the channel parameters like baud rate, master, slave.

**Note**
The function opens all acceptance filters for LIN. In other words, the application receives XL_LIN_MSG events for all LIN IDs. Resets all DLC’s (xlLinSetDLC())!

**Input parameters**
- **portHandle**
The port handle retrieved by xlOpenPort().
- **accessMask**
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **statPar**
Defines the mode of the LIN channel and the baud rate (see section XLlinStatPar on page 185).

**Return value**
Returns an error code (see section Error Codes on page 482).

7.4.2 xlLinSetDLC

**Syntax**

```c
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 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 482).

Example

Setting DLC for LIN message with ID 0x04 to 8 and for all other IDs to undefined.

```c
unsigned char DLC[64];
for (int i=0;i<64;i++) DLC[i] = XL_LIN_UNDEFINDED_DLC;
DLC[4] = 8;
xlStatus = xlLinSetDLC(m_XLportHandle,
    m_xiChannelMask[MASTER],
    DLC);
```

7.4.3  xlLinSetChecksum

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).
### Example

Setting the checksum for a LIN message with the ID 0x04 to "enhanced" and for all other IDs to "undefined"

```c
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);
```

### 7.4.4 `xlLinSetSlave`

#### Syntax

```c
XLstatus xlLinSetSlave (  
    XlPortHandle portHandle,  
    XLAccess accessMask,  
    unsigned char linId,  
    unsigned char data[8],  
    unsigned char dlc,  
    unsigned short checksum)
```

#### Description

Sets up a LIN slave. This function must be called before activating a channel and for each slave ID separately. After activating the channel it is only possible to change the data, dlc and checksum but not the linID.

This function is also used to setup a slave task within a master node. If the function is not called but activated the channel is only listening.

#### Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

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);

7.4.5 xlLinSwitchSlave

Syntax

XLstatus xlLinSwitchSlave (  
    XLportHandle  portHandle,
    XLaccess      accessMask,
    unsigned char linId,
    unsigned int  mode)

Description

The function can switch on/off a LIN slave during measurement.

Input parameters

► portHandle
The port handle retrieved by xlOpenPort().

► accessMask
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).
7.4.6 xlLinSendRequest

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 482).

7.4.7 xlLinWakeUp

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

Return value

Returns an error code (see section Error Codes on page 482).

7.4.8 xlLinSetSleepMode

Syntax

```c
XLstatus xlLinSetSleepMode (   
  XLportHandle  portHandle,   
  XLaccess      accessMask)
```
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).
## 7.5 Structs

### 7.5.1 XLlinStatPar

#### Syntax

```c
typedef struct {
    unsigned int LINMode;
    int           baudrate;
    unsigned int LINVersion;
    unsigned int reserved;
} XLlinStatPar;
```

#### Parameters

- **LINMode**
  - Sets the channel mode.
  
  - `XL_LIN_MASTER`
    - Set channel to a LIN master.
  
  - `XL_LIN_SLAVE`
    - Set channel to LIN slave.

- **baudrate**
  - Sets the baud rate, e.g. 9600, 19200, ...
  
  The baud rate range is 200 ... 30.000 Bd. Please note that the functionality of the XL API is guaranteed for 200 ... 20.000 Bd according to the LIN specification. Higher values should be used with care. With LINpiggy 7269mag and the onboard LIN of VN1611 and VN1531, up to 300.000 Bd ("Flash Mode") can be configured, depending on the bus physic.

- **LINVersion**
  
  - `XL_LIN_VERSION_1_3`
    - Use LIN 1.3 protocol
  
  - `XL_LIN_VERSION_2_0`
    - Use LIN 2.0 protocol

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

#### Example

**Setting up channel as a SLAVE to 9k6 and LIN 1.3**

```c
XLlinStatPar xlStatPar;
xlStatPar.LINMode = XL_LIN_SLAVE;
xlStatPar.baudrate = 9600;

// use LIN 1.3
xlStatPar.LINVersion = XL_LIN_VERSION_1_3;
xlStatus = xlLinSetChannelParams(m_XLportHandle,
                                 m_xlChannelMask[SLAVE],
                                 xlStatPar);
```
7.6 Events

7.6.1 XL LIN Message API

Syntax

```c
union s_xl_lin_msg_api {
    struct s_xl_lin_msg    linMsg;
    struct s_xl_lin_no_ans  linNoAns;
    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 186).
- **linNoAns**
  Structure for the LIN message that gets no answer (see section LIN No Answer on page 187).
- **linWakeUp**
  Structure for the wake up events (see section LIN Wake Up on page 187).
- **linSleep**
  Structure for the sleep events (see section LIN Sleep on page 188).
- **linCRCinfo**
  Structure for the CRC info events (see section LIN CRC Info on page 188).

7.6.2 XL LIN Message

Syntax

```c
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 77).

Parameters

- **id**
  Received LIN message ID.
- **dlc**
  The DLC of the received LIN message.
- **flags**
  **XL_LIN_MSGFLAG_TX**
  The LIN message was sent by the same LIN channel.
  **XL_LIN_MSGFLAG_CRCERROR**
  LIN CRC error.
- **data**
  Content of the message.
- **crc**
  Checksum.
7.6.3 XL LIN Error Message

Tag                  XL_LIN_ERRMSG (see section XLevent on page 77).

7.6.4 XL LIN Sync Error

Description         Notifies an error in analyzing the sync field.
Tag                  XL_LIN_SYNC_ERR (see section XLevent on page 77).

7.6.5 LIN No Answer

Syntax               struct s_lin_NoAns {
                      unsigned char id;
                    }

Description         If a LIN master request gets no slave response a linNoAns event is received.
Tag                  XL_LIN_NOANS (see section XLevent on page 77).

Parameters
  ► id
  The LIN ID on which was the master request.

7.6.6 LIN Wake Up

Syntax               struct s_xl_lin_wake_up {
                      unsigned char flag;
                      unsigned char unused[3];
                      unsigned int startOffs;
                      unsigned int width;
                    }

Description         This event indicates that the channel received a wake up signal.
Tag                  XL_LIN_WAKEUP (see section XLevent on page 77).

Parameters
  ► flag
  If the wake-up signal comes from the internal hardware, the flag is set to XL_LIN_WAKEUP_INTERNAL otherwise it is not set (external wake-up).

  ► unused
  Reserved for future use.

  ► startOffs
  Timestamp correction offset.

  ► width
  Timestamp correction width.

Note
The real time stamp can be calculated as follows:

time stamp = (pxlEvent→timeStamp - wakeUp.StartOffs) + wakeUp.Width
7.6.7 LIN Sleep

**Syntax**

struct s_lin_Sleep {
    unsigned char flag;
};

**Description**

This event indicates changes in the sleep state of the channel.

**Tag**

XL_LIN_SLEEP (see section XLevent on page 77).

**Parameters**

- **flag**
  - XL_LIN_SET_SLEEPMODE
    Channel entered sleep mode.
  - XL_LIN_COMESFROM_SLEEPMODE
    Channel exited sleep mode.
  - XL_LIN_STAYALIVE
    Channel should enter sleep mode but cannot (for example because of bus activity).

7.6.8 LIN CRC Info

**Syntax**

struct s_xl_lin_crc_info {
    unsigned char id;
    unsigned char flags;
};

**Description**

This event is only used if the LIN protocol is >= 2.0.

If a LIN >= 2.0 node is initialized and the function xlLinSetChecksum() is not called (and no checksum model is defined) the hardware detects the according checksum model by itself. The event occurs only one time for the according LIN ID.

**Tag**

XL_LIN_CRCINFO (see section XLevent on page 77).

**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.
7.7 Application Examples

7.7.1 xlLINExample

7.7.1.1 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.

![Example Image]

7.7.1.2 Classes

Description
The example has the following class structure:

- **CaboutDlg**
  About box. → AboutDlg.cpp

- **CLINExampleApp**
  Main MFC class → xlLINExample.cpp

- **CLINExampleDlg**
  The 'main' dialog box → xlLINExampleDlg.cpp

- **CLINFunctions**
  Contains all functions for the LIN access → xlLINFunctions.cpp
7.7.1.3 Functions

**Description**

- **LINGetDevice**
  In order to get the channel mask, use `xlGetChannelMask()` to read all hardware parameters. `xlGetAppConfig()` checks whether the application has already been assigned. If not, a new entry with `xlSetAppConfig()` is created.

- **LINInit**
  `LINInit` opens one port for one channel, or if available two channels (CH1 and CH2). The first channel will be initialized as LIN master including a LIN slave (id=4), the other channel as LIN slave (id=5). After a successfully `xlOpenPort()` call, a Rx thread is created. Use `xlLinSetChannelParams()` in order to initialize the channels (like master/slave and the baud rate). It is also recommended to set up the LIN dlc with `xlLinSetDLC()`.

- **linInitMaster**
  In order to use the LIN bus, it is necessary to define the specific DLC for each LIN ID. → `xlLinSetDLC()`. This must be done only for a LIN master and before you go ‘onBus’.

- **linInitSlave**
  Use `xlLinSetSlave()` to set up slave. Before you go ‘onBus’ it is needed to define the LIN slave ID that cannot be changed after `xlActivateChannel()`. All other parameters like the data values or the DLC can be varied.

- **LINSendMasterReq**
  After the LIN network is specified and the master/slaves are ‘onBus’, the master can transmit master requests with `xlLinSendRequest()`.

- **LINClose**
  When all is done, the port is closed with `xIClosePort()`.
# 8 K-Line Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Introduction</td>
<td>192</td>
</tr>
<tr>
<td>8.2 Flowchart</td>
<td>193</td>
</tr>
<tr>
<td>8.3 Functions</td>
<td>194</td>
</tr>
<tr>
<td>8.4 Structs</td>
<td>199</td>
</tr>
<tr>
<td>8.5 Events</td>
<td>203</td>
</tr>
</tbody>
</table>
8.1 Introduction

Description

The XL Driver Library enables the development of K-Line applications for supported Vector devices (see section System Requirements on page 34). A K-Line application always requires init access (see section xlOpenPort on page 44) multiple K-Line applications cannot use a common physical K-Line channel at the same time.

Depending on the channel property init access (see page 31), the application's main features are as follows:

With init access

► channel parameters can be changed/configured
► K-Line messages can be transmitted on the channel
► K-Line messages can be received on the channel

Without init access

► Not supported. If the application gets no init access on a specific channel, no further function call is possible on the according channel.

Reference

See the flowchart on the next page for all available functions and the according calling sequence.
### 8.2 Flowchart

Calling sequence

---

**Driver Init**
- start
  - xlOpenDriver()
- xGetDriverConfig()
- xGetChannelMask()
- xlOpenPort()

**K-Line Tester**
- xKlineSetParam()
- xKlineSetBaudrate()
- xKlineInitBdTester()
- xActivateChannel()

**Channel Setup**
- xKlineSwitchTesterResistor()
- xKlineSetCommunicationTimingTester()
- xKlineFastInitTester()
- xKlineInit5BdTester()
- xKlineSetCommunicationTimingEcu()
- xKlineInit5BdEcu()

**On Bus**
- xKlineTransmit()
- xKlineSetBaudrate()
- xDeactivateChannel()

**K-Line ECU**
- xlClosePort()
- xlCloseDriver()

---

Figure 27: Function calls for K-Line applications
8.3 Functions

8.3.1 xlKlineFastInitTester

Syntax

```c
XLstatus xlKlineFastInitTester (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    unsigned int length,  
    unsigned char *data,  
    XLklineInitTester *pxlKlineInitTester  
)
```

Description

Execute fast init sequence followed by a data frame (tester). Called after activating the channel.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **length**
  See `xlKlineTransmit` on page 198.

- **data**
  See `xlKlineTransmit` on page 198.

- **pxlKlineInitTester**
  Pointer to the Tester structure (see `XLklineInitTester` on page 201).

Return value

Returns an error code (see section Error Codes on page 482).

8.3.2 xlKlineInit5BdEcu

Syntax

```c
XLstatus xlKlineInit5BdEcu (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLkline5BdEcu *pxlKline5BdEcu  
)
```

Description

Configure the 5Bd-init response frame per ECU. Can be called only once per ECU before activating the channel.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
8 K-Line Commands

- `pxlKline5BdEcu`
  See `XLkline5BdEcu` on page 199.

**Return value**
Returns an error code (see section Error Codes on page 482).

### 8.3.3 `xlKlineInit5BdTester`

**Syntax**
```c
XLstatus xlKlineInit5BdTester (XLportHandle portHandle, XLaccess accessMask, XLkline5BdTester *pxlKline5BdTester)
```

**Description**
Execute 5Baud init pattern (tester). Called after activating the channel.

**Input parameters**
- `portHandle`
The port handle retrieved by `xlOpenPort()`.
- `accessMask`
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- `pxlKline5BdTester`
  See `XLkline5BdTester` on page 200.

**Return value**
Returns an error code (see section Error Codes on page 482).

### 8.3.4 `xlKlineSetBaudrate`

**Syntax**
```c
XLstatus xlKlineSetBaudrate (XLportHandle portHandle, XLaccess accessMask, unsigned int baudrate)
```

**Description**
Sets the baudrate of the tester or the ECU.

**Input parameters**
- `portHandle`
The port handle retrieved by `xlOpenPort()`.
- `accessMask`
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- `baudrate`
  In baud. 200 Bd …115.2 kBd.

**Return value**
Returns an error code (see section Error Codes on page 482).
8.3.5 xlKlineSetCommunicationTimingEcu

Syntax

```c
XLstatus xlKlineSetCommunicationTimingEcu (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLklineSetComEcu *pxlKlineSetComEcu
}
```

Description
Setup the ECU communication parameters. It returns a KLINE_CONFIRMATION event (see K-Line Confirmation Event on page 203).

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pxlKlineSetComEcu**
  Pointer to the Tester structure (see XLklineSetComEcu on page 201).

Return value
Returns an error code (see section Error Codes on page 482).

8.3.6 xlKlineSetCommunicationTimingTester

Syntax

```c
XLstatus xlKlineSetCommunicationTimingTester (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLklineSetComTester *pxlKlineSetComTester
}
```

Description
Setup the tester communication parameters. It returns a KLINE_CONFIRMATION event (see K-Line Confirmation Event on page 203).

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pxlKlineSetComTester**
  Pointer to the structure (see XLklineSetComTester on page 201).

Return value
Returns an error code (see section Error Codes on page 482).

8.3.7 xlKlineSetUartParams

Syntax

```c
XLstatus xlKlineSetUartParams (  
    XLportHandle portHandle,  
    ```$
8.3.8 xlKlineSwitchHighspeedMode

**Syntax**

```c
XLstatus xlKlineSwitchHighspeedMode(
    XLportHandle portHandle,
    XLaccess accessMask,
    unsigned int trxMode
)
```

**Description**

Switches the high-speed mode for LINPiggy7269 on or off. Required for baudrates >30kBd. It is an asynchronous command and returns a K-Line confirmation event.

**Input parameters**

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 29.

- **trxMode**
  Select the speed mode:
  - `XL_KLINE_TRXMODE_NORMAL`
  - `XL_KLINE_TRXMODE_HIGHSPEED`

**Return value**

Returns an error code (see section **Error Codes** on page 482).

---

8.3.9 xlKlineSwitchTesterResistor

**Syntax**

```c
XLstatus xlKlineSwitchTesterResistor(
    XLportHandle portHandle,
    XLaccess accessMask,
    unsigned int testerR
)
```

**Description**

Sets up the UART parameters.

**Input parameters**

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 29.

- **pxlKlineUartParams**
  See **XLklineUartParameter** on page 202

**Return value**

Returns an error code (see section **Error Codes** on page 482).
8.3.10 xlKlineTransmit

Syntax

```c
XLstatus xlKlineTransmit (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    unsigned int length,  
    unsigned char *data  
)
```

Description

Transmits a complete K-Line frame. Can also be used to send a single byte.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **length**
  Data buffer length in bytes. Set to ‘1’ to send a single byte of a K-Line frame. The maximum length is limited to 300 bytes.

- **data**
  K-Line data buffer. Contains a complete K-Line TX frame, e.g.:
  ```c
  unsigned char fmtByte;  
  unsigned char srcAddr;  
  unsigned char trgAddr;  
  unsigned char length;  
  unsigned char data[frame_length]  
  unsigned char checksum;
  ```

Return value

Returns an error code (see section Error Codes on page 482).
8.4 Structs

8.4.1 XLkline5BdEcu

Syntax

```c
struct s_xl_kline_init_5BdEcu {
    unsigned int configure;
    unsigned int addr;
    unsigned int rate5bd;
    unsigned int syncPattern;
    unsigned int W1;
    unsigned int W2;
    unsigned int W3;
    unsigned int W4;
    unsigned int W4min;
    unsigned int W4max;
    unsigned int kb1;
    unsigned int kb2;
    unsigned int addrNot;
    unsigned int reserved;
} XLkline5BdEcu;
```

Parameters

- **configure**
  Flag to configure / unconfigure the ECU:
  - XL_KLINE_UNCONFIGURE_ECU
  - XL_KLINE_CONFIGURE_ECU

- **addr**
  ECU address.

- **rate5bd**
  Baudrate of initial communication: 4.6 ... 5.4 Bd in 1/10 Baud (46 to 54).

- **W1**
  In 1 μs (resolution 10 μs).

- **W2**
  In 1 μs (resolution 10 μs).

- **W3**
  In 1 μs (resolution 10 μs).

- **W4**
  In 1 μs (resolution 10 μs).

- **W4min**
  In 1 μs (resolution 10 μs).

- **W4max**
  In 1 μs (resolution 10 μs).

- **kb1**
  Key byte 1.

- **kb2**
  Key byte 2.

- **addrNot**
  Inverted ECU address with flag | XL_KLINE_FLAG_ASSUME_ADDR_NOT.
8.4.2 XLkline5BdTester

Syntax

```c
struct s_xl_kline_init_5BdTester {
    unsigned int addr;
    unsigned int rate5bd;
    unsigned int W1min;
    unsigned int W1max;
    unsigned int W2min;
    unsigned int W2max;
    unsigned int W3min
    unsigned int W3max;
    unsigned int W4;
    unsigned int W4min;
    unsigned int W4max;
    unsigned int kb2Not;
    unsigned int reserved;
} XLkline5BdTester;
```

Parameters

- **addr**
  ECU address.

- **rate5bd**
  Baudrate of initial communication: 4.6 Bd ... 5.4 Bd in 1/10 Baud (46 to 54).

- **W1min**
  In 1 µs (resolution 10 µs).

- **W1max**
  In 1 µs (resolution 10 µs).

- **W2min**
  In 1 µs (resolution 10 µs).

- **W2max**
  In 1 µs (resolution 10 µs).

- **W3min**
  In 1 µs (resolution 10 µs).

- **W3max**
  In 1 µs (resolution 10 µs).

- **W4**
  In 1 µs (resolution 10 µs).

- **W4min**
  In 1 µs (resolution 10 µs).

- **W4max**
  In 1 µs (resolution 10 µs).

- **kb2Not**
  Inverted key byte 2 with flag | XL_KLINE_FLAG.Take_kb2Not.

- **reserved**
  For future use.
8.4.3 XLklineInitTester

**Syntax**

```c
struct s_xl_kline_init_tester {
    unsigned int TiniL;
    unsigned int Twup;
    unsigned int reserved;
} XLklineInitTester;
```

**Parameters**

- **TiniL**  
  Low phase of the wake-up pattern in 1 µs (resolution 10 µs).

- **Twup**  
  Length of the wake-up pattern in 1 µs (resolution 10 µs)

- **reserved**  
  For future use.

8.4.4 XLklineSetComEcu

**Syntax**

```c
struct s_xl_kline_set_com_ecu {
    unsigned int P1;
    unsigned int P4min;
    unsigned int TinilMin;
    unsigned int TinilMax;
    unsigned int TwupMin;
    unsigned int TwupMax;
    unsigned int reserved;
} XLklineSetComEcu;
```

**Parameters**

- **P1**  
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.

- **P4min**  
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.

- **TinilMin**  
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.

- **TinilMax**  
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.

- **TwupMin**  
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.

- **TwupMax**  
  In 1 µs (resolution 10 µs). Set to ‘0’ for no supervision.

- **reserved**  
  For future use.

8.4.5 XLklineSetComTester

**Syntax**

```c
struct s_xl_kline_set_com_tester {
    unsigned int P1min;
    unsigned int P4;
    unsigned int reserved;
} XLklineSetComTester;
```
Parameters

- **P1min**
  In 1 µs (resolution 10 µs). Set to '0' for no supervision.

- **P4**
  In 1 µs (resolution 10 µs).

- **reserved**
  For future use.

### 8.4.6 XLklineUartParameter

#### Syntax

```c
struct s_xl_kline_uart_params {
    unsigned int databits;
    unsigned int stopbits;
    unsigned int parity;
} XLklineUartParameter;
```

#### Parameters

- **databits**
  Number of data bits (7, 8 or 9).

- **stopbits**
  Number of stop bits (1 or 2).

- **parity**
  - XL_XLINE_UART_PARITY_NONE
  - XL_KLINE_UART_PARITY_EVEN
  - XL_KLINE_UART_PARITY_ODD
8.5 Events

8.5.1 K-Line Data

Syntax

```c
struct s_xl_kline_data {
    unsigned int klineEvtTag;
    unsigned int reserved;

    union {
        XL_KLINE_RX_DATA klineRx;
        XL_KLINE_TX_DATA klineTx;
        XL_KLINE_TESTER_5BD klineTester5Bd;
        XL_KLINE_ECU_5BD klineEcu5Bd;
        XL_KLINE_TESTER_FI_WU_PATTERN klineTesterFiWu;
        XL_KLINE_ECU_FI_WU_PATTERN klineEcuFiWu;
        XL_KLINE_CONFIRMATION klineConfirmation;
        XL_KLINE_ERROR klineError;
    } data;
}; XL_KLINE_DATA;
```

Description
Frame for all K-Line events.

