Manual

XL Driver Library

API Description

Version 9.0

English

Imprint

Vector Informatik GmbH
Ingersheimer Straße 24
D-70499 Stuttgart

Contents

1 Introduction

In this chapter you find the following information:

1.1 About this User Manual

Conventions

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

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

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

1.1.1 Certification

Certified Quality Management System

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

1.1.2 Warranty

Restriction of warranty

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

1.1.3 Registered Trademarks

Registered trademarks

All trademarks mentioned in this user manual 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 user manual, fails, should not mean that a name is free of third party rights.

➔ Windows, Windows XP, Windows Vista, Windows 7, Windows 8 are trademarks of the Microsoft Corporation.

2 XL Driver Library Overview

In this chapter you find the following information:

2.1 General Information

Supported hardware

This document describes the API for the **XL Driver Library**. The library enables the development of own applications for CAN, LIN, MOST, FlexRay or digital/analog I/O based on Vector Network Interfaces (e.g. CANcardXLe, VN16xx, VN26xx, VN3xxx, VN76xx, VN89xx, …).

Info: The library does not support CANAC2 PCI, CANAC2 ISA and CANpari.

XL Driver Library

The library is available for several XL interfaces including the corresponding drivers for following operating systems:

- Windows XP (32 bit)
- Windows Vista (32 bit)
- Windows 7 (32 bit / 64 bit)
- Windows 8 (32 bit / 64 bit)

Furthermore, it is possible to build applications that run on different hardware and operation systems without any code changes. Hardware related settings can be configured in the Vector Hardware Configuration tool. It is possible to read those settings during execution.

The **XL Driver Library** can be linked with your application which grants access to a CANcab/piggy, LINcab/piggy, IOcab or to MOST. The library contains also a couple of examples (including the source code) which show the handling of the different functions for initialization, transmitting and receiving of messages.

Figure 1 depicts a basic overview of the construction of library application.

Applications overview

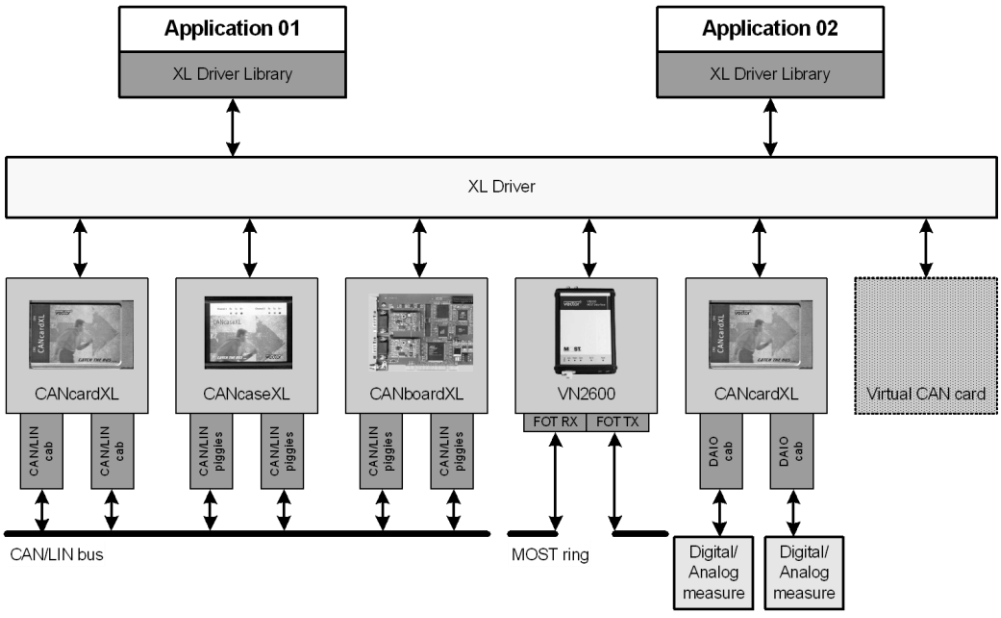


Figure 1: Possible applications with the XL Driver.

Hardware installation

Please refer to the user manual of your hardware for detailed information about the hardware installation.

2.2 Features

Multi hardware

The API is hardware independent and supports various Vector XL and VN interfaces. The bus type depends on the interface and the used Cabs or Piggybacks. Please refer to the user manual of the corresponding hardware for additional information or to the accessories manual on the Vector Driver Disk.

Multi application

The driver is designed for multi-processing (multi-tasking) operating systems, i.e. multiple applications can use the same channel of a CAN hardware at the same time (see Figure 2).

Info: If a Vector XL or VN interface is used for LIN, MOST or DAIO, a channel can only be used by one application at the same time.

Principle structure for CAN applications

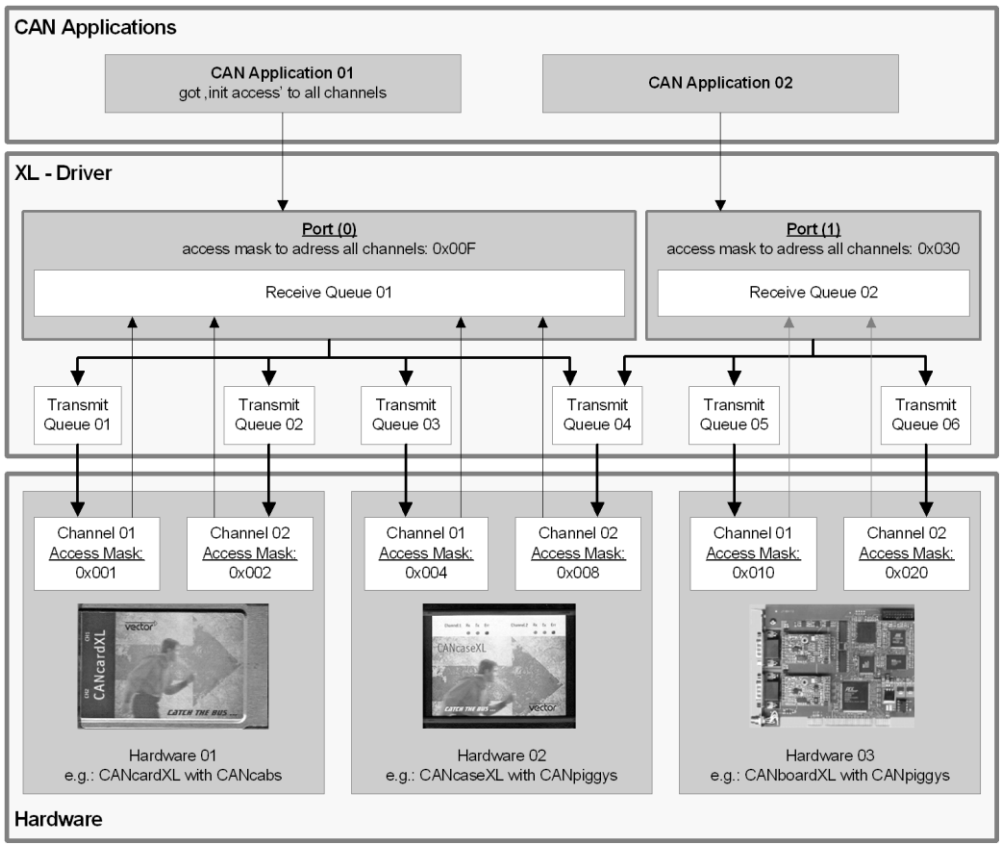


Figure 2: Accessing XL interfaces.

CAN

The library is designed to run multiple CAN applications using the same hardware concurrently by enveloping the hardware interfaces. The sequential calling convention is shown on page 15.

LIN

The LIN implementation supports no multi-application functionality like for CAN, i.e. only one application can access a channel (must have **init access,** see `xlOpenPort`). The sequential calling convention is shown on page 16.

MOST

The MOST implementation currently supports no multi-application functionality. It is also required that an application has **init access** (see `xlOpenPort`). The API description is available in the separate document

`XL Driver Library - MOST API Description.pdf`

which can be found in the **doc** folder of the XL Driver Library.

FlexRay

The API description is available in the separate document

`XL Driver Library - FlexRay API Description.pdf`

which can be found in the **doc** folder of the XL Driver Library. The implementation supports multi-application functionality. For further information see section Features on page 11, “Multi application”.

DAIO

The DAIO implementation supports limited multi-application functionality, i.e. only the first application (the one with granted **init access**, see `xlOpenPort`) can change DAIO parameters. All other applications can receive measured messages only, if the IOcab is configured for measurement by the first application. Please refer to the IOcab documentation for more details about measurement and input/output configuration. The sequential calling convention is shown on page 18.

General use of the XL Driver Library

In order to get driver access, the application must open a driver port and retrieve a port handle. This port handle is used for all subsequent calls to the driver. If a second application is demanding driver access, it gets the handle to another port. An application can open multiple ports.

Transmitting and receiving messages

In order to transmit a message, the application has to choose one or more physical channels which are connected to the port. The application calls the driver afterwards. Bit masks identify the channels (here it is called **access mask** or **channel mask**). The message is passed to every selected channel and is transmitted when possible.

If a hardware channel receives a message, it passes the message to every port that is using this channel. Each port maintains its own receive queue. The application at this port can poll the queue to determine whether there are incoming messages. See Figure 2 for an overview.

E.g. in C/C++

A thread reads out the driver message queue after an event was notified by a `WaitForSingleObject`.

Consequently, an application may demand initialization access for a channel. A channel only allows one port to have this access. For a LIN port it is needed to have **init access (**see `xlOpenPort`).

C/C++ access

The applications can get driver access by using a Windows DLL and a C header file.

.NET Access

A .NET wrapper is provided for .NET 2.0 or later in order to use the XL API in any .NET language. See the separate documentation

`XL Driver Library - .NET Wrapper Description.pdf`

for detailed information.

Files

File name	Description
vxlapi.dll	32 bit DLL for Windows XP/Vista/7
vxlapi64.dll	64 bit DLL for Windows 7
vxlapi.h	C header

Files

File name	Description
vxlapi_NET.dll	.NET wrapper. Supports 32 bit and 64 bit version of the vxlapi.dll.
vxlapi_NET.xml	Wrapper documentation, used by IntelliSense function

Dynamically loading of the XL Driver Library

If you want to load the `vxlapi.dll` dynamically, please insert `xlLoadlib.cpp` into your project. (This module is used within the **xlCANcontrol** demo program). The `vxlapi.h` supports loading of `vxlapi.dll` dynamically. It is only needed to set the `DYNAMIC_XLDRIVER_DLL` define. It is not necessary to change your source code, since `xlOpenDriver()` loads the dll and `xlCloseDriver()` unloads it.