Tag
XL_KLINE_MSG

Parameters

- **klineEvtTag**
  - XL_KLINE_EVT_RX_DATA
    Tag indicating a received data frame.
  - XL_KLINE_EVT_TX_DATA
    Tag indicating a transmitted data frame.
  - XL_KLINE_EVT_TESTER_5BD
    5Bd init related event (tester).
  - XL_KLINE_EVT_ECU_5BD
    5Bd init related event (ECU).
  - XL_KLINE_EVT_TESTER_FI_WU_PATTERN
    Tag indicating a fast init wake-up pattern (tester).
  - XL_KLINE_EVT_ECU_FI_WU_PATTERN
    Tag indicating a fast init wake-up pattern (ECU).
  - XL_KLINE_EVT_ERROR
    Tag indicating a K-Line communication error.
  - XL_KLINE_EVT_CONFIRMATION
    Confirmation event for asynchronous commands.

8.5.2 K-Line Confirmation Event

Syntax

```c
typedef struct s_xl_kline_confirmation {
    unsigned int channel;
    unsigned int confTag;
    unsigned int result;
} XL_KLINE_CONFIRMATION;
```

Description
Confirmation event for asynchronous commands.

Tag
XL_KLINE_EVT_CONFIRMATION
Parameters

► channel
K-Line channel.

► confTag
The confirmation tag:
- XL_KLINE_EVT_TAG_SET_COMM_PARAM_TESTER
  Indicates (asynchronously) the result of setting the communication timing parameters (tester).
- XL_KLINE_EVT_TAG_COMM_PARAM_ECU
  Indicates (asynchronously) the result of setting the communication timing parameters (ECU).
- XL_KLINE_EVT_TAG_SWITCH_HIGHSPEED
  Indicates (asynchronously) the result of setting the high speed mode.

► result
Success or error (0,1).

8.5.3 K-Line Error Event

Description
Event indicating a K-Line communication error.

Tag
XL_KLINE_EVT_ERROR

Parameters
► klineErrorTag
- XL_KLINE_ERROR_TYPE_RXTX_ERROR
  Indicates byte level errors.
- XL_KLINE_ERROR_TYPE_5BD_TESTER
  Indicates timing errors during 5Bd init detected by the tester.
- XL_KLINE_ERROR_TYPE_5BD_ECU
  Indicates timing errors during 5Bd init detected by the ECU.
- XL_KLINE_ERROR_TYPE_IBS
  Indicates an inter-byte space timing error during regular communication.
- XL_KLINE_ERROR_TYPE_FI
  Indicates timing errors during fast init.

8.5.4 K-Line ECU 5Bd Error

Syntax
typedef struct s_xl_kline_error_5bd_ecu {
  unsigned int ecu5BdErr;
} XL_KLINE_ERROR_ECU_5BD;
8.5.5 K-Line Tester 5Bd Error

Syntax

typedef struct s_xl_kline_error_5bd_tester {
  unsigned int tester5BdErr;
} XL_KLINE_ERROR_TESTER_5BD;

Description
Sub structure of XL_KLINE_ERROR with details on timing errors during 5Bd init detected by the tester.

Tag
XL_KLINE_ERROR_TYPE_5BD_TESTER

Parameters

- tester5BdErr
  - XL_KLINE_ERR_TESTER_W1MIN
    Timing violation of W1min during 5Bd init.
  - XL_KLINE_ERR_TESTER_W1MAX
    Timing violation of W1max during 5Bd init.
  - XL_KLINE_ERR_TESTER_W2MIN
    Timing violation of W2min during 5Bd init.
  - XL_KLINE_ERR_TESTER_W2MAX
    Timing violation of W2max during 5Bd init.
  - XL_KLINE_ERR_TESTER_W3MIN
    Timing violation of W3min during 5Bd init.
  - XL_KLINE_ERR_TESTER_W3MAX
    Timing violation of W3max during 5Bd init.
  - XL_KLINE_ERR_TESTER_W4MIN
    Timing violation of W4min during 5Bd init.
  - XL_KLINE_ERR_TESTER_W4MAX
    Timing violation of W4max during 5Bd init.

8.5.6 K-Line ibrErr Error

Syntax

typedef struct s_xl_kline_error_ibs {
  unsigned int ibrErr;
  unsigned int rxtxErrData;
} XL_KLINE_ERROR_IBS;

Description
Sub structure of XL_KLINE_ERROR with details on inter-byte space timing errors during regular communication.
8.5.7 K-Line RXTX Error

Syntax

typedef struct s_xl_kline_error_rxtx {
    unsigned int rxtxErrData;
} XL_KLINE_ERROR_RXTX;

Description
Sub structure of XL_KLINE_ERROR with details on byte level errors.

Parameters
- **rxtxErrData**
  - XL_KLINE_ERR_RXTX_UA
    Unexpected activity.
  - XL_KLINE_ERR_RXTX_MA
    Missing activity.
  - XL_KLINE_ERR_RXTX_ISB
    Invalid sync byte during 5Bd init.

8.5.8 K-Line RX Data

Syntax

struct s_xl_kline_rx_data {
    unsigned int timeDiff;
    unsigned int data;
    unsigned int error;
} XL_KLINE_RX_DATA;

Description
Event for a data frame.

Tag
XL_KLINE_EVT_RX_DATA

Parameters
- **timeDiff**
  Difference between EOF and SOF timestamps in nano seconds.
- **data**
  Data field containing up to 9 bits.
- **error**
  Error mask:
  - XL_KLINE_BYTE_PARITY_ERROR_MASK
    Parity check of the data field failed, e.g. transmission error, parity settings mismatch.
8.5.9 K-Line Tester 5Bd

Syntax

```c
struct s_xl_kline_tester_5bd {
  unsigned int tag5bd;
  unsigned int timeDiff;
  unsigned int data;
} XL_KLINE_TESTER_5BD;
```

Description

5Bd init related event (tester).

Tag

XL_KLINE_EVT_TESTER_5BD

Parameters

▶ tag5bd
  - XL_KLINE_EVT_TAG_5BD_ADDR
    Event tag for ECU address during 5Bd init.
  - XL_KLINE_EVT_TAG_5BD_BAUDRATE
    Event tag indicating regular communication baudrate after 5Bd init.
  - XL_KLINE_EVT_TAG_5BD_KB1
    Event tag indicating key byte 1 during 5Bd init.
  - XL_KLINE_EVT_TAG_5BD_KB2
    Event tag indicating key byte 2 during 5Bd init.
  - XL_KLINE_EVT_TAG_5BD_KB2NOT
    Event tag indicating inverted key byte 2 during 5Bd init.
  - XL_KLINE_EVT_TAG_5BD_ADDRNOT
    Event tag indicating inverted ECU address during 5Bd init.

▶ timeDiff
  Difference between EOF and SOF timestamps in nano seconds.

▶ data
  Data byte or address depending on tag5bd.

8.5.10 K-Line ECU Fastinit WU Pattern

Syntax

```c
struct s_xl_kline_ecu_fastinit_wu_pattern {
  unsigned int timeDiff;
  unsigned int fastInitEdgeTimeDiff;
} XL_KLINE_ECU_FI_WU_PATTERN;
```

Description

Event indicating a fast init wake-up pattern (ECU).

Tag

XL_KLINE_EVT_ECU_FI_WU_PATTERN

Parameters

▶ timeDiff
  Difference between EOF and SOF timestamps in nano seconds.

▶ fastInitEdgeTimeDiff
  In nano seconds (Twup - TiniL).
8.5.11 K-Line Tester Fastinit WU Pattern

**Syntax**

```c
struct s_xl_kline_tester_fastinit_wu_pattern {
    unsigned int timeDiff;
    unsigned int fastInitEdgeTimeDiff;
} XL_KLINE_TESTER_FI_WU_PATTERN;
```

**Description**

Event indicating a fast init wake-up pattern (tester).

**Tag**

XL_KLINE_EVT_TESTER_FI_WU_PATTERN

**Parameters**

- **timeDiff**
  
  Difference between EOF and SOF timestamps in nano seconds.

- **fastInitEdgeTimeDiff**
  
  In nano seconds (Twup - TiniL).

8.5.12 K-Line TX Data

**Syntax**

```c
struct s_xl_kline_tx_data {
    unsigned int timeDiff;
    unsigned int data;
    unsigned int error;
} XL_KLINE_TX_DATA;
```

**Description**

Event for a transmitted data frame.

**Tag**

XL_KLINE_EVT_TX_DATA

**Parameters**

- **timeDiff**
  
  Difference between EOF and SOF timestamps in nano seconds.

- **data**
  
  Data field containing up to 9 bits.

- **error**
  
  Error mask:
  
  - XL_KLINE_BYTE_PARITY_ERROR_MASK
    
    Parity check of the data field failed,
    
    e.g. transmission error, parity settings mismatch.
  
  - XL_KLINE_BYTE_FRAMING_ERROR_MASK
    
    Framing error detected,
    
    e.g. wrong number of data or stop bits.
9 D/A IO Commands (IOcab)

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Introduction</td>
<td>210</td>
</tr>
<tr>
<td>9.2 Flowchart</td>
<td>211</td>
</tr>
<tr>
<td>9.3 Functions</td>
<td>212</td>
</tr>
<tr>
<td>9.4 Events</td>
<td>219</td>
</tr>
<tr>
<td>9.5 Application Examples</td>
<td>221</td>
</tr>
</tbody>
</table>
9.1 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 31), 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

Driver Init

- xlOpenDriver()
- xlGetDriverConfig()
- xlOpenPort()

IO Setup

- xlGetChannelMask()
- xlOpenPort()
- xlGetDriverConfig()
- xlGetApplConfig()
- xlSetApplConfig()
- xlGetChannelIndex()
- xlResetClock()
- xlSetTimerRate()
- * xlDAIOSetAnalogOutput()
- * xlDAIOSetAnalogParameters()
- * xlDAIOSetAnalogTrigger()
- * xlDAIOSetDigitalOutput()
- * xlDAIOSetDigitalParameters()
- * xlDAIOSetPWMOutput()
- * xlDAIOSetMeasurementFrequency()

Measurement

- * xlDAIOSetDigitalOutput()
- * xlDAIOSetDigitalParameters()
- * xlDAIOSetPWMOutput()
- * xlDAIOSetMeasurementFrequency()
- xlReceive()
- xlFlushReceiveQueue()

End

init access required

Function()
Special DAIO API function
Function()
Common API function

* if access required

Figure 28: Function calls for DAIO applications
9.3 Functions

9.3.1 xlDAIOSetAnalogParameters

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).
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)

9.3.2 xlDAIOSetAnalogOutput

Syntax

XLstatus xlDAIOSetAnalogOutput (    XLportHandle portHandle,    XLaccess accessMask,    unsigned int analogLine1,    unsigned int analogLine2,    unsigned int analogLine3,    unsigned int analogLine4)

Description

Sets analog output line to voltage level as requested (specified in millivolts). Optionally, the flag XL_DAIO_IGNORE_CHANNEL can be used not to change line’s current level.

Input parameters

► portHandle
  The port handle retrieved by xlOpenPort().

► accessMask
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► analogLine1
  Voltage level for AIO0.

► analogLine2
  Voltage level for AIO1.

► analogLine3
  Voltage level for AIO2.

► analogLine4
  Voltage level for AIO3.
9.3.3 xlDAIOSetAnalogTrigger

Syntax

```cpp
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

9.3.4 xlDAIOSetDigitalParameters

Syntax

```cpp
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**
## 9.3.5 xlDAIOSetDigitalOutput

### Syntax

```c
XLstatus xlDAIOSetDigitalOutput (XLportHandle portHandle,
        XLaccess accessMask,
        unsigned int outputMask,
        unsigned int valuePattern)
```

### Description
Sets digital output line to desired logical level.

### Input parameters
- **portHandle**
  The port handle retrieved by `xlOpenPort()`.

### Return value
Returns an error code (see section Error Codes on page 482).

---

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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.
9.3.6 xlDAI0SetPWMOutput

Syntax

```
xlstatus xlDAI0SetPWMOutput (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    unsigned int frequency,  
    unsigned int value)  
```

Description
Changes PWM output to defined frequency and value.

Input parameters

► portHandle
The port handle retrieved by xlOpenPort().

► accessMask
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
9 D/A IO Commands (IOcab)

- **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 482).

**Example**

**Setting up the IOcab8444**

Set PWM frequency to 2500 Hz

\[
\text{frequency} = 2500
\]

Set PWM ratio to 25% (75% pulse low, 25% pulse high)

\[
\text{value} = 2500
\]

### 9.3.7 xlDAIOSetMeasurementFrequency

**Syntax**

```c
XLstatus xlDAIOSetMeasurementFrequency ( 
  XLportHandle portHandle, 
  XLaccess accessMask, 
  unsigned int measurementInterval)
```

**Description**
Sets the measurement frequency. `xlEvents` will be automatically triggered, which can be received by `xlReceive`. For manual trigger, see section `xlDAIORequestMeasurement` on page 217.

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **measurementInterval**
  Measurement frequency in ms.

**Return value**
Returns an error code (see section Error Codes on page 482).

### 9.3.8 xlDAIORequestMeasurement

**Syntax**

```c
XLstatus xlDAIORequestMeasurement ( 
  XLportHandle portHandle, 
  XLaccess accessMask)
```

**Description**
Forces manual measurement of DAIO values.
Input parameters

► portHandle
   The port handle retrieved by `xlOpenPort()`.

► accessMask
   The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

Return value

Returns an error code (see section Error Codes on page 482).
9.4 Events

9.4.1 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 77).

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.
9.5 Application Examples

9.5.1 xIDAIOexample

9.5.1.1 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.

![Running xIDAIOexample](image)

Figure 29: Running xIDAIOexample

9.5.1.2 Setup

Pin definition

The following pins of the IOcab 8444opto are used in this example:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO0</td>
<td>14</td>
<td>Analog output</td>
</tr>
<tr>
<td>AIO1</td>
<td>7</td>
<td>Analog input</td>
</tr>
<tr>
<td>AIO2</td>
<td>15</td>
<td>Analog input</td>
</tr>
<tr>
<td>AIO3</td>
<td>8</td>
<td>Analog input</td>
</tr>
<tr>
<td>DIO0</td>
<td>1</td>
<td>Digital output (shared electronic switch with DIO1).</td>
</tr>
<tr>
<td>DIO1</td>
<td>9</td>
<td>Digital output (supplied by DIO0, when switch is closed).</td>
</tr>
<tr>
<td>DIO2</td>
<td>2</td>
<td>Digital input.</td>
</tr>
<tr>
<td>DIO3</td>
<td>10</td>
<td>Digital input.</td>
</tr>
</tbody>
</table>

![Setup Diagram](image)
9.5.1.3 Keyboard commands

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ENTER&gt;</td>
<td>Toggle digital output.</td>
</tr>
<tr>
<td>&lt;x&gt;</td>
<td>Closes application.</td>
</tr>
</tbody>
</table>

9.5.1.4 Output Examples

Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO0:</td>
<td>4032mV</td>
</tr>
<tr>
<td>AIO1:</td>
<td>0mV</td>
</tr>
<tr>
<td>AIO2:</td>
<td>0mV</td>
</tr>
<tr>
<td>AIO3:</td>
<td>0mV</td>
</tr>
<tr>
<td>Switch selected:</td>
<td>DIO0/DIO1</td>
</tr>
<tr>
<td>Switch states:</td>
<td>OPEN</td>
</tr>
<tr>
<td>Digital Port:</td>
<td>DIO7 DIO6 DIO5 DIO4 DIO3 DIO2 DIO1 DIO0 val</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 1 (1)</td>
</tr>
</tbody>
</table>

Explanation

► “AIO0” displays 4032mV, since it is set to output with maximum output level.
► “AIO1” displays 0mV, since there is no applied voltage at this input.
► “AIO2” displays 0mV, since there is no applied voltage at this input.
► “AIO3” displays 0mV, since there is no applied voltage at this input.
► “Switch selected” displays DIO0/DIO1 (first switch)
► “Switch states” displays the state of switch between DIO0/DIO1
► “Digital Port” shows the single states of DIO7…DIO0:
  - DIO0: displays ‘1’ (always ‘1’, due the voltage supply)
  - DIO1: displays ‘0’ (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays ‘0’ (output of DIO1)
  - DIO3: displays ‘0’ (output of DIO1)
  - DIO4: displays ‘0’ (n.c.)
  - DIO5: displays ‘0’ (n.c.)
  - DIO6: displays ‘0’ (n.c.)
  - DIO7: displays ‘0’ (n.c.)
Example

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

9.5.1.5 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.

9.5.2 xIDAIOdemo

9.5.2.1 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.
9.5.2.2 Classes

**Description**

The example has the following class structure:

- **CXIDAIIdemoApp**
  Main MFC class → xlDAIIdemo.cpp

- **CXIDAIIdemoDlg**
  Handles the window dialog messages and control the IOcab → xlDAIIdemoDlg.cpp

- **ReceiveThread**
  Thread to handle the DAIO events.
10 D/A IO Commands (IOpiggy)

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Introduction</td>
<td>226</td>
</tr>
<tr>
<td>10.2 Flowchart</td>
<td>227</td>
</tr>
<tr>
<td>10.3 Functions</td>
<td>228</td>
</tr>
<tr>
<td>10.4 Structs</td>
<td>232</td>
</tr>
<tr>
<td>10.5 Events</td>
<td>237</td>
</tr>
</tbody>
</table>
10.1 Introduction

**Description**

The XL Driver Library enables the development of DAIO applications for the Vector IOpiggy 8642, the onboard DAIO of VN5640/VN7640 and the DoIP Activation Line of VN5610A and VN5620.

Depending on the channel property init access (see page 31), the application’s main features are as follows:

**With init access**

► channel parameters can be changed/configured
► DAIO lines can be set
► DAIO lines can be read

**Without init access**

► DAIO lines can be read

**Reference**

See the flowchart on the next page for all available functions and the according calling sequence.
10 D/A IO Commands (IOpiggy)

10.2 Flowchart

Calling sequence

Driver Init

- xlOpenDriver()
- xlGetDriverConfig()
- xlOpenPort()
- xlGetChannelIndex()
- xlGetChannelMask()
- xlGetAppConfig()
- xlGetDriverConfig()
- xlGetAppConfig()

IO Setup

- xlActivateChannel()
- xlSetNotification()
- yes
- no
- if no: calls with * marked are not available (only receiving messages)
- * xlIoSetTriggerMode()
- * xlIoConfigurePort()
- * xlIoSetDigInThreshold()
- * xlIoSetDigOutLevel()
- * xlIoSetDigitalOutput()
- * xlIoSetAnalogOutput()
- xlGetDriverConfig()
- xlGetApplConfig()
- xlSetApplConfig()
- xlGetChannelIndex()
- xlResetClock()
- xlResetClock()
- xlIoSetTriggerMode()
- xlIoConfigurePort()
- xlIoSetDigInThreshold()
- xlIoSetDigOutLevel()
- xlIoSetDigitalOutput()
- xlIoSetAnalogOutput()
- xlSetTimerRate()
- xlFlushReceiveQueue()
- xlReceive()
- xlSetNotification()
- * xlActivateChannel()
- * xlEnableTrigger()
- * xlDisableTrigger()
- * xlStopTrigger()
- * xlStartTrigger()

Measurement

- xlDeactivateChannel()
- xlClosePort()
- xlCloseDriver()
- end

Figure 31: Function calls for DAIO (IOpiggy) applications
10.3 Functions

10.3.1 xlIoSetTriggerMode (IOpiggy)

Syntax

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

Description

Sets the DAIO trigger mode for the analog and digital ports.

**Note**

This command can be called only once per port type (analog and digital) and only when the channel is deactivated (see flowchart in section Introduction on page 240).

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pxlDaioTriggerMode**
  Use this structure to define the trigger type (see section XLdaioTriggerMode on page 232).

Return value

Returns an error code (see section Error Codes on page 482).

10.3.2 xlIoConfigurePorts

Syntax

```c
XLstatus xlIoConfigurePorts (  
    XLportHandle  portHandle,  
    XLaccess      accessMask,  
    XLdaioSetPort *pxlDaioSetPort)
```

Description

Configures the DAIO ports.

**Note**

This command can be called only once.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
10.3.3 xlIoSetDigInThreshold

Syntax

```c
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 43). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 29.

- **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 482).

10.3.4 xlIoSetDigOutLevel

Syntax

```c
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 43). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 29.

- **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 482).
10.3.5 xlIoSetDigitalOutput

Syntax

XLstatus xlIoSetDigitalOutput (  
    XLportHandle   portHandle,  
    XLaccess       accessMask,  
    XLdaioDigitalParams* pXlDaioDigitalParams)

Description

Configures the digital output.

Input parameters

► portHandle
The port handle retrieved by xlOpenPort().

► accessMask
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► pXlDaioDigitalParams
Use this structure to set the value of the digital out pin (see section XLdaioDigitalParams (IOpiggy) on page 235).

Return value

Returns an error code (see section Error Codes on page 482).

10.3.6 xlIoSetAnalogOutput

Syntax

XLstatus xlIoSetAnalogOutput (  
    XLportHandle   portHandle,  
    XLaccess       accessMask,  
    XLdaioAnalogParams* pXlDaioAnalogParams)

Description

Configures the analog output.

Input parameters

► portHandle
The port handle retrieved by xlOpenPort().

► accessMask
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► pXlDaioAnalogParams
Use this structure to set the value of the analog out pin (see section XLdaioAnalogParams on page 235).

Return value

Returns an error code (see section Error Codes on page 482).

10.3.7 xlIoStartSampling

Syntax

XLstatus xlIoStartSampling (  
    XLportHandle   portHandle,  
    XLaccess       accessMask,  
    unsigned int   portTypeMask)
Description

This command requests DAIO measurement data and is independent of the defined trigger mode.

Input parameters

► portHandle
The port handle retrieved by `xlOpenPort()`.