DllMain

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

Debug prints

The library includes debug prints for developing. To switch on the **XL Library debug prints**, use the **Vector Hardware Configuration** tool. Go to the section **General information | Settings** and open the **Configuration flag**s dialog. There you can enter the debug flags:

`flags = 0x400000` for the **XL Library**.
`flags = 0x2000` (basic) and `0x4000` (advanced) for **MOST**.
`flags = 0x010000` (basic) and `0x020000` (advanced) for **FlexRay**.

To activate the flags it is needed to restart the driver and the entire application. To view the debug prints, the freeware tool **DebugView** from http://www.sysinternals.com (now Microsoft) can be used.

Vector Hardware Config

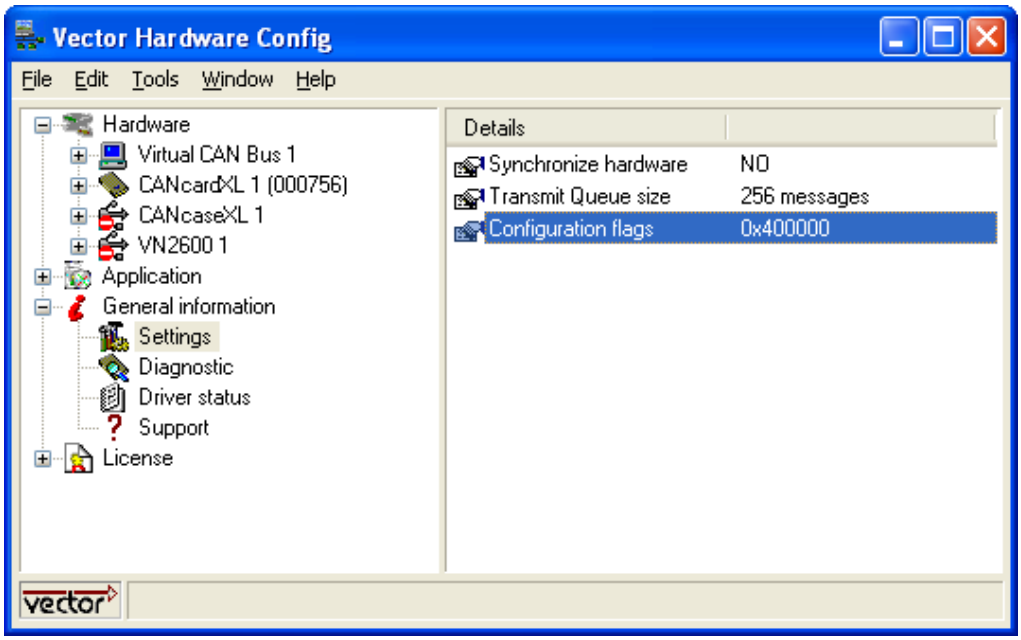
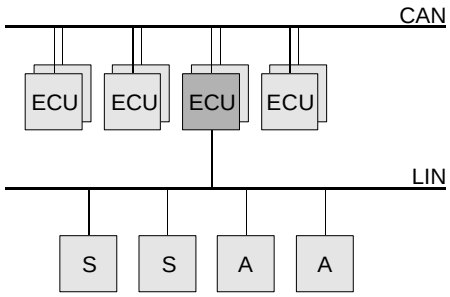


Figure 3: Hardware configuration

2.3 LIN Basics

Advantages of LIN

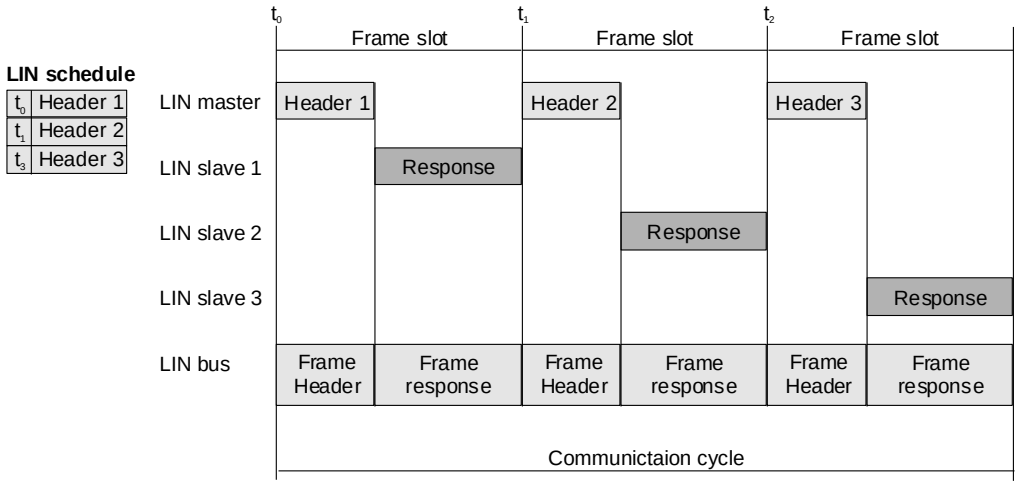
LIN (Local Interconnect Network) is a cheap way to connect many sensors and actuators to an ECU via one common communication medium (bus). This diminishes complexity as well as costs, weight and space problems and in addition it offers the possibility of diagnostics. Furthermore, LIN offers a high flexibility 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 message

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.