► accessMask
The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► portTypeMask
`XL_DAIO_PORT_TYPE_MASK_ANALOG`
`XL_DAIO_PORT_TYPE_MASK_DIGITAL`

Return value

Returns an error code (see section Error Codes on page 482).
10.4 Structs

10.4.1 XLdaioTriggerMode

Syntax

```c
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.
  Specifies the trigger type:
  - XL_DAIO_TRIGGER_TYPE_RISING
  - XL_DAIO_TRIGGER_TYPE_FALLING
  - XL_DAIO_TRIGGER_TYPE_BOTH
10.4.2 XLdaioSetPort

**Syntax**

```c
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_PUSHFULL
XL_DAIO_PORT_DIGITAL_IN
XL_DAIO_PORT_DIGITAL_SWITCH

For analog ports:
XL_DAIO_PORT_ANALOG_IN
XL_DAIO_PORT_ANALOG_OUT
XL_DAIO_PORT_ANALOG_DIFF
XL_DAIO_PORT_ANALOG_OFF

XL_DAIO_PORT_ANALOG_IN and XL_DAIO_PORT_ANALOG_OUT can be defined at the same time.

reserved
Set to 0.

Note
The DoIP Activation Lines of VN5640 and VN7640 are assigned to XL_DAIO_PORT_MASK_DIGITAL_D3 and XL_DAIO_PORT_MASK_DIGITAL_D4.
The DoIP Activation Line of VN5610A and VN5620 is assigned to XL_DAIO_PORT_MASK_DIGITAL_D3.
Example

```c
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);
```

10.4.3 XLdaioDigitalParams (IOpiggy)

**Syntax**

```c
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

10.4.4 XLdaioAnalogParams

**Syntax**

```c
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).
10.5 Events

10.5.1 XL DAIO Piggy Data

Syntax

```c
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 242.
- IOpiggy
  xlIoSetTriggerMode (IOpiggy) on page 228.

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 246 and section XL IO Analog Data on page 245.

Example

```c
Checking digital port D0
if (ev.daioData.digital.digitalInputData &
    XL_DAIO_PORT_MASK_DIGITAL_D0) {...}
```

10.5.2 XL IO Analog Data

Syntax

```c
typedef struct s_xl_io_analog_data {
    unsigned int measuredAnalogData0;
    unsigned int measuredAnalogData1;
    unsigned int measuredAnalogData2;
};
```
Parameters

- **measuredAnalogData0**
  First analog port that is defined as an input.
  This value is 0 for differential input.

- **measuredAnalogData1**
  Second analog port that is defined as an input.
  This value is 0 for differential input.

- **measuredAnalogData2**
  Third analog port that is defined as an input.
  This value is 0 for differential input.

- **measuredAnalogData3**
  Fourth analog port that is defined as an input.
  This value is 0 for differential input.

**Note**

The `measuredAnalogData` returned by the IOpiggy is in millivolt, while VN1630 (A) and VN1640(A) return the value in samples of their ADC.

This ADC has an input range of 18 V and a resolution of 10 bit, therefore the application must multiply the `measuredAnalogData` returned by VN1630(A) or VN1640(A) by 17.58 mV.

For more information on the ADC including the use of series resistors, refer to the VN1600 Interface Family manual.

10.5.3 XL IO Digital Data

**Syntax**

```c
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.
In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Introduction</td>
<td>240</td>
</tr>
<tr>
<td>11.2 Flowchart</td>
<td>241</td>
</tr>
<tr>
<td>11.3 Functions</td>
<td>242</td>
</tr>
<tr>
<td>11.4 Structs</td>
<td>244</td>
</tr>
<tr>
<td>11.5 Events</td>
<td>245</td>
</tr>
</tbody>
</table>
11.1 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 31), 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.
11.2 Flowchart

Calling sequence

Driver Init
- `xlOpenDriver()`
- `xlGetDriverConfig()`
- `xlOpenPort()`
- `xlGetChannelMask()`

IO Setup
- `xlGetApplConfig()`
- `xlSetApplConfig()`
- `xlGetChannelIndex()`
- `xlIoSetTriggerMode()` (init access required)
- `xlIoSetDigitalOutput()` (only receiving messages)

Measurement
- `xlResetClock()`
- `xlSetTimerRate()`
- `xlIoStartSampling()` (init access required)
- `xlReceive()`
- `xlFlushReceiveQueue()`

End
- `xlDeactivateChannel()`
- `xlClosePort()`
- `xlCloseDriver()`

Figure 32: Function calls for DAIO (VN1600) applications
11.3 Functions

11.3.1 xlIoSetTriggerMode (VN1600)

Syntax

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

Description

Sets the DAIO trigger mode for the analog and digital ports. A port group must not have more than one trigger source.

Note

This command can be called only once before xlActivateChannel().

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pxlDaioTriggerMode**
  Use this structure to define the trigger type (see section XLdaioTriggerMode on page 232).
  Note: Currently only XL_DAIO_TRIGGER_TYPE_CYCLIC is supported.

Return value

Returns an error code (see section Error Codes on page 482).

11.3.2 xlIoSetDigitalOutput

Syntax

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

Description

Configures the digital output.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pxlDaioDigitalParams**
  Use this structure to set the value of the digital out pin (see section XLdaioDigitalParams (VN1600) on page 244).
| Return value | Returns an error code (see section Error Codes on page 482). |
11.4 Structs

11.4.1 XLdaioDigitalParams (VN1600)

**Syntax**
```c
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
11.5 Events

11.5.1 XL DAIO Piggy Data

Syntax

```c
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 242.

► IOpiggy

xlIoSetTriggerMode (IOpiggy) on page 228.

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:

```c
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 246 and section XL IO Analog Data on page 245.

Example

```
Checking digital port D0
if (ev.daioData.digital.digitalInputData & XL_DAIO_PORT_MASK_DIGITAL_D0) {...}
```

11.5.2 XL IO Analog Data

Syntax

```c
typedef struct s_xl_io_analog_data {
    unsigned int measuredAnalogData0;
    unsigned int measuredAnalogData1;
    unsigned int measuredAnalogData2;
};
```
Parameters

- **measuredAnalogData0**
  First analog port that is defined as an input.
  This value is 0 for differential input.

- **measuredAnalogData1**
  Second analog port that is defined as an input.
  This value is 0 for differential input.

- **measuredAnalogData2**
  Third analog port that is defined as an input.
  This value is 0 for differential input.

- **measuredAnalogData3**
  Fourth analog port that is defined as an input.
  This value is 0 for differential input.

**Note**
The measuredAnalogData returned by the IOpiggy is in millivolt, while VN1630 (A) and VN1640(A) return the value in samples of their ADC.

This ADC has an input range of 18 V and a resolution of 10 bit, therefore the application must multiply the measuredAnalogData returned by VN1630(A) or VN1640(A) by 17.58 mV.

For more information on the ADC including the use of series resistors, refer to the VN1600 Interface Family manual.

### 11.5.3 XL IO Digital Data

**Syntax**
```c
typedef struct s_xl_io_digital_data {
    unsigned int digitalInputData;
} XL_IO_DIGITAL_DATA;
```

**Parameters**
- **digitalInputData**
  Contains the data of port 0 .. 7. It is independent of the port function.
12 MOST Commands

In this chapter you find the following information:

12.1 Introduction ................................................................. 248
12.2 Flowchart ........................................................................... 249
12.3 Specific OS8104 Registers ................................................... 251
12.4 Functions .......................................................................... 252
12.5 Structs ............................................................................... 284
12.6 Events ............................................................................... 287
12.7 Application Examples ........................................................ 307
12.1 Introduction

**Description**

The XL Driver Library enables the development of MOST applications for supported Vector devices (see section System Requirements on page 34). A MOST application always requires *init access* (see section xlOpenPort on page 44) multiple MOST applications cannot use a common physical MOST channel at the same time.

Depending on the channel property *init access* (see page 31), 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:

<table>
<thead>
<tr>
<th>frequency</th>
<th>44.1 kHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node address</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>Group address</td>
<td>0x300</td>
</tr>
<tr>
<td>Alternate packet address</td>
<td>0xFFF</td>
</tr>
</tbody>
</table>
12.2 Flowchart

Calling sequence

Driver Init

start

 xlOpenDriver()

 xlGetDriverConfig()

 xlGetChannelMask()

 xlOpenPort()

 Setup

 no

 Driver access?

 yes

 xlSetNotification()

 xlActivateChannel()

 Measurement (1/2)

 xlResetClock()

 xlMostGenerateLightError()

 xlMostGenerateLockError()

 xlMostCtrlRxBuffer()

 xlMostTwinklePowerLed()

 xlSetTimerRate()

 xlMostSwitchEventSources()

 xlMostSetAllBypass()

 xlMostSetFrequency()

 xlMostSetTimingMode()

 xlMostSyncVolume()

 xlMostWriteRegister()

 xlMostGetAllBypass()

 xlMostGetFrequency()

 xlMostGetTimingMode()

 xlMostReadRegister()

 xlMostSyncGetVolumeStatus()

 xlMostSyncGetMuteStatus()

 xlMostGetRxLight()

 xlMostGetTxLight()

 xlMostGetLockStatus()

 Figure 33: Function calls for MOST applications (1/2)
Calling sequence

Figure 34: Function calls for MOST applications (2/2)
### 12.3 Specific OS8104 Registers

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

- The time units are approximately:
  - 421 µs at Fs = 38 kHz
  - 363 µs at Fs = 44.1 kHz
  - 333 µs at Fs = 48 kHz
12.4 Functions

12.4.1 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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).

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_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 482).

12.4.2 xlMostSetAllBypass

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

Description
Opens/closes the bypass 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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
XL_MOST_ALLBYPASS

Return value
Returns an error code (see section Error Codes on page 482).

12.4.3 xlMostGetAllBypass

Syntax
XLstatus xlMostGetAllBypass(
XLportHandle portHandle,
XLaccess accessMask,
XLuserHandle userHandle)

Description
Gets the bypass 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

Return event
XL_MOST_ALLBYPASS

Return value
Returns an error code (see section Error Codes on page 482).

12.4.4 xlMostSetTimingMode

Syntax
XLstatus xlMostSetTimingMode(
XLportHandle portHandle,
XLaccess accessMask,
XLuserHandle userHandle,
unsigned char timingMode)

Description
Sets the timing mode between master/slave.
## 12.4.5 `xlMostGetTimingMode`

### Syntax

```c
XLstatus xlMostGetTimingMode(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle)
```

### Description

Gets the timing mode (timing master/ timing 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 43). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 29.

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

### Return event

`XL_MOST_TIMINGMODE, XL_MOST_TIMINGMODE_SPDIF`

### Return value

Returns an error code (see section **Error Codes** on page 482).
12.4.6 xlMostSetFrequency

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

12.4.7 xlMostGetFrequency

Syntax

```c
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

- **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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

Return event
XL_MOST_FREQUENCY

Return value
Returns an error code (see section Error Codes on page 482).

12.4.8 xlMostWriteRegister

Syntax
XLstatus xlMostWriteRegister(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned short adr,
    unsigned char numBytes,
    unsigned char data[16])

Description
Writes up to 16 register values of a hardware chip and returns a write confirmation. Refer also to xlMostWriteSpecialRegister().

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- 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 251).

- numBytes
  Number of bytes.

- data[16]
  Register values.

Return event
XL_MOST_REGISTER_BYTES

Return value
Returns an error code (see section Error Codes on page 482).
Example

**Group setup to address 0x0300**

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

### 12.4.9 xlMostReadRegister

**Syntax**

```c
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 `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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 251).

- **numBytes**
  Number of bytes.

**Return event**

`XL_MOST_REGISTER_BYTES`

**Return value**

Returns an error code (see section Error Codes on page 482).

### 12.4.10 xlMostWriteRegisterBit

**Syntax**

```c
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.
12 MOST Commands

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 251).

- **mask**
  Bit mask.

- **Value**
  Register value.

Return event

`XL_MOST_REGISTER_BITS`

Return value

Returns an error code (see section Error Codes on page 482).

### 12.4.11 `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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
12.4.12 xlMostAsyncTransmit

**Syntax**

```c
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 `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 43). For further information on channel/access masks please also refer to section *Principles of the XL Driver Library* on page 29.

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

- **pAsyncMsg**
  See section `XLMOST_CTRL_MSG_EV` on page 295 (structure `s_xl_most_ctrl_msg`).

**Return event**

`XL_MOST_CTRL_TX`

**Return value**

Returns an error code (see section *Error Codes* on page 482).

12.4.13 xlMostSyncGetAllocTable

**Syntax**

```c
XLstatus xlMostSyncGetAllocTable(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle)
```
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 43). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 29.

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

Return event

`XL_MOST_SYNC_ALLOCTABLE`

Return value

Returns an error code (see section **Error Codes** on page 482).

### 12.4.14 `xlMostCtrlSyncAudio`

**Syntax**

```c
XLstatus xlMostCtrlSyncAudio(
  XLportHandle portHandle,
  XLaccess accessMask,
  XLuserData 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 43). For further information on channel/access masks please also refer to section **Principles of the XL Driver Library** on page 29.

- **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 482).

### 12.4.15 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

**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).
12 MOST Commands

- **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 482).

### 12.4.16 xlMostSyncVolume

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

devicex
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 482).

12.4.17 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- 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 482).

12.4.18 xlMostSyncMute

Syntax
XLstatus xlMostSyncMute (  
XLportHandle portHandle,
### 12.4.19 xlMostSyncGetMuteStatus

**Syntax**

```c
XLstatus xlMostSyncGetMuteStatus ( 
    XLportHandle portHandle, 
    XLaccess accessMask, 
    XLUserHandle userHandle, 
    unsigned int device)
```

**Description**

Requests mute state.

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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
  ```

- **mute**
  ```
  XL_MOST_NO_MUTE
  Port not muted.
  
  XL_MOST_MUTE
  Port is muted.
  ```

**Return event**

`XL_MOST_SYNC_MUTE_STATUS`

**Return value**

Returns an error code (see section Error Codes on page 482).
12.4.20 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

12.4.21 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► userHandle
The handle is created by the application and is used for the event assignment.
12 MOST Commands

- **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 482).

### 12.4.22 xlMostGetTxLight

**Syntax**

```c
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 `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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

### 12.4.23 xlMostSetLightPower

**Syntax**

```c
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 `xlOpenPort()`.
12.4.24 xlMostGetLockStatus

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

12.4.25 xlMostGenerateLightError

**Syntax**

```c
XLstatus xlMostGenerateLightError (  
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned long lightofftime,
```

**Description**

- **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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

**Return event**

`XL_MOST_TXLIGHT_POWER`

**Return value**

Returns an error code (see section Error Codes on page 482).
### 12.4.26 xlMostGenerateLockError

**Syntax**

```c
XLstatus xlMostGenerateLockError(
    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 `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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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 482).

12.4.27 xlMostCtrlRxBuffer

Syntax
```
XLstatus xlMostCtrlRxBuffer ( 
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 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

12.4.28 xlMostCtrlConfigureBusload

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
  ► 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 284).

Return event
None.

Return value
Returns an error code (see section Error Codes on page 482).

12.4.29 xlMostCtrlGenerateBusload

Syntax
XLstatus xlMostCtrlGenerateBusload(
  XLportHandle portHandle,
  XLaccess accessMask,
  XLuserHandle userHandle,
  unsigned long numberCtrlFrames)
12.4.30 xlMostAsyncConfigureBusload

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 284).

Return event
None.

Return value
Returns an error code (see section Error Codes on page 482).
12.4.31 xlMostAsyncGenerateBusload

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 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

12.4.32 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

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

- **pEventBuffer**
  Pointer the event buffer.
  Buffer size: XL_MOST_EVENTBUFFER_SIZE.

Return event

If the queue is empty: XL_ERR_QUEUE_IS_EMPTY. If the buffer within the application is too small, the function returns XL_ERR_BUFFER_TOO_SMALL. 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 482).
12.4.33 xlMostTwinklePowerLed

Syntax

```c
XLstatus xlMostTwinklePowerLed (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLuserHandle userHandle)
```

Description

The MOST device 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 43). For further information on channel/access masks please also refer to section Prin-ciples of the XL Driver Library on page 29.

- **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 482).
12.4.34 Streaming

12.4.34.1 General Information

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 xlMostStreamOpen(), 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 xlMostStreamBufferAllocate(). 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 xlMostStreamOpen on page 277). 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 xlMostStreamBufferSetNext().

4. The stream is started with xlMostStreamStart(). The successful start is acknowledged with an XL_MOST_STREAM_STATE event \( (\text{streamState} = \text{XL_MOST_STREAM_STATE_STARTED}) \).

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 xlMostStreamBufferSetNext(). This is repeated cyclically until the stream is stopped with xlMostStreamStop().

6. A stream is stopped by xlMostStreamStop(). 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 xlMostStreamBufferDeAllocateAll().

8. The stream is closed with xlMostStreamClose() 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 \( (\text{streamState} = \text{XL_MOST_STREAM_STATE_STOPPED}) \).

It is possible to clear all buffers of a certain stream with xlMostStreamClearBuffers(). 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 xlMostStreamBufferAllocate() and acknowledged by the event XL_MOST_STREAM_BUFFER.

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

**Note**

If the application reports a filled buffer to the driver by xlMostStreamBufferSetNext() 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 xlMostStreamBufferSetNext() 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.

### 12.4.34.2 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.
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 = 0x00000001 is opened.

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 = 0x00000001 is opened.

In this case it has the following format:

<table>
<thead>
<tr>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bit</td>
<td>Start of frame time stamp from hardware clock; unsynchronized; in 20 ns; LSB first</td>
</tr>
<tr>
<td>2, 4, 6... 60 bytes</td>
<td>MOST frame data; the number of bytes depends on parameter numChannels of MostSyncStrmOpen; for odd values of numChannels a fill byte (0xFB) is inserted</td>
</tr>
<tr>
<td>8 bit</td>
<td>Reserved</td>
</tr>
<tr>
<td>4 bit</td>
<td>SBC (mask: 0b11110000)</td>
</tr>
<tr>
<td>1 bit</td>
<td>Light status (mask: 0b00001000)</td>
</tr>
<tr>
<td>1 bit</td>
<td>Lock status (mask: 0b00000100)</td>
</tr>
<tr>
<td>1 bit</td>
<td>Overflow flag (mask: 0b00000010)</td>
</tr>
<tr>
<td>1 bit</td>
<td>Underflow flag (mask: 0b00000001)</td>
</tr>
</tbody>
</table>

### 12.4.35 xlMostStreamOpen

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

12.4.36 xlMostStreamClose

Syntax
XLstatus xlMostStreamClose(
  XLportHandle portHandle,
  XLaccessMask accessMask,
  XLuserHandle userHandle,
  unsigned int streamHandle)

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
► 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
► 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 = closed).

Return value
Returns an error code (see section Error Codes on page 482).

12.4.37 xlMostStreamStart

Syntax
XLstatus xlMostStreamStart(
  XLportHandle portHandle,
  XLaccessMask accessMask,
  XLuserHandle userHandle,
  unsigned int streamHandle,
  unsigned char syncChannels[MOST_ALLOC_TABLE_SIZE])

Description
Starts the transmission of data from or to the buffer. The application will be informed by an XL_MOST_STREAM_BUFFER event if the buffer is ready. Before starting the stream, some buffers have to be allocated by calling xlMostStreamBufferAllocate().

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
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 482).

12.4.38 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

12.4.39 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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 482).

12.4.40 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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 482).
12.4.41 xlMostStreamBufferSetNext

Syntax

```c
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 release 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

12.4.42 xlMostStreamClearBuffers

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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 482).

12.4.43 xlMostStreamGetInfo

Syntax

```c
XLstatus xlMostStreamGetInfo(
    XLportHandle portHandle,
    XLaccessMask accessMask,
    XLuserRepository 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 `xlMostStreamStart()`.

**Return event**  
None.

**Return value**  
Returns an error code (see section Error Codes on page 482).
12.5 Structs

12.5.1 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 297

12.5.2 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.

### 12.5.3 XL_MOST_STREAM_OPEN

**Syntax**
```c
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.
12.6 Events

12.6.1 s-xl_event_most

Syntax

```c
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_MOSTSpy`

  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.
12.6.2 s_xl_most_tag_data

Syntax

```c
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_SYNC_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_NV mostCtrlSyncAudio;
    XL_MOST_CTRL_SYNC_AUDIO_EX_NV 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**
  section XL_MOST_CTRL_SPY_EV on page 294.

- **mostCtrlMsg**
  See section XL_MOST_CTRL_MSG_EV on page 295.

- **mostAsyncMsg**
  section XL_MOST_ASYNC_MSG_EV on page 297.

- **mostAsyncTx**
  section XL_MOST_ASYNC_TX_EV on page 297.

- **mostSpecialRegister**
  See section XL_MOST_SPECIAL_REGISTER_EV on page 292.

- **mostEventSource**
  See section XL_MOST_EVENT_SOURCE_EV on page 290.

- **mostAllBypass**
  See section XL_MOST_ALL_BYPASS_EV on page 290.

- **mostTimingMode**
  See section XL_MOST_TIMING_MODE_EV on page 290.

- **mostTimingModeSpdif**
  See section XL_MOST_TIMING_MODE_SPDIF_EV on page 291.

- **mostFrequency**
  See section XL_MOST_FREQUENCY_EV on page 291.

- **mostRegisterBytes**
  See section XL_MOST_REGISTER_BYTES on page 291.
12 MOST Commands

► mostRegisterBits
See section XL_MOST_REGISTER_BITS_EV on page 292.

► mostSyncAlloc
See (see section XL_MOST_SYNC_ALLOC_EV on page 298).

► mostCtrlSyncAudio
See section XL_MOST_CTRL_SYNC_AUDIO_EV on page 301.

► mostCtrlSyncAudioEx
See section XL_MOST_CTRL_SYNC_AUDIO_EX on page 301.

► mostSyncVolumeStatus
See section XL_MOST_SYNC_VOLUME_STATUS_EV on page 298.

► mostSyncMutesStatus
See section XL_MOST_SYNC_MUTES_STATUS_EV on page 302.

► mostRxLight
See section XL_MOST_RX_LIGHT_EV on page 299.

► mostTxLight
See section XL_MOST_TX_LIGHT_EV on page 299.

► mostLightPower
See section XL_MOST_LIGHT_POWER_EV on page 302.

► mostLockStatus
See section XL_MOST_LOCK_STATUS_EV on page 299.

► mostGenLightError
See section XL_MOST_GEN_LIGHT_ERROR_EV on page 302.

► mostGenLockError
See section XL_MOST_GEN_LOCK_ERROR_EV on page 303.

► mostRxBuffer
See section XL_MOST_RX_BUFFER_EV on page 300.

► mostError
See section XL_MOST_ERROR_EV on page 300.

► mostSyncPulse
See section XL_SyncPulse on page 78.

► mostCtrlBusload
See section XL_MOST_CTRL_BUSLOAD_EV on page 303.

► mostAsyncBusload
See section XL_MOST_ASYNC_BUSLOAD_EV on page 304.

12.6.3 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
12.6.4 XL_MOST_STOP

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

Tag: XL_MOST_STOP

12.6.5 XL_MOST_EVENT_SOURCE_EV

Syntax:
```c
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

12.6.6 XL_MOST_ALLBYPASS_EV

Syntax:
```c
typedef struct s_xl_most_all_bypass {
    unsigned int bypassState;
} XL_MOST_ALLBYPASS_EV;
```

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 open.

Tag: XL_MOST_ALLBYPASS

12.6.7 XL_MOST_TIMING_MODE_EV

Syntax:
```c
typedef struct s_xl_most_timing_mode {
    unsigned int timingmode;
} XL_MOST_TIMING_MODE_EV;
```

Description: Reports state of master/slave bits (see xlMostSetTimingMode(), xlMostGetTimingMode()).
Parameters

- timingmode
  XL_MOST_TIMING_SLAVE
  XL_MOST_TIMING_MASTER

Tag

XL_MOST_TIMINGMODE

12.6.8 XL_MOST_TIMING_MODE_SPDIF_EV

Syntax

```c
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 xlMostSetTimingMode(), xlMostGetTimingMode()).

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

12.6.9 XL_MOST_FREQUENCY_EV

Syntax

```c
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

12.6.10 XL_MOST_REGISTER_BYTES

Syntax

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

Description
This event is returned after a read or write request (see xlMostReadRegister() and xlMostWriteRegister()).

Parameters
► number
   Number of bytes (max 16).
► address
   Start address of the data.
► value
   Requested data.

Tag
XL_MOST_REGISTER_BYTES

12.6.11 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 xlMostWriteRegisterBit on page 258).

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

12.6.12 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;
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_APA_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 251).

- **register_bNAL**
  - Node address low byte (see section Specific OS8104 Registers on page 251).

- **register_bGA**
  - Group address (see section Specific OS8104 Registers on page 251).

- **register_bAPAH**
  - Alternate packet address high byte (see section Specific OS8104 Registers on page 251).

- **register_bAPAL**
  - Alternate packet address low byte (see section Specific OS8104 Registers on page 251).

- **register_bNPR**
  - Node position register (see section Specific OS8104 Registers on page 251).
  - Maximum position register (see section Specific OS8104 Registers on page 251).
  - Node delay register (see section Specific OS8104 Registers on page 251).

- **register_bMDR**
  - Maximum delay register (see section Specific OS8104 Registers on page 251).

- **register_bSBC**
  - Synchronous bandwidth control (see section Specific OS8104 Registers on page 251).

- **register_bXTIM**
  - Transmit retry time register (see section Specific OS8104 Registers on page 251).

- **register_bXRTY**
  - Transmit retry register (see section Specific OS8104 Registers on page 251).

**Tag**

`XL_MOST_SPECIAL_REGISTER`
12.6.13 XL_MOST_CTRL_SPY_EV

Syntax

```c
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 biphase 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.

12.6.14 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 0xFF (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 sent.

- XL_MOST_TX_TIMEOUT
  Error while transmitting to the OS8104 or switched off os8104 events (see section xlMostSwitchEventSources on page 252).
12.6.15 XL_MOST_CTRL_TX

Description  See section XL_MOST_CTRL_MSG_EV on page 295.

Tag  XL_MOST_CTRL_TX

12.6.16 XL_MOST_ASYNC_MSG_EV

Syntax
```c
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:
    - \((node\ \text{position} \times 2) + 1\).

- **length**
  - Databytes + 2 Byte in quadlets (4 Bytes).

- **targetAddress**
  - Unused.

- **sourceAddress**
  - Unused.

- **asyncData**
  - Unused.

Tag  XL_MOST_ASYNC_MSG

12.6.17 XL_MOST_ASYNC_TX_EV

Syntax
```c
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 xlMostAsyncTransmit()).

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

### 12.6.18 XL_MOST_SYNC_ALLOC_EV

**Syntax**
```c
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 xlMostSyncGetAllocTable() (userHandle != 0).

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

Tag
XL_MOST_SYNC_ALLOCTABLE

### 12.6.19 XL_MOST_SYNC_VOLUME_STATUS_EV

**Syntax**
```c
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
12.6.20 XL_MOST_RX_LIGHT_EV

**Syntax**
```c
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 xlMostGetRxLight() 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

12.6.21 XL_MOST_TX_LIGHT_EV

**Syntax**
```c
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 xlMostSetTxLight() and xlMostGetTxLight() (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

12.6.22 XL_MOST_LOCK_STATUS_EV

**Syntax**
```c
typedef struct s_xl_most_lock_status {
    unsigned int LockStatus;
} XL_MOST_LOCK_STATUS_EV;
```
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

### 12.6.23 XL_MOST_ERROR_EV

**Syntax**

```c
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_SYNCPULSE_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

### 12.6.24 XL_MOST_RX_BUFFER_EV

**Syntax**

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

This event confirms the xlMostCtrlRxBuffer() call.

Parameters

- **mode**
  - 0
    - Off.
  - 1
    - Simulation of full Rx buffer on.

Tag

XL_MOST_CTRL_RXBUFFER

**12.6.25 XL_MOST_CTRL_SYNC_AUDIO_EV**

Syntax

```c
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 xlMostCtrlSyncAudio() 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 xlMostCtrlSyncAudio on page 261

Tag

XL_MOST_CTRL_SYNC_AUDIO

**12.6.26 XL_MOST_CTRL_SYNC_AUDIO_EX**

Syntax

```c
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 xlMostCtrlSyncAudioEx() 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
12 MOST Commands

12.6.27 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`

12.6.28 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`

12.6.29 XL_MOST_GEN_LIGHT_ERROR_EV

Syntax

```
typedef struct s_xl_most_gen_light_error {
    unsigned int lightOnTime;
    unsigned int lightOffTime;
} XL_MOST_GEN_LIGHT_ERROR_EV;
```
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 xlMostGenerateLightError on page 268).

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

### 12.6.30 XL_MOST_GEN_LOCK_ERROR_EV

**Syntax**
```c
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 xlMostGenerateLockError on page 269).

**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 xlMostGenerateLockError ()).

Tag
XL_MOST_GENLOCKERROR

### 12.6.31 XL_MOST_CTRL_BUSLOAD_EV

**Syntax**
```c
typedef struct s_xl_most_ctrl_busload {
    unsigned int busloadCtrlStarted;
} XL_MOST_CTRL_BUSLOAD_EV;
```
Description

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

Parameters

- **busloadCtrlStarted**
  
  **XL_MODE_ACTIVATE**
  Busload test started.
  
  **XL_MODE_DEACTIVATE**
  Busload test stopped.

Tag

`XL_MOST_CTRL_BUSLOAD`

### 12.6.32 XL_MOST_ASYNC_BUSLOAD_EV

Syntax

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

Description

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

Parameters

- **busloadAsyncStarted**
  
  **XL_MODE_ACTIVATE**
  Busload test started.
  
  **XL_MODE_DEACTIVATE**
  Busload test stopped.

Tag

`XL_MOST_ASYNC_BUSLOAD`

### 12.6.33 XL_MOST_STREAM_BUFFER

Syntax

```c
typedef struct s_xl_most_stream_buffer {
  unsigned int streamHandle;
  unsigned char *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).
12.6.34 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

12.6.35 XL_MOST_SYNC_TX_UNDERFLOW_EV

Syntax

typedef struct s_xl_most_sync_tx_underflow {

status
XL_SUCCESS
OK

XL_BUFFER_ERROR
Data is lost

pBuffer_highpart
The upper DWORD of the data pointer on 64 bit systems.
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 xlMostStreamOpen()).
► reserved
For future use.

Tag
XL_MOST_SYNC_TX_UNDERFLOW

12.6.36 XL_MOST_SYNC_RX_OVERFLOW_EV

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 xlMostStreamOpen()).
► reserved
For future use.

Tag
XL_MOST_SYNC_RX_OVERFLOW
12.7 Application Examples

12.7.1 xlMOSTView

12.7.1.1 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 devices 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.
12.7.1.2 Classes

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

12.7.1.3 Functions

<table>
<thead>
<tr>
<th>Description</th>
<th>► CGeneral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contains only general functions for handling, e. g. string converting.</td>
</tr>
</tbody>
</table>
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 store 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.
13 MOST 150 Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 Introduction</td>
<td>312</td>
</tr>
<tr>
<td>13.2 Flowchart</td>
<td>313</td>
</tr>
<tr>
<td>13.3 Functions</td>
<td>315</td>
</tr>
<tr>
<td>13.4 Structs</td>
<td>352</td>
</tr>
<tr>
<td>13.5 Events</td>
<td>359</td>
</tr>
<tr>
<td>13.6 Application Examples</td>
<td>391</td>
</tr>
</tbody>
</table>
13.1 Introduction

Description

The XL Driver Library enables the development of MOST applications for supported Vector devices (see section System Requirements on page 34). A MOST application always requires init access (see section xlOpenPort on page 44) multiple MOST applications cannot use a common physical MOST channel at the same time.

Depending on the channel property init access (see page 31), 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>44.1 kHz</td>
</tr>
<tr>
<td>Node address</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>Group address</td>
<td>0x300</td>
</tr>
<tr>
<td>MAC address</td>
<td>0xFFFFFFFFFFFF</td>
</tr>
</tbody>
</table>
### 13.2 Flowchart

#### Calling sequence

![Flowchart Diagram]

*Figure 35: Function calls for MOST150 applications (1/2)*
Calling sequence

Measurement (2/2)

- xlDeactivateChannel()
- xlClosePort()
- xlCloseDriver()

end

Function()

Special MOST API Functions
- xlMost150AsyncTransmit()
- xlMost150EthernetTransmit()
- xlMost150GenerateBypassStress()
- xlMost150CtrlTransmit()
- xlMost150Receive()
- xlMost150SyncGetAllocTable()
- xlFlushReceiveQueue()
- xlGetReceiveQueueLevel()
- xlMost150SyncGetVolume()
- xlMost150SyncGetMute()
- xlMost150GetTxLight()
- xlMost150GetRxLightLockStatus()
- xlMost150SGetECLInfo()
- xlMost150StreamInitRxFifo()
- xlMost150StreamOpen()
- xlMost150StreamStart()
- xlMost150StreamTransmitData()
- xlMost150StreamClearTxFifo()
- xlMost150StreamGetInfo()
- xlMost150TransmitData()
- xlMost150ReceiveData()
- xlMost150StreamStop()
- xlMost150StreamClose()

Common API function
- xDeactivateChannel()
- xClosePort()
- xCloseDriver()

end

Figure 36: Function calls for MOST 150 applications (2/2)
13.3 Functions

13.3.1 xlMost150SwitchEventSources

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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).

- **XL_MOST150_SOURCE_SPECIAL_NODE**
  Switch on the `XL_MOST150_SPECIAL_NODE_INFO_EV` events.

- **XL_MOST150_SOURCE_LIGHTLOCK_INIT**
  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.
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 482).

13.3.2 xlMost150SetDeviceMode

Syntax
XLstatus xlMost150SetDeviceMode (  
  XLportHandle portHandle,  
  XLaccess accessMask,  
  XLuserHandle userHandle,  
);
```c
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 xlMost150Shutdown() 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 `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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 482).

### 13.3.3 `xlMost150GetDeviceMode`

**Syntax**
```c
XLstatus xlMost150GetDeviceMode (  
    XLPortHandle portHandle,  
    XLaccess accessMask,  
    XLUserHandle userHandle  
)
```

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

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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 482).

13.3.4 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 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

spdifMode
Describes the S/PDIF mode.

Return event
XL_MOST150_SPDIFMODE

Return value
Returns an error code (see section Error Codes on page 482).

13.3.5 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

Return event

`XL_MOST150_SPDIFMODE`

Return value

Returns an error code (see section Error Codes on page 482).

### 13.3.6 `xlMost150SetSpecialNodeInfo`

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

► **pSpecialNodeInfo**
  Contains all data (see section `XLmost150SetSpecialNodeInfo` on page 355).

**Return event**

`XL_MOST150_SPECIAL_NODE_INFO`

**Return value**

Returns an error code (see section Error Codes on page 482).

### 13.3.7 `xlMost150GetSpecialNodeInfo`

**Syntax**

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

...
13.3.8 xlMost150SetFrequency

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

13.3.9 xlMost150GetFrequency

Syntax
XLstatus xlMost150GetFrequency (  
  XLportHandle portHandle,  
  XLaccess 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).
13.3.10 \texttt{xlMost150GetSystemLockFlag}

\textbf{Syntax}

\begin{verbatim}
XLstatus xlMost150GetSystemLockFlag ( 
     XLportHandle portHandle, 
     XLaccess accessMask, 
     XLuserHandle userHandle, 
)
\end{verbatim}

\textbf{Description}

Requests the state of the \texttt{SystemLock} flag detected by the spy.

\textbf{Input parameters}

\begin{itemize}
    \item \texttt{portHandle}
        The port handle retrieved by \texttt{xlOpenPort}().
    \item \texttt{accessMask}
        The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the \cite{VectorHardwareConfiguration} tool if there is a prepared application setup (see section \texttt{xlGetChannelMask} on page 43). For further information on channel/access masks please also refer to section \texttt{Principles of the XL Driver Library} on page 29.
    \item \texttt{userHandle}
        The handle is created by the application and is used for the event assignment.
\end{itemize}

\textbf{Return event}

\texttt{XL\_MOST150\_SYSTEMLOCK\_FLAG}

\textbf{Return value}

Returns an error code (see section \texttt{Error Codes} on page 482).

13.3.11 \texttt{xlMost150GetShutdownFlag}

\textbf{Syntax}

\begin{verbatim}
XLstatus xlMost150GetShutdownFlag ( 
     XLportHandle portHandle, 
     XLaccess accessMask, 
     XLuserHandle userHandle 
)
\end{verbatim}

\textbf{Description}

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

\textbf{Input parameters}

\begin{itemize}
    \item \texttt{portHandle}
        The port handle retrieved by \texttt{xlOpenPort}().
    \item \texttt{accessMask}
        The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the \cite{VectorHardwareConfiguration} tool if there is a prepared application setup (see section \texttt{xlGetChannelMask} on page 43). For further information on channel/access masks please also refer to section \texttt{Principles of the XL Driver Library} on page 29.
    \item \texttt{userHandle}
        The handle is created by the application and is used for the event assignment.
\end{itemize}

\textbf{Return event}

\texttt{XL\_MOST150\_SHUTDOWN\_FLAG}

\textbf{Return value}

Returns an error code (see section \texttt{Error Codes} on page 482).
13.3.12 xlMost150Shutdown

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

13.3.13 xlMost150Startup

Syntax

```c
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

- **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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

Return event

`XL_MOST150_NW_STARTUP`
Return value

Returns an error code (see section Error Codes on page 482).

### 13.3.14 xlMost150SetSSOResult

**Syntax**

```c
xlStatus xlMost150SetSSOResult (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLuserHandle userHandle,  
    unsigned int ssoCUStatus
)
```

**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**

- **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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

- **ssoCUStatus**
  SSO result value to be set. Only `XL_MOST150_SSO_RESULT_NO_RESULT` is allowed.

**Return event**

`XL_MOST150_SSO_RESULT`

**Return value**

Returns an error code (see section Error Codes on page 482).

### 13.3.15 xlMost150GetSSOResult

**Syntax**

```c
xlStatus xlMost150GetSSOResult (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLuserHandle userHandle,
)
```

**Description**

Requests the stored SSO result value.

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
userHandle
   The handle is created by the application and is used for the event assignment.

Return event  XL_MOST150_SSO_RESULT
Return value  Returns an error code (see section Error Codes on page 482).

13.3.16 xlMost150CtrlTransmit

Syntax

```
xLstatus xlMost150CtrlTransmit (  
   XLportHandle portHandle,  
   XLaccess accessMask,  
   XLuserHandle userHandle,  
   XLmost150CtrlTxMsg *pCtrlTxMsg  
)  
```

Description  Transmits a message over the control channel. The transmit confirmation is reported as XL_MOST150_CTRL_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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

- **pCtrlTxMsg**
  Control message to be transmitted (see section XLmost150CtrlTxMsg on page 354).

Return event  XL_MOST150_CTRL_TX_ACK
Return value  Returns an error code (see section Error Codes on page 482).

13.3.17 xlMost150AsyncTransmit

Syntax

```
xLstatus xlMost150AsyncTransmit (  
   XLportHandle portHandle,  
   XLaccess 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().
13.3.18 xlMost150EthernetTransmit

**Syntax**

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

**Description**

Transmits an Ethernet packet (MEP) over the asynchronous channel and returns the point of time of transmission as confirmation. The transmit confirmation is reported as XL_MOST150ETHERNET_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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 362).

**Return event**

XL_MOST150ASYNC_TX_ACK

**Return value**

Returns an error code (see section Error Codes on page 482).

13.3.19 xlMost150SyncGetAllocTable

**Syntax**

```c
XLstatus xlMost150SyncGetAllocTable ( 
)
```

**Return event**

XL_MOST150ETHERNET_TX_ACK

**Return value**

Returns an error code (see section Error Codes on page 482).
13.3.20  xlMost150CtrlSyncAudio

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

► 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

► pSyncAudioParameter
   Audio parameter to be transmitted (see section XLmost150SyncAudioParameter on page 357).

Return event

XL_MOST150_CTRL_SYNC_AUDIO

Return value

Returns an error code (see section Error Codes on page 482).
### 13.3.21 xlMost150SyncSetVolume

**Syntax**

```c
XLstatus xlMost150SyncSetVolume (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLUserHandle userHandle,  
    unsigned int device,  
    unsigned int volume
)
```

**Description**

Sets the input gain of the device (line in/out). 100% means maximum level, 0% minimum level (no 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

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

- **device**
  - `XL_MOST150_DEVICE_LINE_IN`
  - `XL_MOST150DEVICE_LINE_OUT`

- **volume**
  Value range 0...255 (means 0%...100%).

**Return event**

`XL_MOST150_SYNC_VOLUME_STATUS`

**Return value**

Returns an error code (see section Error Codes on page 482).

### 13.3.22 xlMost150SyncGetVolume

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
13.3.23 xlMost150SyncSetMute

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

13.3.24 xlMost150SyncGetMute

Syntax

```c
XLstatus xlMost150SyncGetMute (
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int device
)
```
13.3.25 xlMost150GetRxLightLockStatus

Syntax

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

Description

Requests light & lock state either from INIC (light state at FOR and the PLL state) or from 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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

XL_MOST150_RXLIGHT_LOCKSTATUS

The flagsChip member in the event header determines whether the event is from spy (see flagsChip parameter values).

Return value

Returns an error code (see section Error Codes on page 482).

13.3.26 xlMost150SetTxLight

Syntax

XLstatus xlMost150SetTxLight (  
XLportHandle portHandle,  
XLaccess accessMask,  
XLuserHandle userHandle,  
unsigned int 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

13.3.27 xlMost150GetTxLight

Syntax

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

Description

Requests light status at FOT.

Input parameters

► portHandle
  The port handle retrieved by xlOpenPort().
13.3.28 xlMost150SetTxLightPower

Syntax

```c
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 `xlGetChannelMask` on page 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 482).

13.3.29 xlMost150GenerateLightError

Syntax

```c
XLstatus xlMost150GenerateLightError (  
    XLportHandle portHandle,
    XLaccess accessMask,
    XLuserHandle userHandle,
    unsigned int lightOffTime,
    unsigned int lightOnTime,
    unsigned int repeat
)
```

Return event
`XL_MOST150_LIGHT_POWER`

Return value
Returns an error code (see section Error Codes on page 482).
13 MOST 150 Commands

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 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 482).

13.3.30 xMost150GenerateLockError

Syntax

```c
XLstatus xMost150GenerateLockError ( 
    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().
13.3.31 xlMost150CtrlConfigureBusload

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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.
13.3.32 xlMost150CtrlGenerateBusload

Syntax

```c
XLstatus xlMost150CtrlGenerateBusload (  
    XLportHandle   portHandle,  
    XLaccess       accessMask,  
    XLuserHandle   userHandle,  
    unsigned long numberCtrlFrames
)
```

Description

Starts/stops 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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

`XL_MOST150_CTRL_BUSLOAD`

Return value

Returns an error code (see section Error Codes on page 482).

13.3.33 xlMost150AsyncConfigureBusload

Syntax

```c
XLstatus xlMost150AsyncConfigureBusload (  
    XLportHandle   portHandle,  
    XLaccess       accessMask,  
    XLuserHandle   userHandle,  
    XLmost150AsyncBusloadConfig *pAsyncBusLoad
)
```

Description

Configures busload generation of MOST Data or Ethernet packets.

Input parameters

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

13.3.34 xlMost150AsyncGenerateBusload

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).
13.3.35 `xlMost150ConfigureRxBuffer`

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

13.3.36 `xlMost150GenerateBypassStress`

**Syntax**

```c
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.
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

## 13.3.37 xlMost150SetECLLine

### Syntax

```c
XLstatus xlMost150SetECLLine (  
  XLportHandle portHandle, 
  XLaccess accessMask, 
  XLuserHandle userHandle, 
  unsigned int eclLineState
)
```

### Description

Sets the state of the ECL (high or low level).

### Note

In case the ECL is pulled down to low level by another device, it cannot be pulled up to high level!
13 MOST 150 Commands

► 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

13.3.38 xlMost150SetECLTermination

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

13.3.39 xlMost150GetECLInfo

Syntax
XLstatus xlMost150GetECLInfo (  
XLportHandle portHandle,

XLportHandle portHandle,
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► 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 482).

13.3.40 xlMost150ECLConfigureSeq

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► userHandle
The handle is created by the application and is used for the event assignment.
13 MOST 150 Commands

- **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 482).

### 13.3.41 xlMost150ECLGenerateSeq

**Syntax**
```c
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 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

### 13.3.42 xlMost150SetECLGlitchFilter
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **userHandle**
The handle is created by the application and is used for the event assignment.
- **duration**
  Duration (in µs) of glitches to be filtered. Value range: 50 µs .. 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 482).
13.3.43 Streaming

13.3.43.1 General Information

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.