2.4 Flowcharts

2.4.1 CAN Application

Calling sequence

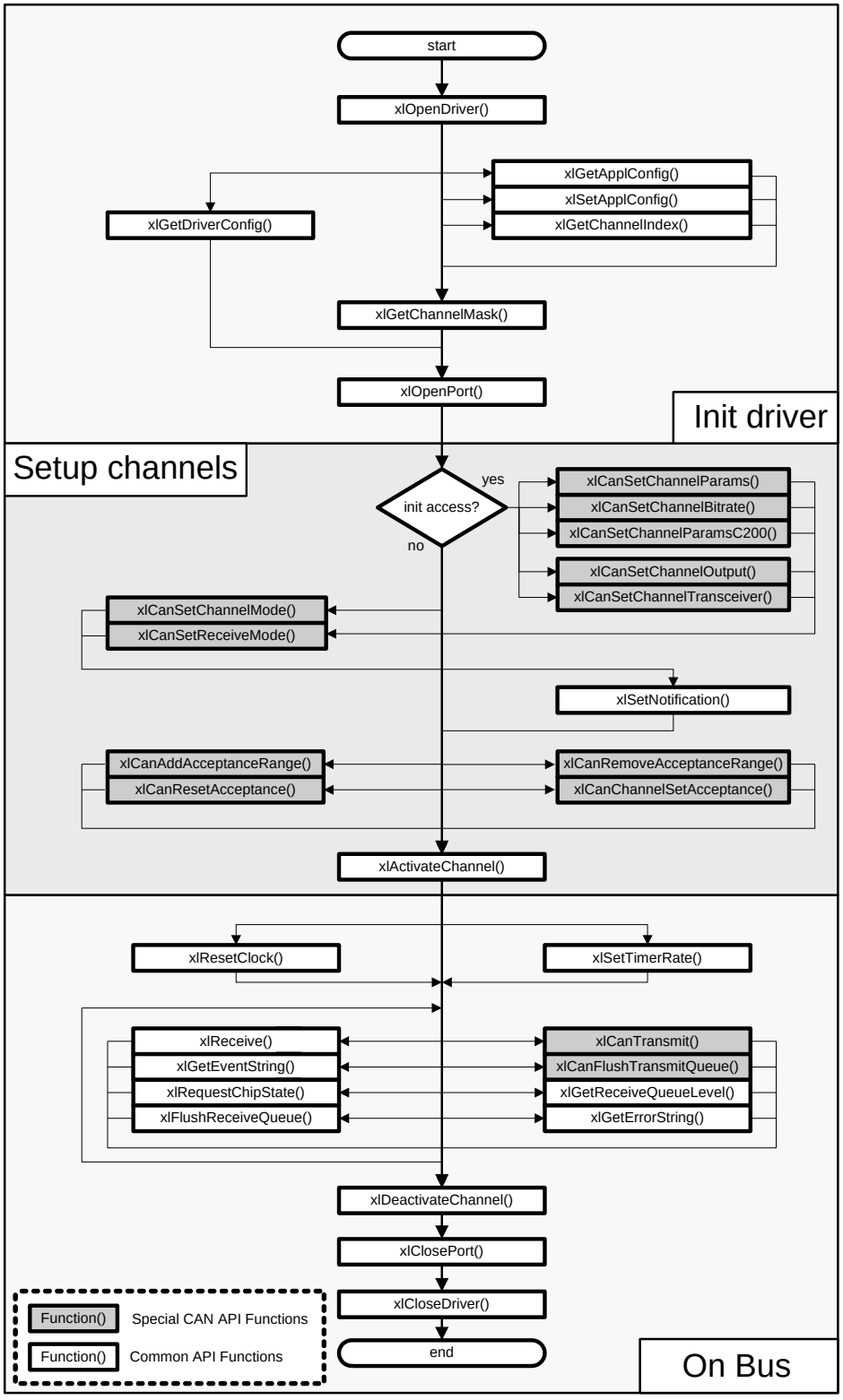


Figure 4: Function calls for CAN applications

2.4.2 CAN FD Application

Calling sequence

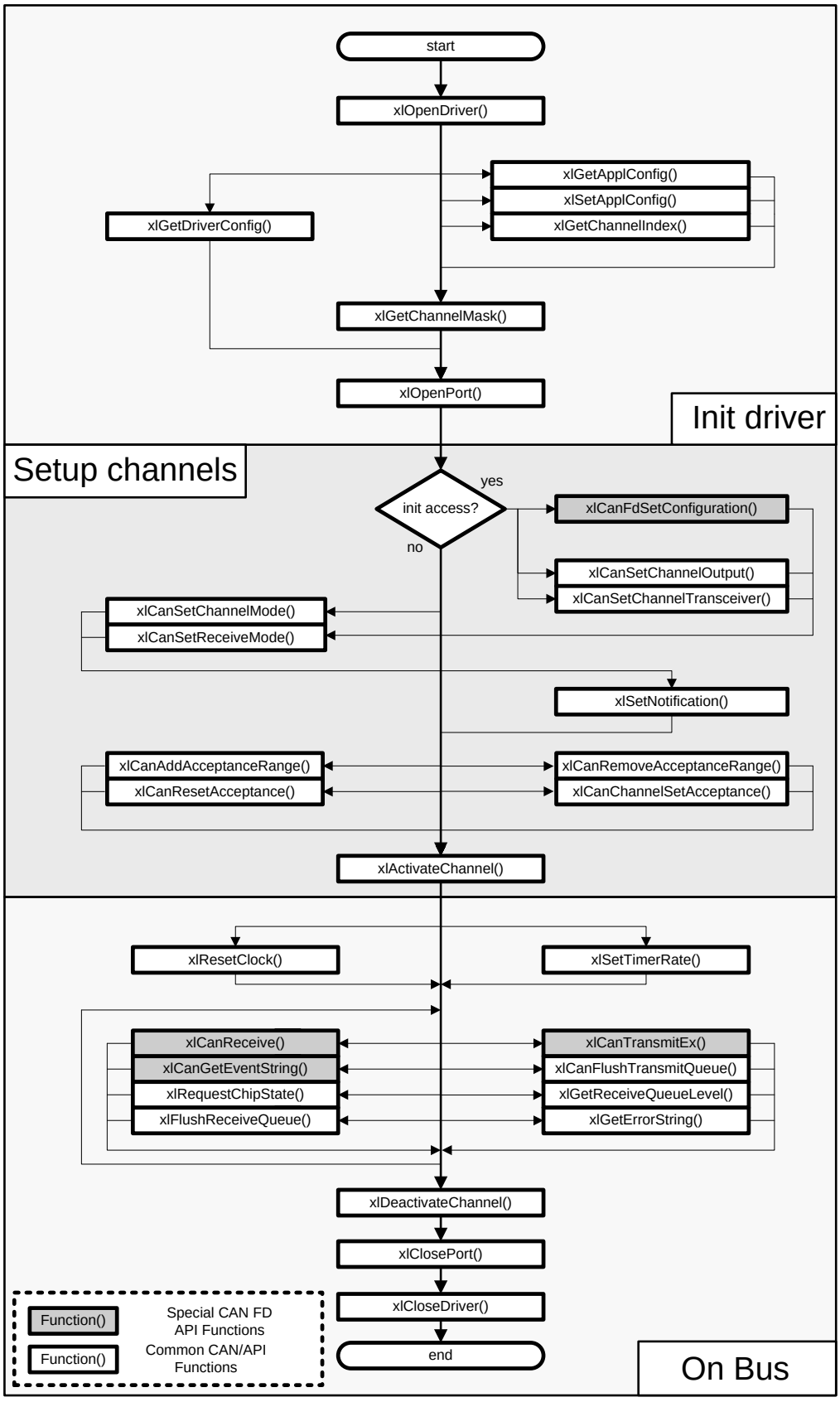


Figure 5: Function calls for CAN FD applications

2.4.3 LIN Application

Calling sequence

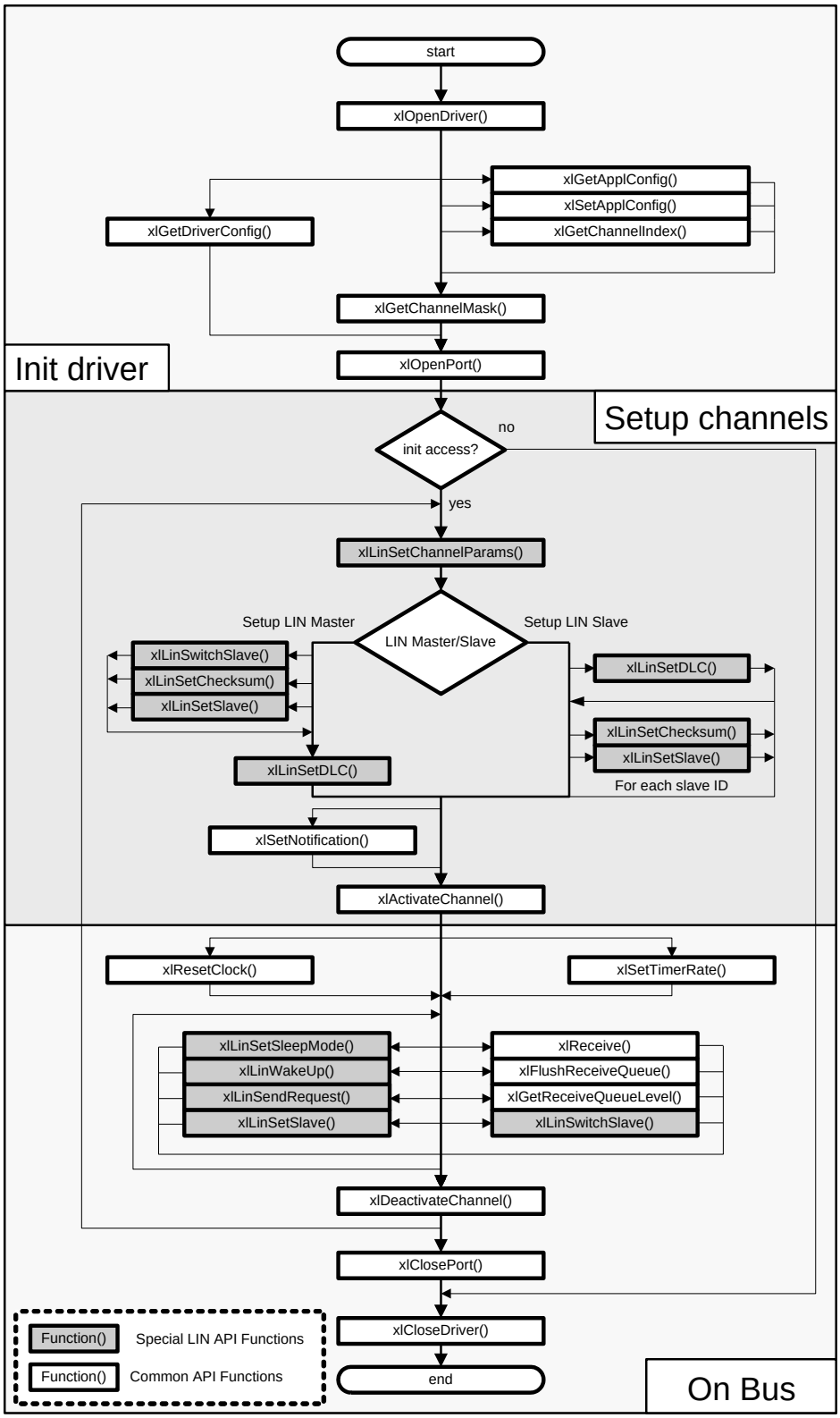


Figure 6: Function calls for LIN applications

2.4.4 DAIO CANcab Application

Calling sequence

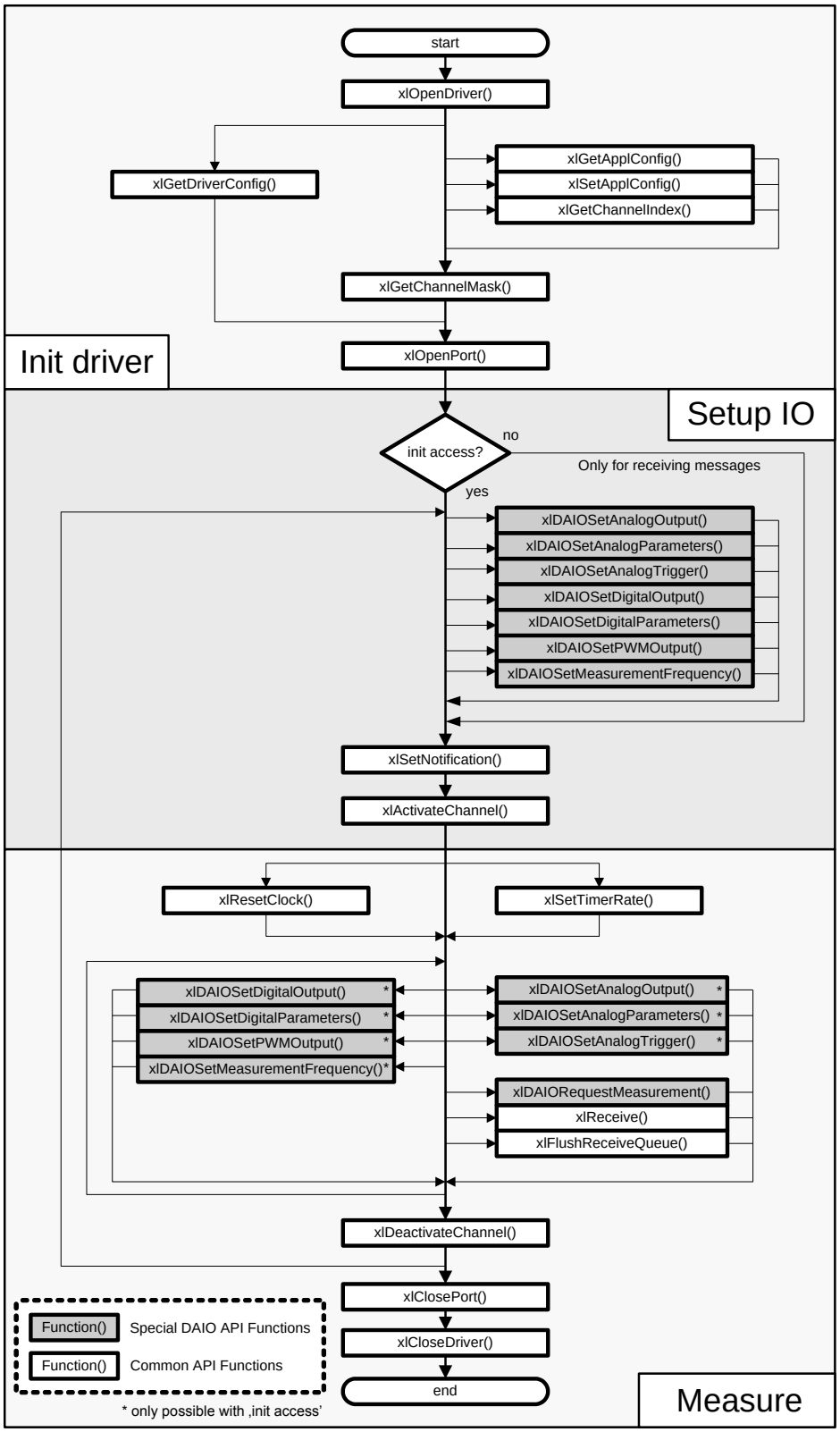


Figure 7: Function calls for DAIO CANcab applications

2.4.5 DAIO VN1630/VN1640 Application

Calling sequence

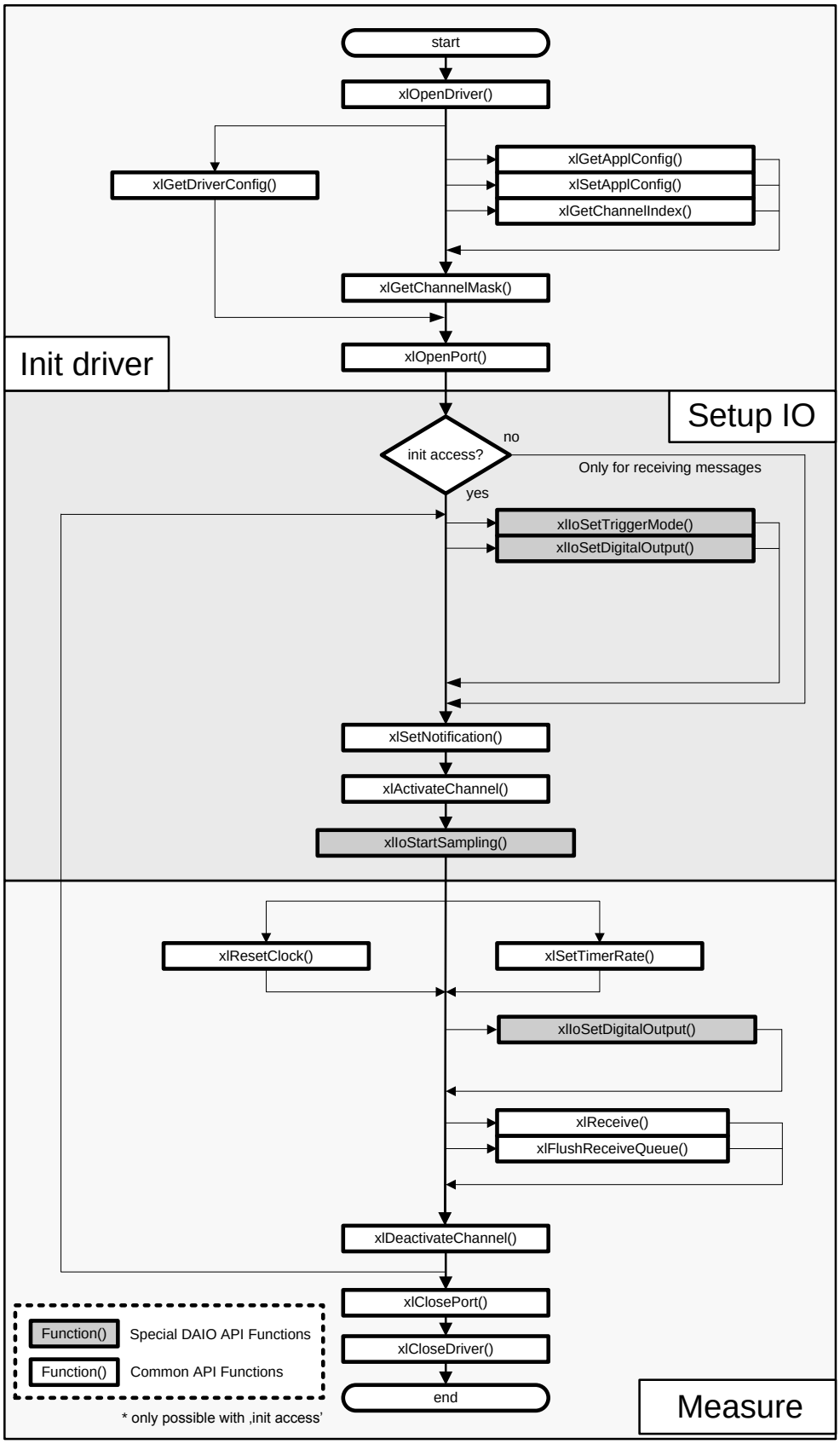


Figure 8: Function calls for DAIO VN1630/VN1640 applications

2.4.6 DAIO IOpiggy Application

Calling sequence

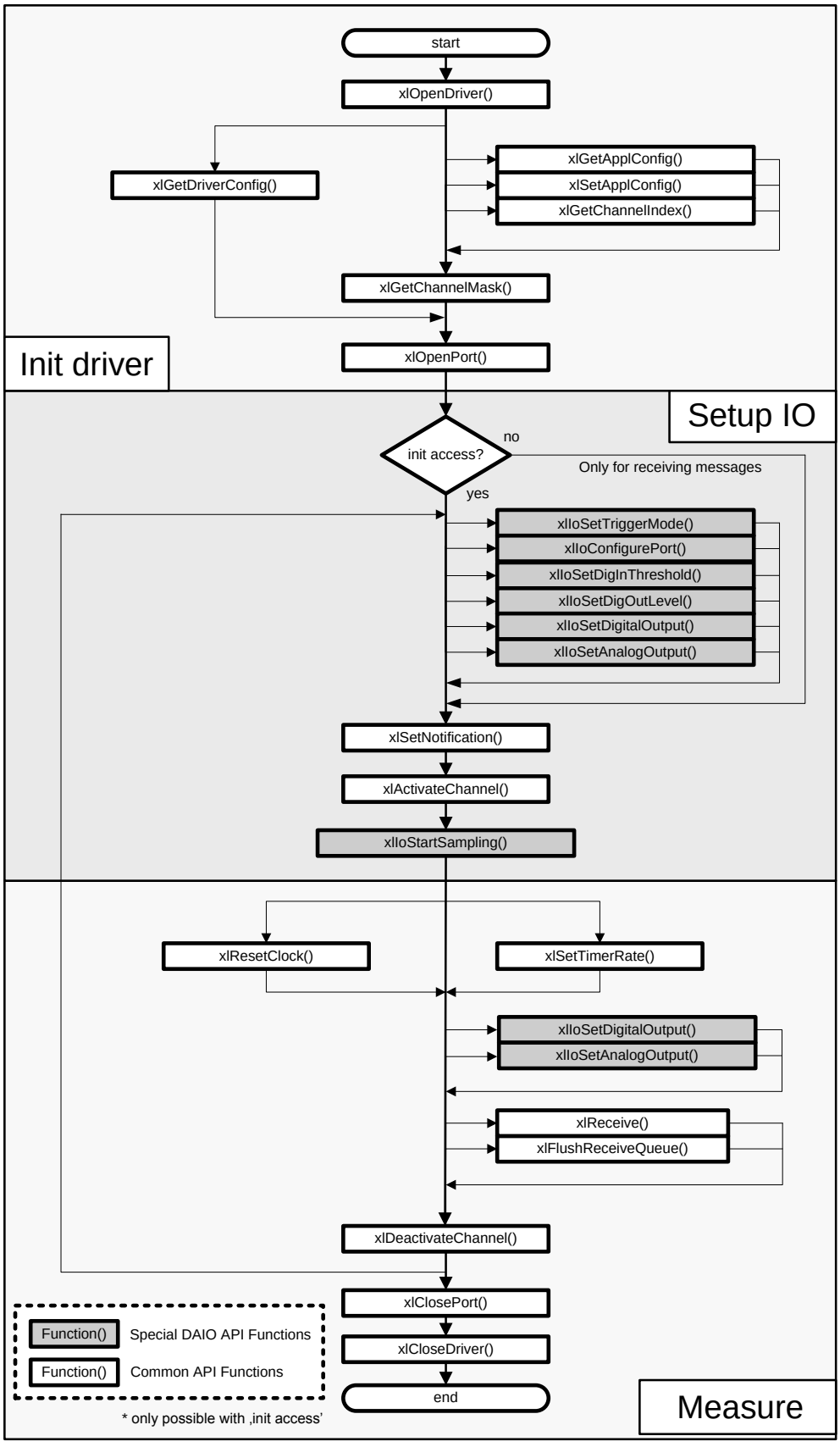


Figure 9: Function calls for IOpiggy applications

3 User API Description

In this chapter you find the following information:

3.1 Bus Independent Commands

3.1.1 xlOpenDriver

Syntax

```
XLstatus xlOpenDriver(void)
```

Description