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 xlMost150StreamOpen(). 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 `xlMost150StreamTransmitData()` before starting the stream, thus avoiding to stream “0” data initially just after starting the stream.

4. The stream is started by calling `xlMost150StreamStart()`. 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 `xlMost150StreamTransmitData()`. This cyclic notification is done until the stream is stopped.

6. The stream is stopped by calling `xlMost150StreamStop()`. This is acknowledged with an `XL_MOST150_STREAM_STATE` event (state = XL_MOST150_STREAM_STATE_STOPPED).

7. The stream is closed by calling `xlMost150StreamClose()`. 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 xlMost150StreamInitRxFifo() once to initialize the Rx FIFO.

2. Initially a stream has to be opened by calling xlMost150StreamOpen(). 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 xlMost150StreamStart(). 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 xlMost150StreamReceiveData(). This cyclic notification is done until the stream is stopped.

6. The stream is stopped by calling xlMost150StreamStop(). 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 xlMost150StreamClose(). 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 xlMost150StreamClearTxFifo(). 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.

13.3.43.2 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 xlMost150StreamStart()). 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 xlMost150StreamStart().
13.3.44 xlMost150StreamOpen

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

13.3.45 xlMost150StreamClose

Syntax

```c
XLstatus xlMost150StreamClose ( 
    XLportHandle portHandle, 
    XLaccess accessMask, 
    XLuserHandle userHandle, 
    unsigned int streamHandle 
)
```

Description

Closes an opened stream (Tx / Rx) used for routing synchronous data to or from the MOST bus (synchronous channel). Additionally for a Tx stream, the allocated band-
width 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 43). For further information on channel/access masks please also refer to section *Principles of the XL Driver Library* on page 29.

- **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 482).

### 13.3.46 `xlMost150StreamStart`

**Syntax**

```c
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 `xlGetChannelMask` on page 43). For further information on channel/access masks please also refer to section *Principles of the XL Driver Library* on page 29.

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

- **streamHandle**
  Stream handle (returned by `xlMost150StreamOpen()`).
13.3.47 xlMost150StreamStop

Syntax

```c
XLstatus xlMost150StreamStop (  
    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 `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 43). For further information on channel/access masks please also refer to section `Principles of the XL Driver Library` on page 29.
- **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 482).

13.3.48 xlMost150StreamTransmitData

Syntax

```c
XLstatus xlMost150StreamTransmitData (  
    XLportHandle portHandle,  
    XLaccess accessMask,  
    XLuserHandle userHandle,  
    unsigned int streamHandle,  
    unsigned char* pBuffer,  
    unsigned int* pNumberOfBytes  
)
```
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

### 13.3.49 xlMost150StreamClearTxFifo

#### Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
13.3.50 xlMost150StreamInitRxFifo

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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

Return event
None.

Return value
Returns an error code (see section Error Codes on page 482).

13.3.51 xlMost150StreamReceiveData

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().
13.3.52 \texttt{xlMost150StreamGetInfo}

\textbf{Syntax}

\begin{verbatim}
XLstatus xlMost150StreamGetInfo ( 
    XLportHandle     portHandle, 
    XLaccess         accessMask, 
    XLuserHandle     userHandle, 
    XLmost150StreamInfo* pStreamInfo
)
\end{verbatim}

\textbf{Description}

This function retrieves the streaming information of the respective stream determined by the \textit{streamHandle} parameter. In case the stream is closed there is no valid stream handle and the function return an error \texttt{XL\_ERR\_WRONG\_PARAMETER}.

\textbf{Input parameters}

\begin{itemize}
  \item \textbf{portHandle} \\
      The port handle retrieved by \texttt{xlOpenPort}().
  \item \textbf{accessMask} \\
      The access mask specifies the channels to be accessed. Typically, the access mask can be directly retrieved from the \textbf{Vector Hardware Configuration} tool if there is a prepared application setup (see section \texttt{xlGetChannelMask} on page 43). For further information on channel/access masks please also refer to section \textbf{Principles of the XL Driver Library} on page 29.
\end{itemize}

\textbf{Output parameters}

\begin{itemize}
  \item \textbf{pStreamInfo} \\
      Pointer to structure \texttt{XLmost150StreamInfo}.
\end{itemize}

\textbf{Return event} None.

\textbf{Return value} Returns an error code (see section \textbf{Error Codes} on page 482).
13.3.53 xlMost150Receive

Syntax

```c
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 359).
  
  Buffer size: `XL_MOST150_MAX_EVENT_DATA_SIZE`.

Return event

None.

Return value

Returns an error code (see section Error Codes on page 482).

13.3.54 xlMost150TwinklePowerLed

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).
13.4 Structs

13.4.1 XLmost150AsyncBusloadConfig

**Syntax**

```c
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_TYPEETHERNET_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`:

<table>
<thead>
<tr>
<th>Counter Type</th>
<th>MDP</th>
<th>MEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Byte</td>
<td>0..1523</td>
<td>0..1505</td>
</tr>
<tr>
<td>2 Byte</td>
<td>1..1523</td>
<td>1..1505</td>
</tr>
<tr>
<td>3 Byte</td>
<td>2..1523</td>
<td>2..1505</td>
</tr>
<tr>
<td>4 Byte</td>
<td>3..1523</td>
<td>3..1505</td>
</tr>
</tbody>
</table>

- **busloadAsyncPkt**
  See section XLmost150AsyncTxMsg on page 361.

- **busloadEthernetPkt**
  See section XLmost150EthernetTxMsg on page 362.

13.4.2 XLmost150AsyncTxMsg

**Syntax**

```c
typedef struct s_xl_most150_async_tx_msg {
```

[...rest of the document...]
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 `xILmost150SetSpecialNodeInfo()` 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).

13.4.3 **xILmost150CtrlBusloadConfig**

Syntax
typedef struct s_xl_most150_ctrl_busload_config {
    unsigned int  transmissionRate;
    unsigned int  counterType;
    unsigned int  counterPosition;
} 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.
13.4.4 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

13.4.5 XLmost150EthernetTxMsg

Syntax

typedef struct s_xl_most150_ethernet_tx_msg {
  unsigned int priority;
  unsigned int ethSendAttempts;
  unsigned int sourceAddress[6];
  unsigned int targetAddress[6];
  unsigned int length;
  unsigned char ethernetData[XL_MOST150_ETHETERNET_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).

13.4.6 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..0x0FEF, 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.

13.4.7 XLmost150StreamInfo

**Syntax**
```c
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.

13.4.8 XLmost150StreamOpen

**Syntax**
```c
typedef struct s_xl_most150_stream_open {
    unsigned int* pStreamHandle;
    unsigned int direction;
    unsigned int numBytesPerFrame;
} XLmost150StreamOpen;
```
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

13.4.9 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 (0xFFF) 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
13.5 Events

13.5.1 XLmost150event

Syntax

```c
struct s_xl_event_most150 {
    unsigned int size;
    XLmostEventTag tag;
    unsigned short channelIndex;
    unsigned int userHandle;
    unsigned short flagsChip;
    unsigned short reserved;
    XLuint64 time_stamp;
    XLuint64 time_stamp_sync;

    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_MOST150ETHERNETSpy_EV mostEthernetSpy;
        XL_MOST150ETHERNETRx_EV mostEthernetRx;
        XL_MOST150SYSTEMLOCK_FLAG_EV mostSystemLockFlag;
        XL_MOST150SHUTDOWN_FLAG_EV mostShutdownFlag;
        XL_MOST150NW_STARTUP_EV mostStartup;
        XL_MOST150NW_SHUTDOWN_EV mostShutdown;
        XL_MOST150_ECL_EV mostEclEvent;
        XL_MOST150_ECLTERMINATION_EV mostEclTermination;
        XL_MOST150ECLSEQUENCE_EV mostEclSequence;
        XL_MOST150ECL_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_MOST150GEN_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.

13.5.2 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
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:

<table>
<thead>
<tr>
<th>Counter Type</th>
<th>Counter Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDP</td>
<td>MEP</td>
</tr>
<tr>
<td>1 Byte</td>
<td>0..1523</td>
</tr>
<tr>
<td>2 Byte</td>
<td>1..1523</td>
</tr>
<tr>
<td>3 Byte</td>
<td>2..1523</td>
</tr>
<tr>
<td>4 Byte</td>
<td>3..1523</td>
</tr>
</tbody>
</table>

busloadAsyncPkt
See section XLmost150AsyncTxMsg on page 361.

busloadEthernetPkt
See section XLmost150EthernetTxMsg on page 362.

13.5.3 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 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 bytes.

targetAddress
Logical target address of the data packet.
13.5.4 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_MOST150ETHERNETSEND_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_MOST150ETHERNET_TX_ACK will report a maximum of 1506 byte.

► ethernetData
Payload of the Ethernet packet (depends on length).

13.5.5 XL_START

Description
This event is returned after an xlActivateChannel() 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 359.

13.5.6 XL_STOP

Description
This event is returned after an xlDeactivateChannel() function call, without event data.

Tag
XL_STOP
See s_xl_event_most150.tag in section XLmost150event on page 359.
13 MOST 150 Commands

13.5.7 XL_MOST150_EVENT_SOURCE_EV

Syntax

```c
typedef struct s_xl_most150_event_source{
    unsigned int sourceMask;
} XL_MOST150_EVENT_SOURCE_EV;
```

Description

This event is returned after `xlMost150SwitchEventSources()`. Parameters

- `sourceMask`

```c
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_BUSLOADASYNC
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 359.

13.5.8 XL_MOST150_DEVICE_MODE_EV

Syntax

```c
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 `xlMost150SetDeviceMode()`), `xlMost150GetDeviceMode()`).

Parameters

- `deviceMode`

```c
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 359.
13.5.9 XL_MOST150_SPDIF_MODE_EV

**Syntax**

```c
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 xlMost150SetSPDIFMode(), xlMost150GetSPDIFMode()).

**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 359.

13.5.10 XL_MOST150_FREQUENCY_EV

**Syntax**

```c
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 359.

13.5.11 XL_MOST150_SPECIAL_NODE_INFO_EV

**Syntax**

```c
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_NPRSpy_CHANGED
  XL_MOST150_MPRSpy_CHANGED
  XL_MOST150_SBCSpy_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.
**inicNISate**
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 359.

### 13.5.12 XL_MOST150_CTRL_SPY_EV

**Syntax**
```c
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_PACKET
XL_MOST150_VALID_CACK
XL_MOST150_VALID_PINDEX
XL_MOST150_VALID_PRIORITY
XL_MOST150_VALID_CRC
XL_MOST150_VALID_CRCRCALCULATED
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_CTRLSpy
13.5.13 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 tellId;
    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.
- **tellId**
  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_EV
See s_xl_event_most150.tag in section XLmost150event on page 359.

13.5.14 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;
This event reports a transmit acknowledge of a control message. Refer to xlMost150CtrlTransmit().

**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 xlMost150SetSpecialNodeInfo() 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

  ```c
  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 359.

### 13.5.15 XL_MOST150_ASYNC_SPY_EV

```c
typedef struct s_xl_most150_async_spy_msg {
    unsigned in frameCount;
    unsigned int pktDuration;
    unsigned short asyncDataLenAnnounced;
} s_xl_most150_async_spy_msg;
```
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_DATALENAANNOUNCED
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_CRC_CALCULATED
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 s_xl_event_most150.tag in section XLmost150event on page 359.

13.5.16 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 359.
13.5.17 XL_MOST150_ASYNC_TX_ACK_EV

**Syntax**

```c
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 xlMost150AsyncTransmit().

**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, 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 359.

---

13.5.18 XL_MOST150_CL_INFO

**Syntax**

```c
#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 xlMost150SyncGetAllocTable().
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 359.

13.5.19 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 372

13.5.20 XL_MOST150_TX_LIGHT_EV

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 xlMost150SetTxLight() and xlMost150GetTxLight() 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 359.

13.5.21 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 xlMostGetRxLightLockStatus(). The flagsChip value determines whether the event is reported by the node (INIC) or spy.
13 MOST 150 Commands

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 359.

13.5.22 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 359.

13.5.23 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 xlMost150CtrlSyncAudio() 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_MOST150DEVICE_SPDIF_OUT
- XL_MOST150DEVICE_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_MOST150DEVICE_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_MOST150DEVICE_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.
13.5.24 XL_MOST150_SYNC_VOLUME_STATUS_EV

Syntax

```c
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_MOST150DEVICE_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 359.

13.5.25 XL_MOST150_SYNC_MUTE_STATUS_EV

Syntax

```c
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_MOST150DEVICE_LINE_OUT
  - XL_MOST150DEVICE_SPDIF_IN
  - XL_MOST150DEVICE_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 359.

13.5.26 XL_MOST150_LIGHT_POWER_EV

Syntax

```c
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 359.

13.5.27 XL_MOST150_GEN_LIGHT_ERROR_EV

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 xlMost150GenerateLightError()).

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 359.

13.5.28 XL_MOST150_GEN_LOCK_ERROR_EV

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 xlMost150GenerateLockError()).

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 359.

13.5.29 XL_MOST150_CONFIGURE_RX_BUFFER_EV
**13 MOST 150 Commands**

### Syntax

```c
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 359.

#### 13.5.30 XL_MOST150_CTRL_BUSLOAD_EV

### Syntax

```c
typedef struct s-xl_most150_ctrl_busload {
    unsigned int busloadStarted;
} XL_MOST150_CTRL_BUSLOAD_EV;
```

### Description

This is the response event for the `xlMost150CtrlGenerateBusload()` and shows the start/stop of the busload generation. The function `xlMost150CtrlConfigureBusload()` 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 359.

#### 13.5.31 XL_MOST150_ASYNC_BUSLOAD_EV

### Syntax

```c
typedef struct s-xl_most150_async_busload {
    unsigned int busloadStarted;
} XL_MOST150_ASYNC_BUSLOAD_EV;
```
Description
This is the response event on a xlMost150AsyncGenerateBusload() function call and shows the start/stop of the busload generation. The function xlMost150AsyncConfigureBusload() 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 359.

13.5.32 XL_MOST150ETHERNETSpy_EV

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 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_MOST150ETHERNETSpy_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.
13.5.33 XL_MOST150ETHERNETRX_EV

Tag
XL_MOST150 ETHERNET_SPY
See s_xl_event_most150.tag in section XLmost150event on page 359.

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 (NIC).
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_MOST150ETHERNET_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 359.

13.5.34 XL_MOST150_ETHERNET_TX_ACK_EV

Syntax

```c
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 xlMost150EthernetTransmit().

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().

- **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).
13.5.35 XL_MOST150_SYSTEMLOCK_FLAG_EV

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 359.

13.5.36 XL_MOST150_SHUTDOWN_FLAG_EV

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 359.

13.5.37 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 xlMost150Startup()).

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 \texttt{XL\_MOST150\_NW\_STARTUP}
See \texttt{s\_xl\_event\_most150\_tag} in section \texttt{XLmost150event} on page 359.

### 13.5.38 XL\_MOST150\_NW\_SHUTDOWN\_EV

**Syntax**
```c
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 \texttt{xlMost150Shutdown()}).

**Parameters**
- **error**
  \texttt{XL\_MOST150\_SHUTDOWN\_NO\_ERROR}
  Otherwise the respective MOST ErrorCode from INIC is reported.
- **errorInfo**
  \texttt{XL\_MOST150\_SHUTDOWN\_NO\_ERRORINFO}
  Otherwise the respective MOST ErrorInfo from INIC is reported.

**Tag**
\texttt{XL\_MOST150\_NW\_SHUTDOWN}
See \texttt{s\_xl\_event\_most150\_tag} in section \texttt{XLmost150event} on page 359.

### 13.5.39 XL\_MOST150\_ECL\_EV

**Syntax**
```c
typedef struct s_xl_most150_ecl {
  unsigned int eclLineState;
} XL\_MOST150\_ECL\_EV;
```

**Description**
Reports an ECL line signal change.

**Parameters**
- **eclLineState**
  \texttt{XL\_MOST150\_ECL\_LINE\_LOW}
  \texttt{XL\_MOST150\_ECL\_LINE\_HIGH}

**Tag**
\texttt{XL\_MOST150\_ECL\_LINE\_CHANGED}
See \texttt{s\_xl\_event\_most150\_tag} in section \texttt{XLmost150event} on page 359.

### 13.5.40 XL\_MOST150\_ECL\_TERMINATION\_EV

**Syntax**
```c
typedef struct s_xl_most150_ecl_termination {
  unsigned int resistorEnabled;
} XL\_MOST150\_ECL\_TERMINATION\_EV;
```

**Description**
Reports a termination change of ECL.

**Parameters**
- **resistorEnabled**
  \texttt{XL\_MOST150\_ECL\_LINE\_PULL\_UP\_NOT\_ACTIVE}
  \texttt{XL\_MOST150\_ECL\_LINE\_PULL\_UP\_ACTIVE}

**Tag**
\texttt{XL\_MOST150\_ECL\_TERMINATION\_CHANGED}
See \texttt{s\_xl\_event\_most150\_tag} in section \texttt{XLmost150event} on page 359.
13.5.41 XL_MOST150_ECL_SEQUENCE_EV

**Syntax**

```c
typedef struct s_xl_most150_ecl_sequence {
    unsigned int sequenceStarted;
} XL_MOST150_ECL_SEQUENCE_EV;
```

**Description**

This is the response event on an `xlMost150ECLGenerateSeq()` function call and shows the start/stop of the sequence generation. The function `xlMost150ECLConfigureSeq()` 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 359.

13.5.42 XL_MOST150_ECL_GLITCH_FILTER_EV

**Syntax**

```c
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 µs) of glitches to be filtered. Value range: 50 µs .. 50 ms.
  - Default: 1 ms

**Tag**

`XL_MOST150_ECL_GLITCH_FILTER`

See `s_xl_event_most150.tag` in section `XLmost150event` on page 359.

13.5.43 XL_MOST150_STREAM_STATE_EV

**Syntax**

```c
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 `xlMost150StreamOpen()`).
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 or 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 359.
13.5.44 XL_MOST150_STREAM_TX_BUFFER_EV

Syntax

```c
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 xlMost150StreamTransmitData().

Parameters

- **streamHandle**
  Stream handle (returned by xlMost150StreamOpen()).

- **numberOfBytes**
  Number of bytes that can at least by written into the Tx FIFO (see xlMost150StreamTransmitData()).

- **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 xlMost150StreamTransmitData().

  Note: In this case the application should provide at least 2x `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 359.

13.5.45 XL_MOST150_STREAM_TX_LABEL_EV

Syntax

```c
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 xlMost150StreamOpen()).

- **connLabel**
  Connection label of the Tx stream.
13. MOST 150 Commands

► 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 359.

13.5.46 XL_MOST150_STREAM_TX_UNDERFLOW_EV

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 xlMost150StreamOpen()).

Tag
XL_MOST150_STREAM_TX_UNDERFLOW
See s_xl_event_most150.tag in section XLmost150event on page 359.
13.5.47 XL_MOST150_STREAM_RX_BUFFER_EV

Syntax

```c
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 `xlMost150StreamReceiveData()` 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 `xlMost150StreamOpen()`).

- **numberOfBytes**
  Number of bytes available in the Rx FIFO (see `xlMost150StreamReceiveData()`).

- **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.
13 MOST 150 Commands

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 to keep MOST frame alignment.

The CL(s) are provided when calling xlMost150StreamStart() (parameter pConLabels). The first CL corresponds to bit 0, the second to bit 1 and so on:

<table>
<thead>
<tr>
<th>CL0</th>
<th>CL1</th>
<th>CL2</th>
<th>CL3</th>
<th>CL4</th>
<th>CL5</th>
<th>CL6</th>
<th>CL7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 359.

13.5.48 XL_MOST150_GEN_BYPASS_STRESS_EV

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 xlMost150GenerateBypassStress()).

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 xlMost150SetDeviceMode().

Tag
XL_MOST150_GEN_BYPASS_STRESS
See s_xl_event_most150.tag in section XLmost150event on page 359.

13.5.49 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 xlMost150GetSSOResult() call. The event stores the reason for a MOST network
shutdown.

**Parameters**

- **status**
  - `XL_MOST150_SSO_RESULT_NO_RESULT`
    No result available or reset (see `xlMost150SetSSOResult()`).
  - `XL_MOST150_SSO_RESULT_NO_FAULT_SAVED`
    No fault saved - normal MOST network shutdown.
  - `XL_MOST150_SSO_RESULT_SUDDEN_SIGNAL_OFF`
    Sudden signal off detected.
  - `XL_MOST150_SSO_RESULT_CRITICAL_UNLOCK`
    Critical unlock detected.

**Tag**

- `XL_MOST150_SSO_RESULT`
  See `s_xl_event_most150.tag` in section `XLmost150event` on page 359.
13.6 Application Examples

13.6.1 xlMOST150View

13.6.1.1 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 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.

13.6.1.2 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.

13.6.1.3 Functions

<table>
<thead>
<tr>
<th>Description</th>
<th>CGeneral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contains only general functions for handling, e.g. string converting.</td>
</tr>
</tbody>
</table>
► CMOST150 Functions
Implementation for the XL MOST API handling.

MOST150 Init
Initializes all connected MOST150 devices. For every device a thread is created. Every device gets a separate port which is activated.

MOST150 Close
Close the threads and port handles.

MOST150 Activate
Activates the selected MOST150 channel.

MOST150 Deactivate
Deactivates the selected MOST150 channel.

MOST Ctrl Transmit
Transmits a control frame to the selected channel.

MOST 150 Async Transmit
Transmits an asynchronous frame to the selected channel.

MOST 150 Setup Node
Sets up the MOST node (node group address, device mode and frequency).

MOST 150 Nw Startup
Triggers a network startup.

MOST 150 Nw Shutdown
Triggers a network shutdown.

MOST 150 Get Info
Requests the information of a MOST150 channel (like timing mode, bypass mode...).

MOST 150 Twinkle Power Led
Twinkles the power LEDs.

MOST 150 Generate Light Error
Generates light errors depending on the counter.

MOST 150 Generate Lock Error
Generates lock errors depending on the counter.

► CMOST150 General Test
Handles the dialog box MOST150 General Test.

► CMOST150 Node Config
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.
# 14 FlexRay Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 Introduction</td>
<td>396</td>
</tr>
<tr>
<td>14.2 Flowchart</td>
<td>397</td>
</tr>
<tr>
<td>14.3 Free Library and Advanced Library</td>
<td>398</td>
</tr>
<tr>
<td>14.4 FlexRay Basics</td>
<td>399</td>
</tr>
<tr>
<td>14.5 Functions</td>
<td>405</td>
</tr>
<tr>
<td>14.6 Structs</td>
<td>412</td>
</tr>
<tr>
<td>14.7 Events</td>
<td>420</td>
</tr>
<tr>
<td>14.8 Application Examples</td>
<td>434</td>
</tr>
</tbody>
</table>
14.1 Introduction

Description

The XL Driver Library enables the development of FlexRay applications for supported Vector devices (see section System Requirements on page 34).

Depending on the channel property init access (see page 31), 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.
14.2 Flowchart

Figure 37: Function calls for FlexRay applications
14.3 Free Library and Advanced Library

The XL Driver Library for FlexRay is split into a free and an advanced version. The differences are as follows:

### Init commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Free</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlFrSetConfiguration</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xlFrSetMode</td>
<td>Limited</td>
<td>X</td>
</tr>
<tr>
<td>(only E-Ray can be used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xlFrInitStartupAndSync</td>
<td>Limited</td>
<td>X</td>
</tr>
<tr>
<td>(only E-Ray can be used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xlFrSetupSymbolWindow</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xlSetTimerBasedNotify</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xlFrActivateSpy</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Messages

<table>
<thead>
<tr>
<th>Command</th>
<th>Free</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlFrReceive</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xlFrTransmit</td>
<td>Limited</td>
<td>X</td>
</tr>
<tr>
<td>(only 128 different txFrames can be used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(no PayloadIncrement)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xlFrSendSymbolWindow</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

### General commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Free</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlFrSetTransceiverMode</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xlFrAcceptanceFilter</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

**Note**

The advanced version is unlocked by the Advanced FlexRay Library License or a CANoe/CANalyzer FlexRay license (see section License Management on page 36).
14.4 FlexRay Basics

14.4.1 Introduction

Deterministic and quick data transmission
Implementations of ever more challenging safety and driver-assistance functions go hand in hand with the increasingly more intensive integration of electronic ECUs in the automobile. These implementations require very high data rates to transmit the increasing number of control and status signals. They are signals that not only need to be transmitted extremely quickly; their transmission also needs to be absolutely deterministic.

Fault-tolerant structures required
That is the reason for the growing importance of communication systems that guarantee fast and deterministic data transmission in the automobile. Potential use of by-wire systems further requires the design of fault-tolerant structures and mechanisms. Although by-wire systems may offer wide-ranging capabilities and the benefits of increased design freedom, simplified assembly, personalization of the vehicle, etc., data transmission requirements in the automobile are elevated considerably, because these systems belong to the class of fail-operational systems. They must continue to operate acceptably even when an error occurs.

CAN cannot satisfy these requires due to its event-driven and priority-driven bus access, its limited bandwidth of 500 KBit/sec based on physical constraints in the automobile, and lack of fault-tolerant structures and mechanisms.

14.4.2 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.
14.4.3 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.

Figure 38: 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 independently 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.
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.

Figure 39: Combined topology of passive bus and active star

Figure 40: Passive bus structure with two communication channels minimizes failure risk
14.4.4 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.

Figure 41: 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.
14.4.5 CRC-Protected Data Transmission

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.

**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 42: Structure of the FlexRay message with header, payload and trailer](image)

**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.
14.5 Functions

14.5.1 xlFrSetConfiguration

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **pxlFrClusterConfig**
  Pointer to the cluster config structure (see section `XLfrClusterConfig` on page 412).

Return value

Returns an error code (see section Error Codes on page 482).

14.5.2 xlFrGetChannelConfiguration

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **pxlFrChannelConfig**
  Pointer the config structure (see section `XLfrChannelConfig` on page 417). Contains the cluster configuration parameters.

Return value

Returns an error code (see section Error Codes on page 482).
### 14.5.3 xlFrSetMode

**Syntax**

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **frMode**
  Structure of different operational modes (see section XLfrMode on page 417).

**Return value**

Returns an error code (see section Error Codes on page 482).

### 14.5.4 xlFrInitStartupAndSync

**Syntax**

```c
XLstatus xlFrInitStartupAndSync(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLfrEvent *pEventBuffer)
```

**Description**

Initializes the coldstart and defines the sync frame. The function must be called before xlActivateChannel() and requires init access. To select the channel and CC, use the flagsChip parameter within the basic event structure. To setup different data for FlexRay channels A and B, call it twice. Be sure that the FlexRay config parameters pKeySlotUsedForSync and pKeySlotUsedForStartup are set! The function requires init access.

**Input parameters**

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pEventBuffer**
  Pointer to the event buffer which includes the sync frame (see section XLfrEvent on page 420). It is an XL_FR_TX_FRAME event with set `XL_FR_FRAMEFLAG_SYNC/STARTUP` flag.
14.5.5  xlFrSetupSymbolWindow

Syntax

```c
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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 482).

14.5.6  xlFrActivateSpy

Syntax

```c
XLstatus xlFrActivateSpy(
    XLportHandle portHandle,
    XLaccess accessMask,
    unsigned int mode)
```

Description

In asynchronous mode, all FlexRay frames and symbols are received by the spy, but no frame transmission is possible at all. If this mode is selected, only the baudrate has to be passed in the pxlClusterConfig parameter of xlFrSetConfiguration(), no further FlexRay configuration data is required.

The function call is optional. If this function is not called, the FlexRay frame reception is done by E-Ray after the Vector device node is integrated in the cluster and the
cluster is synchronized.

The function may be called after xlFrSetConfiguration() and requires init access.

Input parameters

► portHandle
  The port handle retrieved by xlOpenPort().

► accessMask
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

► mode
  Mode of the Spy:
  XL_FRSpy_MODEASYNCHRONOUS

Return value

Returns an error code (see section Error Codes on page 482).

14.5.7 xlSetTimerBaseNotify

Syntax

XLstatus xlSetTimerBaseNotify(
  XLportHandle portHandle,
  XLhandle* pHandle)

Description

Sets up an event to notify the application based on the timerate which can be set by xlSetTimerRate() and xlSetTimerRateAndChannel().

Input parameters

► portHandle
  The port handle retrieved by xlOpenPort().

Output parameters

► phandle
  Pointer to a WIN32 event handle.

Return value

Returns an error code (see section Error Codes on page 482).

14.5.8 xlFrReceive

Syntax

XLstatus xlFrReceive(
  XLportHandle portHandle,
  XLfrEvent *pEventBuffer)

Description

Reads one event from the FlexRay receive queue. Calls to xlFrReceive() can be triggered by a notification event (see section xlSetNotification on page 48). An overrun of the receive queue can be determined by the message flag XL_FR_QUEUE_OVERFLOW in XLfrEvent.flagsChip.

Input parameters

► portHandle
  The port handle retrieved by xlOpenPort().

► pEventBuffer
  Pointer to an application buffer in which the received event is copied (see section XLfrEvent on page 420).
14.5.9 xlFrTransmit

Syntax

```c
XLstatus xlFrTransmit(
    XlportHandle portHandle,
    Xlaccess accessMask,
    XLfrEvent *pEventBuffer)
```

Description

The function sends static and dynamic frames with the event tag Tx or can be used for updates in case of cyclic frames. Additionally, a frame payload increment can be configured. To configure different payload increment modes for different frChannels, the function has to be called twice (one time for every channel).

This function can be called before and after channel activation.

Basic conflict checking of the frame configuration is also done by this function. If the frame to be sent conflicts with already configured frames (repetition overlapping / cycle overlapping), the frame is not transmitted and the function returns with error. If the frame to be sent is already configured by another application, the frame is not transmitted and the functions returns with error as well.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pEventBuffer**
  Pointer to the event buffer (see section XLfrEvent on page 420).

  Buffersize: `XL_FR_MAX_EVENT_SIZE`

Return value

Returns an error code (see section Error Codes on page 482).

14.5.10 xlFrSetTransceiverMode

Syntax

```c
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().
14.5.11  xlFrSendSymbolWindow

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **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 482).

14.5.12  xlFrSetAcceptanceFilter

**Syntax**
```c
XLstatus xlFrSetAcceptanceFilter(
    XLportHandle portHandle,
    XLaccess accessMask,
    XLfrAcceptanceFilter *pAcceptanceFilter)
```

**Description**
This function sets the acceptance filter used for the next symbol window.

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.
- **pAcceptanceFilter**
  Points to the acceptance filter.

**Return value**
Returns an error code (see section Error Codes on page 482).
<table>
<thead>
<tr>
<th>Description</th>
<th>This function modifies the acceptance filter for FlexRay frames. The function requires init access.</th>
</tr>
</thead>
</table>
| 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.  
• **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 418). |
| Return value | Returns an error code (see section Error Codes on page 482). |
14.6 Structs

14.6.1 XLfrClusterConfig

Syntax

```c
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 gdMinislots;
    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 gdWakeupSymbolTxDidle;
    unsigned int gdWakeupSymbolTxLow;
    unsigned int gdWakeupSymbolTxWindow;
    unsigned int pdAllowedHaltDueToClock;
    unsigned int pdAllowedPassiveToActive;
    unsigned int pChannels;
    unsigned int pClusterDriftDamping;
    unsigned int pDecodingCorrection;
    unsigned int pDelayCompensationA;
    unsigned int pDelayCompensationB;
    unsigned int pExternalOffsetCorrection;
    unsigned int pExternalRateCorrection;
    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 pdAllowedStartupRange;
    unsigned int pdListenTimeout;
    unsigned int pdMaxDrift;

}```
unsigned int pdMicrotick;
unsigned int gdCASRxLowMax;
unsigned int gChannels;
unsigned int vExternOffsetControl;
unsigned int vExternRateControl;
unsigned int pChannelsMTS;
unsigned int reserved[16];
} XLfrClusterConfig;

**Parameters**

- **busGuardianEnable**
  For future use. Has to be set to 0.

- **baudrate**
  FlexRay baudrate. Supported values are:
  - 10 Mbit: 10.000
  - 5 Mbit: 5.000
  - 2,5 Mbit: 2.500

- **busGuardianTick**
  For future use. Has to be set to 0.

- **externalClockCorrectionMode**
  Not used. Has to be set to 0.

- **gColdStartAttempts**
  Maximum number of times a node in the cluster is permitted to attempt to start the cluster by initiating schedule synchronization.
  Range: 2..31

- **gListenNoise**
  Upper limit for the start up listen timeout and wake up listen timeout in the presence of noise.
  Range: 2..16

- **gMacroPerCycle**
  Number of macroticks in a communication cycle.
  Range: 10..16000.

- **gMaxWithoutClockCorrectionFatal**
  Range 1..15.

- **gMaxWithoutClockCorrectionPassive**
  Range: 1..15.

- **gNetworkManagementVectorLength**
  Length of the NM vector.
  Range: 0..12.

- **gNumberOfMinislots**
  Number of mini slots in the dynamic segment.
  Range: 0..7986.

- **gNumberOfStaticSlots**
  Number of static slots in the static segment.
  Range: 2..1023.

- **gOffsetCorrectionStart**
  Start of the offset correction phase within the NIT, expressed as the number of macro ticks from the start of cycle.
FlexRay Commands

- **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.
FlexRay Commands

► **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.

### 14.6.2 XLfrChannelConfig

**Syntax**

```c
define 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_MODEASYNCHRONOUS

► **reserved**
Reserved for future use.

► **xlFrClusterConfig**
The cluster config (see section XLfrClusterConfig on page 412).

### 14.6.3 XLfrMode

**Syntax**

```c
define 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **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 to do a coldstart leading to initiating the schedule synchronization.

  XL_FR_MODE_COLDSTART_LEADING
  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.

### 14.6.4 XLfrAcceptanceFilter

**Syntax**

```c
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
14.7 Events

14.7.1 XLfrEvent

Syntax

```c
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
In case of events received via RX queue:
Overall size of the event (in bytes).
The maximum size is defined by XL_FR_MAX_EVENT_SIZE.

In case of event used in xlFrTransmit:
Parameter not used.

► Tag
Specifies the FlexRay event type / tag.
XL_FR_START_CYCLE
XL_FR_RX_FRAME
XL_FR_TX_FRAME
XL_FR_TXACK_FRAME
XL_FR_INVALID_FRAME
XL_FR_WAKEUP
XL_FR_SYMBOL_WINDOW
XL_FR_ERROR
XL_FR_STATUS
XL_FR_NM_VECTOR
XL_FR_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 synchroni-
ization 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. Off-
set correction is possible with xlResetClock().

14.7.2 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.

### 14.7.3 XL_FR_RX_FRAME_EV

**Syntax**

```c
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.
**14.7.4 XL_FR_TX_FRAME_EV**

**Syntax**

```c
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.

  ```c
  XL_FR_FRAMEFLAG_SYNC
  Sync bit, set from CC frame buffer. (Only in coldstart mode).
  ```

  ```c
  XL_FR_FRAMEFLAG_STARTUP
  Startup flag, set from CC frame buffer. (Only in coldstart mode).
  ```

  ```c
  XL_FR_FRAMEFLAG_PAYLOAD_PREAMBLE
  Payload preamble bit, set from CC frame buffer.
  ```

  ```c
  XL_FR_FRAMEFLAG_FR_RESERVED
  Reserved by the FlexRay protocol (zero in current FlexRay version V2.1)
  ```

  ```c
  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)*

### 14.7.5 XL_FR_TXACK_FRAME

**Reference**
Same as **XL_FLEXRAY_RX_FRAME**.

### 14.7.6 XL_FR_INVALID_FRAME

**Reference**
Same as **XL_FLEXRAY_RX_FRAME**.

### 14.7.7 XL_FR_WAKEUP_EV

**Syntax**

```c
struct s_xl_fr_wakeup {
  unsigned char cycleCount;
  unsigned char wakeupStatus;
};
```
unsigned char reserved[6];
} XL_FR_WAKEUP_EV;

**Parameters**

- **cycleCount**
  Current cycle count.

- **wakeupStatus**
  
  XL_FR_WAKEUP_UNDEFINED
  No wake up attempt since POC-state XL_FR_STATUS_CONFIG was left. On a received wake up pattern on frChannel A|B, this value will be set.

  XL_FR_WAKEUP_RECEIVED_HEADER
  Set when the CC finishes wake up due to the reception of a frame header without coding violation on either channel in WAKEUP_LISTEN state.

  XL_FR_WAKEUP_RECEIVED_WUP
  Set when the CC finishes wake up due to the reception of a valid wake up pattern on the configured wake up channel in WAKEUP_LISTEN state.

  XL_FR_WAKEUP_COLLISION_HEADER
  Set when the CC stops wake up due to a detected collision during wake up pattern transmission by receiving a valid header on either channel.

  XL_FR_WAKEUP_COLLISION_WUP
  Flag is set if the CC stops wake up due to a detected collision or during wake up pattern transmission by receiving a valid wake up pattern on the configured wake up channel.

  XL_FR_WAKEUP_COLLISION_UNKNOWN
  Set when the CC stops wake up by leaving WAKEUP_DETECT state after 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 wake up pattern.

- **reserved**
  For future use.

### 14.7.8 XL_FR_SYMBOL_WINDOW_EV

**Syntax**

```c
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
14 FlexRay Commands

- **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.

### 14.7.9 XL_FR_ERROR_EV

```
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.

### 14.7.10 XL_FR_ERROR_POC_MODE_EV

```
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.

14.7.11 XL_FR_ERROR_SYNC_FRAMES_BELOWMIN

Description
Not enough sync frames received in cycle.

14.7.12 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.

14.7.13 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.
14 FlexRay Commands

- **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.

14.7.14 **XL.FR_ERROR_NIT_FAILURE_EV**

**Syntax**

```c
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.

14.7.15 **XL.FR_ERROR_CC_ERROR_EV**

**Syntax**

```c
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

### 14.7.16 XL_FR_STATUS_EV

**Syntax**

```c
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_COLDSTARTCONSISTENCY_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_INTEGRATIONCONSISTENCY_CHECK`
   - `XL_FR_STATUS_INITIALIZER_SCHEDULE`
   - `XL_FR_STATUS_ABORT_STARTUP`

► **reserved**
   For future use.

### 14.7.17 XL_FR_NM_VECTOR_EV

**Syntax**

```c
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 412).
14.7.18 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_FRAMEFRAME_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**

14.7.19 XL_FR_Spy_SYMBOL_EV

Syntax

```
typedef struct s_xl_fr_spy_symbol {
    unsigned short lowLength;
    unsigned short reserved;
} XL_FR_Spy_SYMBOL_EV;
```
FlexRay

Parameters

► lowLength
  Length of low part of symbol (WUS, CAS, MTS) in bits.

► reserved
  Reserved for future use.

14.7.20 XL_APPLICATION_NOTIFICATION_EV

Syntax

typedef struct s_xl_application_notification {
  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.
14.8 Application Examples

14.8.1 xlFlexDemo

14.8.1.1 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…).

14.8.1.2 Classes

Description
The example has the following class structure:

► CfFrFunctions
  Implementation of all library functions.

► CfFrParseEvent
  Contains an event parser to display the received events.

14.8.1.3 Functions

Description
► FrInit
  Opens the driver and checks the FlexRay channels.
► FrToggleTrace
  Switches on/off the “Trace Window”.

► FrAddTxFrame
  Calls xlFrTransmit() to add a FlexRay Tx frame.

► FrActivate
  Activates the selected FlexRay channel.

► FrDeactivate
  Deactivates the active FlexRay channel.

Private

► frGetChannelMask
  Gets the device channel masks.

► frInit
  Opens the port.

► frSetConfig
  Sets up the FlexRay cluster configuration.

► frStartUpSync
  Sets up the StartUpAndSync frames depending on the library license.

► frCreateRxThread
  Creates the Rx thread to readout the FlexRay message queue.

14.8.1.4 Events

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| ► 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”.
14.8.2  **xlFlexDemoCmdLine**

### 14.8.2.1 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.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>In order to compile the example, the Microsoft XML parser package MSXML is required.</td>
</tr>
</tbody>
</table>

### 14.8.2.2 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.
14.8.3 Fibex2CSharpReaderDemo

14.8.3.1 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!

14.8.3.2 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.
15 ARINC 429 Commands

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>15.1 Introduction</th>
<th>15.2 Flowchart</th>
<th>15.3 Functions</th>
<th>15.4 Structs</th>
<th>15.5 Events</th>
<th>15.6 Application Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>439</td>
<td>440</td>
<td>441</td>
<td>446</td>
<td>454</td>
<td>462</td>
</tr>
</tbody>
</table>
15.1 Introduction

Description
The XL Driver Library enables the development of ARINC 429 applications for supported Vector devices (see section System Requirements on page 34).

Depending on the channel property init access (see page 31), the application's main features are as follows:

With init access
► Common access.

Without init access
► Not supported. If the application gets no init access on a specific channel, no further function call is possible on the according channel.

Note
Multi-application is not supported. A single ARINC 429 channel can be used by one application exclusively. The used ARINC 429 channel requires init access. A second application gets no init access for the assigned channel. In this case xlOpenPort() returns an error and no messages can be sent or transmitted.

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

Calling sequence

Driver Init

Driver Init

Channel Setup

On Bus

Figure 43: Function calls for ARINC 429 applications
15.3 Functions

15.3.1 xlA429SetChannelParams

Syntax

```c
XLstatus xlA429SetChannelParams(
    XLportHandle    portHandle,
    XLaccess        accessMask,
    XL_A429_PARAMS* pXlA429Params
)
```

Description

Configures basic ARINC 429 parameters. Note that the device does not keep those settings after a restart. This is a synchronous operation and function needs init access.

Input parameters

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

- **accessMask**
  The access mask specifies the channels to be accessed. 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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **pXlA429Params**
  ARINC 429 configuration structure (see section XL_A429_PARAMS on page 446).

**Note**

Each `xlA429SetChannelParams()` call has to be called before `xlActivateChannel()` function call. Parameter changes after `xlActivateChannel()` calls (e.g. bitrate) are not supported. After `xlDeactivateChannel()`, `xlA429SetChannelParams()` can be called again.

**Example**

Configures an A429 channel in Tx channel direction with a bit rate of 100000 bit/s, parity calculation disabled and a default minGap of 4 bit.

```c
XL_A429_PARAMS xlA429Params;
memset(&xlA429Params, 0, sizeof(XL_A429_PARAMS));
xlA429Params.channelDirection = XL_A429_MSG_CHANNEL_DIR_TX;
xlA429Params.data.tx.bitrate = 100000;
xlA429Params.data.tx.minGap = XL_A429_MSG_GAP_4BIT;
xlA429Params.data.tx.parity = XL_A429_MSG_PARITY_DISABLED;

xlStatus = xlA429SetChannelParams(xlPortHandle,
    xlChannelMask,
    &xlA429Params);
```
Example

Configures an A429 channel in Rx channel direction with enabled bit rate detection (expected bit rate should be between minimum bitrate and maximum bitrate), parity calculation disabled and a default minGap of 4 bit.

```c
XL_A429_PARAMS x1A429Params;
memset(&x1A429Params, 0, sizeof(XL_A429_PARAMS));
x1A429Params.channelDirection = XL_A429_MSG_CHANNEL_DIR_RX;
x1A429Params.data.rx.autoBaudrate = XL_A429_MSG_AUTO_BAUDRATE_ENABLED;
x1A429Params.data.rx.minBitrate = 97500;
x1A429Params.data.rx.maxBitrate = 102500;
x1A429Params.data.rx.parity = XL_A429_MSG_PARITY_DISABLED;
x1A429Params.data.rx.minGap = XL_A429_MSG_GAP_4BIT;
x1status = x1A429SetChannelParams(x1PortHandle, x1ChannelMask, &x1A429Params);
```

15.3.2 x1A429Transmit

**Syntax**

```c
XLstatus x1A429Transmit(
    XLportHandle portHandle,
    XLaccess accessMask,
    unsigned int msgCnt,
    unsigned int* pMsgCntSent,
    XL_A429_MSG_TX* pXIA429MsgTx
)
```

**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 x1A429Transmit() call. This function is an asynchronous operation.