Each application must call this function to load the driver. If this call is not successfully, no other API calls are possible.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.2 xlCloseDriver

Syntax

```
XLstatus xlCloseDriver(void)
```

Description

This function closes the driver.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.3 xlGetApplConfig

Syntax

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

Description

Retrieves information about the application assignment which is set in the **Vector Hardware Configuration** tool.

Input parameters

- **appName**
  Name of the application to be read.
  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 CANcardXL Channel 1 or “CANdemo CAN2” to CANcardXL Channel 2). Such an assignment has to be configured in Vector Hardware Config.
- **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`

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 page 124 for further details.

3.1.4 xlSetApplConfig

Syntax

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

Description

Creates a new application in Vector Hardware Config or sets the channel configuration in an existing application.

Input parameters

- **appName**
  Name of the application to be set.
- **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 124 for further details.

3.1.5 xlGetDriverConfig

Syntax

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

Description

Allows reading out more detailed information about the used hardware. This function can be called at any time after a successfully `xlOpenDriver`. The result describes the current state of the driver configuration after each call.

Input parameters

➔ **XLdriverConfig**
Points to a user buffer for the information which is returned by the driver.
See details below for further information.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLdriverConfig

The driver returns the following structure containing the information:

Syntax

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

Parameters

➔ **dllVersion**
The used dll version. (e.g. 0x300 means V3.0)

➔ **channelCount**