**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 43). For further information on channel/access masks please also refer to section Principles of the XL Driver Library on page 29.

- **msgCnt**
  Amount of messages to be transmitted.

- **pXIA429MsgTx**
  Points to a user buffer with messages to be transmitted, e.g. XL_A429_MSG_TX x1A429MsgTx[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 446).

**Output parameters**

- **pMsgCntSent**
  Number of messages successfully transferred to the transmit queue.
Return value

Returns an error code (see section Error Codes on page 482).

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.

```c
XL_A429_MSG_TX xLA429MsgTx;
unsigned int msgCnt = 1;
unsigned int msgCntSent = 0;
memset(&xLA429MsgTx, 0, sizeof(XL_A429_MSG_TX));
xA429MsgTx.userHandle = 0;
xA429MsgTx.flags = XL_A429_MSG_FLAG_ON_REQUEST;
xA429MsgTx.label = 0x04;
xA429MsgTx.gap = 32;
xA429MsgTx.parity = XL_A429_MSG_PARITY_DEFAULT;
xA429MsgTx.data = 0xAABBCC;
xlStatus = xLA429Transmit(portHandle, accessMask, msgCnt, &msgCntSent, &xLA429MsgTx);
```

Example

Setup the hardware scheduler with one ARINC 429 message. This message is triggered every 100000 us (100 ms). The global parity setting is used and a gap time of 8 bit is configured.

```c
XL_A429_MSG_TX xLA429MsgTx;
unsigned int msgCnt = 1;
unsigned int msgCntSent = 0;
memset(&xLA429MsgTx, 0, sizeof(XL_A429_MSG_TX));
xA429MsgTx.userHandle = 0;
xA429MsgTx.cycleTime = 100000;
xA429MsgTx.flags = XL_A429_MSG_FLAG_CYCLIC;
xA429MsgTx.label = 0x04;
xA429MsgTx.gap = 64;
xA429MsgTx.parity = XL_A429_MSG_PARITY_DEFAULT;
xA429MsgTx.data = 0xAABBCC;
xlStatus = xLA429Transmit(portHandle, accessMask, msgCnt, &msgCntSent, &xLA429MsgTx);
```
Example

Transmit a burst of ARINC 429 messages. Messages are sent immediately without cyclic hardware scheduler. The global minimum gap time is used and the parity setting is odd for every single message (not used from global settings).

```c
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<msgCnt;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;
}
xlStatus = xlA429Transmit(portHandle, msgCnt, &msgCntSent, x1A429MsgTx);
```

15.3.3 xlA429Receive

**Syntax**

```c
XLstatus xlA429Receive (  
    XLPortHandle portHandle,  
    XLa429Event* pXlA429Event  
)
```

**Description**

Retrieves one event from the event queue. This operation is synchronous.

**Input parameters**

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

- **pXlA429Event**
  Pointer to the application allocated receive event buffer (see section XLa429Event on page 454).

**Return value**

Returns an error code (see section Error Codes on page 482).
Example

Read each message from the message queue

XLa429Event xla429Event;

x1Status = xla429Receive(portHandle, &xla429Event);

if (x1Status != XL_ERR_QUEUE_IS_EMPTY) {
    switch(x1A429Event.tag) {
        case XL_A429_EV_TAG_TX_OK:
            // do something with received message data
            break;

        case XL_A429_EV_TAG_TX_ERR:
            break;

        case XL_A429_EV_TAG_RX_OK:
            break;

        case XL_A429_EV_TAG_RX_ERR:
            break;

        case XL_A429_EV_TAG_BUS_STATISTIC:
            break;

        default:
            break;
    }
}

15.4 Structs

15.4.1 XL_A429_PARAMS

Syntax

```c
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:
  ```c
  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.

<table>
<thead>
<tr>
<th>P</th>
<th>SSM</th>
<th>Data</th>
<th>Data</th>
<th>SDI</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
</tr>
</tbody>
</table>

parity bit  
data field for parity calculation

<table>
<thead>
<tr>
<th>31 bits of data</th>
<th>count of 1 bits</th>
<th>Odd (parity bit)</th>
<th>Even (parity bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0000 0000 0000 0000 0000 0000 0000</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>000 0100 1000 0010 0010 1111 1000 0000</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>111 0000 1111 1111 1111 0000 1000 0000</td>
<td>16</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>111 1111 1111 1111 1111 1111 1111 1111</td>
<td>31</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**tx.minGap**

Specifies the global minimum gap time between two consecutive messages. The configured gap time is inserted before a message is transmitted. Minimum Gap between two messages is defined in 1/8 bit time steps. This value is limited to 2047 (a min gap time of 255 bit). At a bitrate of 100000 bit/s the bit time is equivalent to 10 us. A setting of 32 (4 bit gap time) corresponds to a minimum gap time of 40 us. It is also possible to overwrite the minGap settings for every ARINC word separately. This is done in the gap field of the structure XL_A429_MSG_TX.
**rx.bitrate**

Specifies the desired Rx channel bitrate. This value is recalculated by hardware interface for internal clock usage (64 MHz) with a guaranteed bitrate precision of +/- 15,625 ns. If `autoBaudrate` is disabled this value is needed for Rx channel settings, otherwise this value is ignored. Following value ranges is allowed for bitrate settings:

- `XL_A429_MSG_BITRATE_SLOW_MIN` (10000 bit/s)
- `XL_A429_MSG_BITRATE_FAST_MINFAST_MIN` (120000 bit/s)

**rx.minBitrate**

Specifies the minimum allowed bitrate for Rx channels. This value is recalculated by hardware interface for internal clock usage (64 MHz) with a guaranteed bitrate precision of +/- 15,625 ns. If measured value is below this value a bitrate error is reported. Minimum allowed bitrate is 10000 bit/s (`XL_A429_MSG_BITRATE_RX_MIN`). The bitrate error check is done for every bit.

**rx.maxBitrate**

Specifies the maximum allowed bitrate for Rx channels. This value is recalculated by hardware interface for internal clock usage (64 MHz) with a guaranteed bitrate precision of +/- 15,625 ns. If measured value is above this value a bitrate error is reported. Maximum allowed bitrate is 120000 bit/s (`XL_A429_MSG_BITRATE_RX_MAX`). The bitrate error check is done for every bit.
- **rx.parity**
  Global parity calculation for each Rx channel. There are three options available.

  `XL_A429_MSG_PARITY_DISABLED`
  Disables the hardware parity check. There is no parity error generated.

  `XL_A429_MSG_PARITY_ODD`
  Enables odd hardware parity check. If the parity check result is even, an error is generated.

  `XL_A429_MSG_PARITY_EVEN`
  Enables even hardware parity check. If the parity check result is odd, an error is generated.

- **rx.minGap**
  Specifies the global minimum gap time between consecutively received messages. Minimum Gap between two messages is defined in 1/8 bit time steps. This value is limited to 2047 (a min gap time of 255 bit). At a bitrate of 100000 bit/s the bit time is equivalent to 10 us. A setting of 32 (4 bit gap time) corresponds to a min gap time of 40 us. A gap error is reported if the measured gap time between two ARINC words is below this configured value.

- **rx.autoBaudrate**
  Enables or disables the automatic bitrate detection for Rx channels.

  `XL_A429_MSG_AUTO_BAUDRATE_DISABLED`
  Disables the automatic bitrate detection. The expected bitrate has to be set and a valid range for minimum and maximum bitrate has to be configured.

  `XL_A429_MSG_AUTO_BAUDRATE_ENABLED`
  For automatic bitrate detection the minimum and maximum bitrate has to be set. The Rx bitrate settings will be ignored. It is possible to use the complete range for minimum and maximum bitrate (XL_A429_MSG_BITRATE_RX_MIN ... XL_A429_MSG_BITRATE_RX_MIN). In this mode the “average bitrate error” and “duty factor” error situation are neither checked nor reported.

- **raw**
  Raw data of the data union.

- **rx.parity**
  Global parity calculation for each Rx channel. There are three options available.

  `XL_A429_MSG_PARITY_DISABLED`
  Disables the hardware parity check. There is no parity error generated.

  `XL_A429_MSG_PARITY_ODD`
  Enables odd hardware parity check. If the parity check result is even, an error is generated.

  `XL_A429_MSG_PARITY_EVEN`
  Enables even hardware parity check. If the parity check result is odd, an error is generated.
rx.minGap
Specifies the global minimum gap time between consecutively received messages. Minimum Gap between two messages is defined in 1/8 bit time steps. This value is limited to 2047 (a min gap time of 255 bit). At a bitrate of 100000 bit/s the bit time is equivalent to 10 us. A setting of 32 (4 bit gap time) corresponds to a min gap time of 40 us. A gap error is reported if the measured gap time between two ARINC words is below this configured value.

rx.autoBaudrate
Enables or disables the automatic bitrate detection for Rx channels.

XL_A429_MSG_AUTO_BAUDRATE_DISABLED
Disables the automatic bitrate detection. The expected bitrate has to be set and a valid range for minimum and maximum bitrate has to be configured.

XL_A429_MSG_AUTO_BAUDRATE_ENABLED
For automatic bitrate detection the minimum and maximum bitrate has to be set. The Rx bitrate settings will be ignored. It is possible to use the complete range for minimum and maximum bitrate (XL_A429_MSG_BITRATE_RX_MIN ... XL_A429_MSG_BITRATE_RX_MIN). In this mode the “average bitrate error” and “duty factor” error situation are neither checked nor reported.

raw
Raw data of the data union.

Note
Successful configured ARINC parameters can be retrieved by xlGetDriverConfig. Depending on bus type XLbusParams contains ARINC configured parameters. These values are the configured parameters of xIA429SetChannelParams() and not the measured/configured values of the hardware interface.

Example
Configures an A429 channel in Rx channel direction with enabled bitrate detection (expected bitrate should be between minimum bitrate and maximum bitrate), parity check is enabled (odd parity) and a default minimum gap setting of 4 bit is used.

XL_A429_PARAMS 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;
xA429Params.data.rx.maxBitrate = 102500;
xA429Params.data.rx.parity = XL_A429_MSG_PARITY_ODD;
xA429Params.data.rx.minGap = XL_A429_MSG_GAP_4BIT;

xlStatus = 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.

```c
XL_A429_PARAMS xLA429Params;
memset(&xLA429Params, 0, sizeof(XL_A429_PARAMS));

xLA429Params.channelDirection = XL_A429_MSG_CHANNEL_DIR_RX;

xLA429Params.data.rx.autoBaudrate = XL_A429_MSG_AUTO_BAUDRATE_DISABLED;

xLA429Params.data.rx.bitrate = 10500;
xLA429Params.data.rx.minBitrate = 10000;
xLA429Params.data.rx.maxBitrate = 11500;
xLA429Params.data.rx.parity = XL_A429_MSG_PARITY_ODD;
xLA429Params.data.rx.minGap = XL_A429_MSG_GAP_4BIT;

xlStatus = xLA429SetChannelParams(xlPortHandle, xlChannelMask, &xLA429Params);
```

15.4.2 XL_A429_MSG_TX

**Syntax**

```c
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 cycleTime 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.
15.5 Events

15.5.1 XLa429Event

Syntax

```c
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.

  ```c
  XL_A429_EV_TAG_TX_OK
  when a message was transmitted completely.
  ```

  ```c
  XL_A429_EV_TAG_TX_ERR
  when an error was detected by the transmitter.
  ```

  ```c
  XL_A429_EV_TAG_RX_OK
  when a message was received entirely.
  ```

  ```c
  XL_A429_EV_TAG_RX_ERR
  when an error is detected by the receiver.
  ```

  ```c
  XL_A429_EV_TAG_BUS_STATISTIC
  when a bus statistic is requested.
  ```

- **channelIndex**
  Contains the logical channel number.
15.5.2 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.

<table>
<thead>
<tr>
<th>P</th>
<th>SSM</th>
<th>Data</th>
<th>Data</th>
<th>SDI</th>
</tr>
</thead>
</table>

Bit position in data field

15.5.3 **XL_A429_EV_TAG_TX_ERR**

**Syntax**
```c
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.

15.5.4 XL_A429_EV_TAG_RX_OK

**Syntax**
```c
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.
15.5.5 XL_A429_EV_TAG_RX_ERR

Syntax

```c
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 (±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.

15.5.6 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.
15.6 Application Examples

15.6.1 xlA429Control

15.6.1.1 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 44: Running xlA429Control
16 .NET Wrapper

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>Overview</td>
<td>464</td>
</tr>
<tr>
<td>16.2</td>
<td>XLDriver - Accessing Driver</td>
<td>465</td>
</tr>
<tr>
<td>16.3</td>
<td>XLClass - Storing Data/Parameters</td>
<td>466</td>
</tr>
<tr>
<td>16.4</td>
<td>XLDefine - Using Predefined Values</td>
<td>467</td>
</tr>
<tr>
<td>16.5</td>
<td>Including the Wrapper in a New .NET Project</td>
<td>468</td>
</tr>
<tr>
<td>16.6</td>
<td>Application Examples</td>
<td>470</td>
</tr>
</tbody>
</table>
16.1 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: \ XL_OpenDriver()

Native XL API: \ xlOpenDriver()

The required parameters of the .NET methods can be looked up by using the IntelliSense feature of the IDE, for example:

```csharp
// Activate channel
status = VDemo.XL_ActivateChannel(portHandle, accessMask, busTypeCAN, XLDefine.XL_AC_FLAGS.XL Lê
Console.WriteLine("XLStatus.XLDriver.XL_ActivateChannel(channel portHandle, using 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 vxlapi.dll/vxlapi64.dll have also to be copied into the execution folder of your application.

---

**Note**

To run the XL API .NET Wrapper, framework .NET 3.5 or higher is required.
16.2 XLDriver - Accessing Driver

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.

```csharp
XLDriver myApp = new XLDriver();
myApp.XL_Op...
```

XLD does the XL_Status XLDriver.XL_OpenDriver() Opens the XL Driver.
16.3 XLClass - Storing Data/Parameters

Predefined classes

Some of the XL API .NET methods expect objects (parameters). For this case, all required classes are predefined in the class XLClass and ready to use. Most of these classes are clones of the XL API structures. Please refer to the general XL Driver Library documentation for further information.

Here are some examples of these predefined classes:

- **xl_driver_config**
  For storing the driver configuration.
  Required by method XL_GetDriverConfig().

- **xl_event**
  Contains data to be transmitted.
  Required by method XL_CanTransmit().

- **xl_event_collection**
  For storing one or more xl_events.
  Required by method XL_CanTransmit().

- **xl_bus_params**
  Used by subclass xl_channel_config.

- **xl_channel_config**
  Used by subclass xl_driver_config.

- **xl_can_message**
  Used by subclass xl_event.

- **xl_chip_params**
  Used by method XL_CanSetChannelParams().

- **xl_linStatPar**
  Used by method XL_LinSetChannelParams().

Note

More predefined classes in XLClass can be found via the IntelliSense feature in your IDE.
16.4 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_DII0
  .XL_BUS_TYPE_FLEXRAY
  ...

Note
More definitions can be found via the IntelliSense feature in your IDE. Example: XL_MessageFlags.
16.5 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.
16.6 Application Examples

16.6.1 xlCANdemo .NET

Description
This example shows how to access a Vector CAN interface.

Starting the example
When the example starts, it looks for the application xlCANdemoNET in 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.
16.6.2 xlCANdemo .NET

Description
This example shows how to access the XL API for CAN FD.

Starting the example
When the example starts, it looks for the application xlCANdemoNET in 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.
16.6.3 xlLINdemo .NET

Description This example shows how to access a Vector LIN interface.

Starting the example When the example starts, it looks for the application xlLINdemoNET in the Vector Hardware 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.
### 16.6.4 xILINdemo Single .NET

| Description | This example is similar to `xILINdemo .NET` on page 472, but uses only one LIN channel. |
16.6.5  xIDAIexample .NET

16.6.5.1 General Information

**Description**  
This example demonstrates how to access an IOcab 8444opto for cyclical measurement.

**Starting the example**  
When the example starts, it looks for the application xIDAIexampleNET 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.

**Setup**  
**Pin definition**  
The following pins of the IOcab 8444opto are used in this example:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO0</td>
<td>14</td>
<td>Analog output</td>
</tr>
</tbody>
</table>
### Signal, Pin, Type

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO1</td>
<td>7</td>
<td>Analog input</td>
</tr>
<tr>
<td>AIO2</td>
<td>15</td>
<td>Analog input</td>
</tr>
<tr>
<td>AIO3</td>
<td>8</td>
<td>Analog input</td>
</tr>
<tr>
<td>DIO0</td>
<td>1</td>
<td>Digital output (shared electronic switch with DIO1).</td>
</tr>
<tr>
<td>DIO1</td>
<td>9</td>
<td>Digital output (supplied by DIO0, when switch is closed).</td>
</tr>
<tr>
<td>DIO2</td>
<td>2</td>
<td>Digital input.</td>
</tr>
<tr>
<td>DIO3</td>
<td>10</td>
<td>Digital input.</td>
</tr>
</tbody>
</table>

### Setup

#### Note

The internal switch between DIO0 (supplied by AIO0) and DIO1 is closed/opened with `xlDAIOSetDigitalOutput()`. If the switch is closed, the applied voltage at DIO0 can be measured at DIO1.

### 16.6.5.3 Keyboard commands

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ENTER&gt;</td>
<td>Toggle digital output.</td>
</tr>
<tr>
<td>&lt;x&gt;</td>
<td>Closes application.</td>
</tr>
</tbody>
</table>
16.6.5.4 Output Examples

Example

| AI00:  | 4032mV |
| AI01:  | 0mV    |
| AI02:  | 0mV    |
| AI03:  | 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

- "AI00" displays 4032mV, since it is set to output with maximum output level.
- "AI01" displays 0mV, since there is no applied voltage at this input.
- "AI02" displays 0mV, since there is no applied voltage at this input.
- "AI03" displays 0mV, since there is no applied voltage at this input.
- "Switch selected" displays DIO0/DIO1 (first switch)
- "Switch states" displays the state of switch between DIO0/DIO1
- "Digital Port" shows the single states of DIO7…DIO0:
  - DIO0: displays ‘1’ (always ‘1’, due the voltage supply)
  - DIO1: displays ‘0’ (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays ‘0’ (output of DIO1)
  - DIO3: displays ‘0’ (output of DIO1)
  - DIO4: displays ‘0’ (n.c.)
  - DIO5: displays ‘0’ (n.c.)
  - DIO6: displays ‘0’ (n.c.)
  - DIO7: displays ‘0’ (n.c.)
**Example**

<table>
<thead>
<tr>
<th>AIO0</th>
<th>AIO1</th>
<th>AIO2</th>
<th>AIO3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4032mV</td>
<td>0mV</td>
<td>4032mV</td>
<td>0mV</td>
</tr>
</tbody>
</table>

Switch selected: DIO0/DIO1
Switch states: CLOSED
Digital Port: | DIO7 | DIO6 | DIO5 | DIO4 | DIO3 | DIO2 | DIO1 | DIO0 | val |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation**

- "AIO0" displays 4032mV, since it is set to output with maximum output level.
- "AIO1" displays 0mV, since there is no applied voltage at this input.
- "AIO0" displays 4032mV, since it is connected to AIO0.
- "AIO3" displays 0mV, since there is no applied voltage at this input.
- "Switch selected" displays DIO0/DIO1 (first switch)
- "Switch state" displays the state of switch between DIO0/DIO1
- "Digital Port" shows the single states of DIO7…DIO0:
  - DIO0: displays ‘1’ (always ‘1’, due the voltage supply)
  - DIO1: displays ‘1’ (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays ‘1’ (output of DIO1)
  - DIO3: displays ‘1’ (output of DIO1)
  - DIO4: displays ‘0’ (n.c.)
  - DIO5: displays ‘0’ (n.c.)
  - DIO6: displays ‘0’ (n.c.)
  - DIO7: displays ‘0’ (n.c.)

**Note**

If you try to connect DIO1 (when output is ‘1’) to one of the inputs DIO4…DIO7, you will notice no changes on the screen. The digital output is supplied by the IOcab 8444opto itself, where the maximum output is 4.096V. Due to different thresholds, the inputs DIO4…DIO7 needs higher voltages (>=4.7V) to toggle from ‘0’ to ‘1’.
16.6.6 xIIOpiggyExample .NET

16.6.6.1 General Information

Description

This example shows how to access the IOpiggy 8642 for analog measurements.

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).

16.6.6.2 Setup

Pin definition

The following pins of the IOpiggy 8642 are used in this example:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO0</td>
<td>14</td>
<td>Analog output</td>
</tr>
<tr>
<td>AIO1</td>
<td>7</td>
<td>Analog input</td>
</tr>
<tr>
<td>AIO2</td>
<td>15</td>
<td>Analog input</td>
</tr>
<tr>
<td>AIO3</td>
<td>8</td>
<td>Analog input</td>
</tr>
</tbody>
</table>
### Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO0</td>
<td>1</td>
<td>Digital output</td>
</tr>
<tr>
<td>DIO1</td>
<td>9</td>
<td>Digital input</td>
</tr>
<tr>
<td>DIO2</td>
<td>2</td>
<td>Digital input</td>
</tr>
<tr>
<td>DIO3</td>
<td>10</td>
<td>Digital input</td>
</tr>
<tr>
<td>DIO4</td>
<td>3</td>
<td>MOS switch</td>
</tr>
<tr>
<td>DIO5</td>
<td>11</td>
<td>MOS switch</td>
</tr>
</tbody>
</table>

### Analog measurement

By pressing the [ENTER] key, the example toggles the analog output level at AIO0 which can be measured at AIO1…A3. The output level at AIO0 cannot be read back at the same time and remains at 0 mV.

### Digital measurement

By pressing the [ENTER] key once, DIO0 outputs 5V. Afterwards, pressing [ENTER] toggles the switch between DIO4 and DIO5.
16.6.7 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 xlEthernetDemo.NET 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.
16.6.8 xlFRdemo .NET

Description
This example shows how to access FlexRay interface (e.g., VN7610) for COLD CC.

Starting the example
When the example starts, it looks for the application xlFRdemoNET in 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.
# 17 Error Codes

In this chapter you find the following information:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1 Common Error Codes</td>
<td>483</td>
</tr>
<tr>
<td>17.2 CAN FD Error Codes</td>
<td>485</td>
</tr>
<tr>
<td>17.3 FlexRay Error Codes</td>
<td>486</td>
</tr>
<tr>
<td>17.4 Ethernet Error Codes</td>
<td>487</td>
</tr>
<tr>
<td>17.5 MOST150 Error Codes</td>
<td>488</td>
</tr>
</tbody>
</table>
## 17.1 Common Error Codes

<table>
<thead>
<tr>
<th>Code (dec)</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>XL_SUCCESS</td>
<td>The driver call was successful.</td>
</tr>
<tr>
<td>1</td>
<td>XL_PENDING</td>
<td>The driver call returned but the requested operation is still pending.</td>
</tr>
<tr>
<td>10</td>
<td>XL_ERR_QUEUE_IS_EMPTY</td>
<td>The receive queue of the port is empty. The user can proceed normally.</td>
</tr>
<tr>
<td>11</td>
<td>XL_ERR_QUEUE_IS_FULL</td>
<td>The transmit queue of a channel is full. The transmit event will be lost.</td>
</tr>
<tr>
<td>12</td>
<td>XL_ERR_TX_NOT_POSSIBLE</td>
<td>The hardware is busy and not able to transmit an event at once.</td>
</tr>
<tr>
<td>14</td>
<td>XL_ERR_NO_LICENSE</td>
<td>The user requested an operation that requires a license to unlock (see section License Management on page 36).</td>
</tr>
<tr>
<td>101</td>
<td>XL_ERR_WRONG_PARAMETER</td>
<td>At least one parameter passed to the driver was wrong or invalid.</td>
</tr>
<tr>
<td>110</td>
<td>XL_ERR_TWICE_REGISTER</td>
<td>The user attempted to register something that is already registered or otherwise in use.</td>
</tr>
<tr>
<td>111</td>
<td>XL_ERR_INVALID_CHAN_INDEX</td>
<td>The driver attempted to access a channel with an invalid index.</td>
</tr>
<tr>
<td>112</td>
<td>XL_ERR_INVALID_ACCESS</td>
<td>The user made a call to a port specifying channel(s) for which he had not declared access at opening of the port.</td>
</tr>
<tr>
<td>113</td>
<td>XL_ERR_PORT_IS_OFFLINE</td>
<td>The user called a port function whose execution must be online, but the port is offline.</td>
</tr>
<tr>
<td>116</td>
<td>XL_ERR_CHAN_IS_ONLINE</td>
<td>The user called a function whose desired channels must be offline, but at least one channel is online.</td>
</tr>
<tr>
<td>117</td>
<td>XL_ERR_NOT_IMPLEMENTED</td>
<td>The user called a feature which is not implemented.</td>
</tr>
<tr>
<td>118</td>
<td>XL_ERR_INVALID_PORT</td>
<td>The driver attempted to access a port by an invalid pointer or index.</td>
</tr>
<tr>
<td>120</td>
<td>XL_ERR_HW_NOT_READY</td>
<td>The accessed hardware is not ready.</td>
</tr>
<tr>
<td>121</td>
<td>XL_ERR_CMD_TIMEOUT</td>
<td>The timeout condition occurred while waiting for the response event of a command.</td>
</tr>
<tr>
<td>122</td>
<td>XL_ERR_CMD_HANDLING</td>
<td>An internal error concerning command execution occurred.</td>
</tr>
<tr>
<td>129</td>
<td>XL_ERR_HW_NOT_PRESENT</td>
<td>The hardware is not present (or could not be found) at a channel. This may occur with removable hardware or faulty hardware.</td>
</tr>
<tr>
<td>131</td>
<td>XL_ERR_NOTIFY_ALREADY_ACTIVE</td>
<td>This error code exists for historic reasons.</td>
</tr>
<tr>
<td>132</td>
<td>XL_ERR_INVALID_TAG</td>
<td>The driver refuses a tag in an event structure.</td>
</tr>
<tr>
<td>133</td>
<td>XL_ERR_INVALID_RESERVED_FLD</td>
<td>The user passed a structure that has reserved bits set to one.</td>
</tr>
<tr>
<td>134</td>
<td>XL_ERR_INVALID_SIZE</td>
<td>The driver refuses a size value in an event structure.</td>
</tr>
<tr>
<td>135</td>
<td>XL_ERR_INSUFFICIENT_BUFFER</td>
<td>The user passed a buffer that is too small to receive all data.</td>
</tr>
<tr>
<td>136</td>
<td>XL_ERR_ERROR_CRC</td>
<td>A checksum calculation failed.</td>
</tr>
<tr>
<td>137</td>
<td>XL_ERR_BAD_EXE_FORMAT</td>
<td>The OS failed to load an executable, e.g. a DLL is compiled for a different architecture.</td>
</tr>
<tr>
<td>Code (dec)</td>
<td>Error</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>138</td>
<td>XL_ERR_NO_SYSTEM_RESOURCES</td>
<td>The OS has insufficient resources (e.g. memory) to perform the requested operation.</td>
</tr>
<tr>
<td>139</td>
<td>XL_ERR_NOT_FOUND</td>
<td>The driver could not find the specified object.</td>
</tr>
<tr>
<td>140</td>
<td>XL_ERR_INVALID_ACCESS</td>
<td>The driver does not permit the specified access, e.g. wrong index, wrong interface version or insufficient privileges.</td>
</tr>
<tr>
<td>141</td>
<td>XL_ERR_REQ_NOT_ACCEPT</td>
<td>The driver refused the operation for an unspecified reason.</td>
</tr>
<tr>
<td>142</td>
<td>XL_ERR_INVALID_LEVEL</td>
<td>An internal error concerning queue levels occurred.</td>
</tr>
<tr>
<td>143</td>
<td>XL_ERR_NO_DATADETected</td>
<td>The driver expects data that is not available.</td>
</tr>
<tr>
<td>144</td>
<td>XL_ERR_INTERNAL_ERROR</td>
<td>An unspecified internal error occurred.</td>
</tr>
<tr>
<td>145</td>
<td>XL_ERR_UNEXP_NET_ERROR</td>
<td>An unexpected networking error occurred.</td>
</tr>
<tr>
<td>146</td>
<td>XL_ERR_INVALID_USER_BUFFER</td>
<td>The driver refused a buffer provided by user-space.</td>
</tr>
<tr>
<td>147</td>
<td>XL_ERR_INVALID_PORT_ACCESS_TYPE</td>
<td>See XL_ERR_INVALID_ACCESS.</td>
</tr>
<tr>
<td>152</td>
<td>XL_ERR_NO_RESOURCES</td>
<td>The driver has insufficient resources (e.g. memory) to perform the requested operation.</td>
</tr>
<tr>
<td>153</td>
<td>XL_ERR_WRONG_CHIP_TYPE</td>
<td>The relevant chip on the device does not support the requested operation.</td>
</tr>
<tr>
<td>154</td>
<td>XL_ERR_WRONG_COMMAND</td>
<td>The user issued a command that was rejected.</td>
</tr>
<tr>
<td>155</td>
<td>XL_ERR_INVALID_HANDLE</td>
<td>The user passed an invalid handle.</td>
</tr>
<tr>
<td>157</td>
<td>XL_ERR_RESERVED_NOT_ZERO</td>
<td>See XL_ERR_INVALID_RESERVED_FLD.</td>
</tr>
<tr>
<td>158</td>
<td>XL_ERR_INIT_ACCESS_MISSING</td>
<td>Function call requires init access.</td>
</tr>
<tr>
<td>160</td>
<td>XL_ERR_WRONG_VERSION</td>
<td>A version mismatch was detected.</td>
</tr>
<tr>
<td>201</td>
<td>XL_ERR_CANNOT_OPEN_DRIVER</td>
<td>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.</td>
</tr>
<tr>
<td>202</td>
<td>XL_ERR_WRONG_BUS_TYPE</td>
<td>The user called a function with the wrong bus type. (e.g. try to activate a LIN channel for CAN).</td>
</tr>
<tr>
<td>203</td>
<td>XL_ERR_DLL_NOT_FOUND</td>
<td>The XL API dll could not be found.</td>
</tr>
<tr>
<td>204</td>
<td>XL_ERR_INVALID_CHANNEL_MASK</td>
<td>Invalid channel mask.</td>
</tr>
<tr>
<td>205</td>
<td>XL_ERR_NOT_SUPPORTED</td>
<td>Function call not supported.</td>
</tr>
<tr>
<td>210</td>
<td>XL_ERR_CONNECTION_BROKEN</td>
<td>The driver lost connection to a remote device.</td>
</tr>
<tr>
<td>211</td>
<td>XL_ERR_CONNECTION_CLOSED</td>
<td>The connection is already closed.</td>
</tr>
<tr>
<td>212</td>
<td>XL_ERR_INVALID_STREAM_NAME</td>
<td>An internal error concerning remote devices occurred.</td>
</tr>
<tr>
<td>213</td>
<td>XL_ERR_CONNECTION_FAILED</td>
<td>An internal error concerning remote devices occurred.</td>
</tr>
<tr>
<td>214</td>
<td>XL_ERR_STREAM_NOT_FOUND</td>
<td>An internal error concerning remote devices occurred.</td>
</tr>
<tr>
<td>215</td>
<td>XL_ERR_STREAM_NOT_CONNECTED</td>
<td>An internal error concerning remote devices occurred.</td>
</tr>
<tr>
<td>216</td>
<td>XL_ERR_QUEUE_OVERRUN</td>
<td>A queue overrun occurred.</td>
</tr>
<tr>
<td>255</td>
<td>XL_ERROR</td>
<td>An unspecified error occurred.</td>
</tr>
</tbody>
</table>
## 17.2 CAN FD Error Codes

<table>
<thead>
<tr>
<th>Code (hex)</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0201</td>
<td>XL_ERR_INVALID_DLC</td>
<td>DLC with invalid value.</td>
</tr>
<tr>
<td>0x0202</td>
<td>XL_ERR_INVALID_CANID</td>
<td>CAN Id has invalid bits set.</td>
</tr>
<tr>
<td>0x0203</td>
<td>XL_ERR_INVALID_FDFLAG_MODE20</td>
<td>Flag set that must not be set when configured for CAN20 (e.g. EDL).</td>
</tr>
<tr>
<td>0x0204</td>
<td>XL_ERR_EDL_RTR</td>
<td>RTR must not be set in combination with EDL.</td>
</tr>
<tr>
<td>0x0205</td>
<td>XL_ERR_EDL_NOT_SET</td>
<td>EDL is not set but BRS and/or ESICTRL is.</td>
</tr>
<tr>
<td>0x0206</td>
<td>XL_ERRUNKNOWN_FLAG</td>
<td>Unknown bit in flags field is set.</td>
</tr>
</tbody>
</table>
## 17.3 FlexRay Error Codes

<table>
<thead>
<tr>
<th>Code (hex)</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0104</td>
<td>XL_ERR_PDU_OUT_OF_MEMORY</td>
<td>Too many PDUs configured or too less system memory free.</td>
</tr>
<tr>
<td>0x0105</td>
<td>XL_ERR_FR_CLUSTERCONFIG_MISSING</td>
<td>No cluster configuration has been sent to the driver but is needed for the command which failed.</td>
</tr>
<tr>
<td>0x0106</td>
<td>XL_ERR_PDU_OFFSETREPET_INVALID</td>
<td>Invalid offset and/or repetition value specified.</td>
</tr>
<tr>
<td>0x0107</td>
<td>XL_ERR_PDU_PAYLOAD_SIZE_INVALID</td>
<td>Specified PDU payload size is invalid (e.g. size is too large) Frame-API: size is different than static payload length configured in cluster config.</td>
</tr>
<tr>
<td>0x0109</td>
<td>XL_ERR_FR_NBR_FRAMES_OVERFLOW</td>
<td>Too many frames specified in parameter.</td>
</tr>
<tr>
<td>0x010B</td>
<td>XL_ERR_FR_SLOT_ID_INVALID</td>
<td>Specified slot-ID exceeds biggest possible ID specified by the cluster configuration.</td>
</tr>
<tr>
<td>0x010C</td>
<td>XL_ERR_FR_SLOT_ALREADY_OCCUPIED_BY_ERAY</td>
<td>Specified slot cannot be used by Coldstart-Controller because it's already in use by the eRay.</td>
</tr>
<tr>
<td>0x010D</td>
<td>XL_ERR_FR_SLOT_ALREADY_OCCUPIED_BY_COLD</td>
<td>Specified slot cannot be used by eRay because it's already in use by the Coldstart-Controller.</td>
</tr>
<tr>
<td>0x010E</td>
<td>XL_ERR_FR_SLOT_OCCUPIED_BY_OTHER_APP</td>
<td>Specified slot cannot be used because it's already in use by another application.</td>
</tr>
<tr>
<td>0x010F</td>
<td>XL_ERR_FR_SLOT_IN_WRONG_SEGMENT</td>
<td>Specified slot is not in correct segment. E.g.: A dynamic slot was specified for startup&amp;sync.</td>
</tr>
<tr>
<td>0x0110</td>
<td>XL_ERR_FR_FRAME_CYCLEMULTIPLEX_ERROR</td>
<td>The given frame-multiplexing rule (specified by offset and repetition) cannot be done because some of the slots are already in use.</td>
</tr>
<tr>
<td>0x0116</td>
<td>XL_ERR_PDU_NO_UNMAP_OF_SYNCFRAME</td>
<td>Unmapping of eRay startup/sync frames is not allowed.</td>
</tr>
<tr>
<td>0x0123</td>
<td>XL_ERR_SYNC FRAME_MODE</td>
<td>Wrong txMode in sync frame.</td>
</tr>
</tbody>
</table>
## 17.4 Ethernet Error Codes

<table>
<thead>
<tr>
<th>Code (hex)</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1100</td>
<td>XL_ERR_ETH_PHY_ACTIVATION_FAILED</td>
<td>The driver failed to activate an Ethernet PHY.</td>
</tr>
<tr>
<td>0x1103</td>
<td>XL_ERR_ETH_PHY_CONFIG_ABORTED</td>
<td>The driver aborted the configuration of an Ethernet PHY.</td>
</tr>
<tr>
<td>0x1104</td>
<td>XL_ERR_ETH_RESET_FAILED</td>
<td>The driver failed to reset the device.</td>
</tr>
<tr>
<td>0x1105</td>
<td>XL_ERR_ETH_SET_CONFIG_DELAYED</td>
<td>Requested config was stored but could not be immediately activated.</td>
</tr>
<tr>
<td>0x1106</td>
<td>XL_ERR_ETH_UNSUPPORTED_FEATURE</td>
<td>Requested feature/function not supported by device.</td>
</tr>
<tr>
<td>0x1107</td>
<td>XL_ERR_ETH_MAC_ACTIVATION_FAILED</td>
<td>The driver failed to activate the Ethernet MAC.</td>
</tr>
<tr>
<td>0x110C</td>
<td>XL_ERR_NET_ETH_SWITCH_IS_ONLINE</td>
<td>Switch has already been activated.</td>
</tr>
</tbody>
</table>
## 17.5 MOST150 Error Codes

<table>
<thead>
<tr>
<th>Code  (dec)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4224</td>
<td>Invalid parameter <code>deviceMode</code> set in <code>xlMost150SetDeviceMode()</code></td>
</tr>
<tr>
<td>4225</td>
<td>Invalid parameter <code>nodeAddress</code> set in <code>xlMost150SetSpecialNodeInfo()</code></td>
</tr>
<tr>
<td>4226</td>
<td>Invalid parameter <code>groupAddress</code> set in <code>xlMost150SetSpecialNodeInfo()</code></td>
</tr>
<tr>
<td>4227</td>
<td>Invalid parameter <code>sbc</code> set in <code>xlMost150SetSpecialNodeInfo()</code></td>
</tr>
<tr>
<td>4228</td>
<td>Invalid parameter <code>CtrlRetryTime</code> or <code>ctrlSendAttempts</code> or async <code>RetryTime</code> or <code>asyncSendAttempts</code> set in <code>xlMost150SetSpecialNodeInfo()</code></td>
</tr>
<tr>
<td>4234</td>
<td>Invalid parameter <code>device</code> set in <code>xlMost150CtrlSyncAudio()</code>, <code>xlMost150SyncSetXXX()</code> or <code>xlMost150SyncGetXXX()</code></td>
</tr>
<tr>
<td>4235</td>
<td>Invalid parameter <code>node</code> set in <code>xlMost150CtrlSyncAudio()</code></td>
</tr>
<tr>
<td>4236</td>
<td>Invalid parameter <code>width</code> set in <code>xlMost150CtrlSyncAudio()</code></td>
</tr>
<tr>
<td>4237</td>
<td>Invalid parameter <code>volume</code> set in <code>xlMost150SyncSetVolume()</code></td>
</tr>
<tr>
<td>4238</td>
<td>Invalid parameter <code>mute</code> set in <code>xlMost150SyncSetMute()</code></td>
</tr>
<tr>
<td>4239</td>
<td>Invalid parameter <code>mode</code> set in <code>xlMost150CtrlSyncAudio()</code></td>
</tr>
<tr>
<td>4240</td>
<td>Invalid parameter <code>sourceMask</code> set in <code>xlMost150SwitchEventSources()</code></td>
</tr>
<tr>
<td>4242</td>
<td>Invalid parameter <code>attenuation</code> set in <code>xlMost150SetTxLightPower()</code></td>
</tr>
<tr>
<td>4243</td>
<td>Invalid parameter <code>txLightset</code> set in <code>xlMost150SetTxLight()</code></td>
</tr>
<tr>
<td>4244</td>
<td>Invalid parameter <code>requestMask</code> set in <code>xlMost150GetSpecialNodeInfo()</code></td>
</tr>
<tr>
<td>4245</td>
<td>Invalid parameter <code>frequency</code> set in <code>xlMost150SetFrequency()</code></td>
</tr>
<tr>
<td>4246</td>
<td>Invalid parameter <code>targetAddress</code> set in <code>xlMost150CtrlConfigureBusload()</code> or <code>xlMost150AsyncConfigureBusload()</code></td>
</tr>
<tr>
<td>4247</td>
<td>Invalid parameter <code>telLen</code> or <code>length</code> set in <code>xlMost150CtrlConfigureBusload()</code> or <code>xlMost150AsyncConfigureBusload()</code></td>
</tr>
<tr>
<td>4248</td>
<td>Invalid parameter <code>counterType</code> set in <code>xlMost150CtrlConfigureBusload()</code> or <code>xlMost150AsyncConfigureBusload()</code></td>
</tr>
<tr>
<td>4249</td>
<td>Invalid parameter <code>counterPosition</code> set in <code>xlMost150CtrlConfigureBusload()</code> or <code>xlMost150AsyncConfigureBusload()</code></td>
</tr>
<tr>
<td>4250</td>
<td>Invalid parameter <code>telLen</code> set in <code>xlMost150CtrlTransmit()</code></td>
</tr>
<tr>
<td>4251</td>
<td>Invalid parameter <code>length</code> set in <code>xlMost150AsyncTransmit()</code></td>
</tr>
<tr>
<td>4252</td>
<td>Invalid parameter <code>length</code> set in <code>xlMost150EthernetTransmit()</code></td>
</tr>
<tr>
<td>4253</td>
<td>Invalid parameter <code>busloadType</code> set in <code>xlMost150AsyncConfigureBusload()</code></td>
</tr>
<tr>
<td>4254</td>
<td>Invalid parameter <code>numBytesPerFrame</code> set in <code>xlMost150StreamOpen()</code></td>
</tr>
<tr>
<td>4255</td>
<td>Invalid parameter <code>latency</code> set in <code>xlMost150StreamOpen()</code></td>
</tr>
<tr>
<td>4256</td>
<td>Invalid parameter <code>direction</code> set in <code>xlMost150StreamOpen()</code></td>
</tr>
<tr>
<td>4257</td>
<td>Invalid parameter <code>streamHandle</code> set in <code>xlMost150StreamXXX()</code></td>
</tr>
<tr>
<td>4258</td>
<td>Invalid parameter <code>pConnLabels</code> set in <code>xlMost150StreamStart</code> (invalid CL)</td>
</tr>
<tr>
<td>4259</td>
<td>Invalid parameter <code>pConnLabels</code> set in <code>xlMost150StreamStart</code> (no CL provided)</td>
</tr>
<tr>
<td>4260</td>
<td>Invalid parameter <code>pConnLabels</code> set in <code>xlMost150StreamStart</code> (duplicate CL)</td>
</tr>
<tr>
<td>Code (dec)</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>4261</td>
<td>Invalid stream state. Rx or Tx stream state does not allow the call of xlMost150StreamXXX.</td>
</tr>
<tr>
<td>4262</td>
<td>Rx stream FIFO not initialized. This error can occur when calling xlMost150StreamStart without calling xlMost150StreamInitRxFifo before.</td>
</tr>
<tr>
<td>4263</td>
<td>Invalid parameter bypassCloseTime or bypassOpenTime set in xlMost150GenerateBypassStress.</td>
</tr>
<tr>
<td>4264</td>
<td>Invalid parameter numStates or pEclStates or pEclStatesDuration set in xlMost150ECLConfigureSeq.</td>
</tr>
<tr>
<td>4265</td>
<td>ECL sequence contains too many entries set in xlMost150ECLConfigureSeq.</td>
</tr>
</tbody>
</table>