The number of available channels.

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

➔ **channel**
Structure containing channels information
(here `XL_CONFIG_MAX_CHANNELS=64`)

XLchannelConfig

The following sub structure is used in structure `XLdriverConfig` (above-mentioned).

```
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;
```

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**
  Only for internal use.

- **channelBusCapabilities**
  Describes the channel and the current transceiver features.

  The channel (hardware) supports the bus types:
  `XL_BUS_COMPATIBLE_CAN`
  `XL_BUS_COMPATIBLE_LIN`
  `XL_BUS_COMPATIBLE_DAIO`
  `XL_BUS_COMPATIBLE_HWSYNC`
  `XL_BUS_COMPATIBLE_MOST`
  `XL_BUS_COMPATIBLE_FLEXRAY`

  The connected Cab or Piggyback supports the bus type:
  `XL_BUS_ACTIVE_CAP_CAN`
  `XL_BUS_ACTIVE_CAP_LIN`
  `XL_BUS_ACTIVE_CAP_DAIO`
  `XL_BUS_ACTIVE_CAP_HWSYNC`
  `XL_BUS_ACTIVE_CAP_MOST`
  `XL_BUS_ACTIVE_CAP_FLEXRAY`
- **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.
- **driverVersion**
  Current driver version.
- **interfaceVersion**
  Current interface API version, e.g. `XL_INTERFACE_VERSION`
- **raw_data**
  Only for internal use.
- **serialNumber**
  Hardware serial number.
- **articleNumber**
  Hardware article number.
- **transceiverName**
  Name of the connected transceiver.
- **specialCabFlags**
  Only for internal use.
- **dominantTimeout**
  Only for internal use.
- **reserved**
  Reserved for future use. Set to 0.

XLbusParams

The following structure is used in structure `XLchannelConfig`.

```
typedef struct {
  unsigned int          busType;
  union {
    struct {
      unsigned int      bitRate;
      unsigned char     sjw;
      unsigned char     tseg1;
      unsigned char     tseg2;
      unsigned char     sam;
      unsigned char     outputMode;
    }can;
    unsigned char       raw[32];
  }data;
} XLbusParams;
```

Parameters

- **busType**
  Specifies the bus type for the application.
- **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.
- **raw**
  Only for internal use.

3.1.6 xlGetRemoteDriverConfig

Syntax

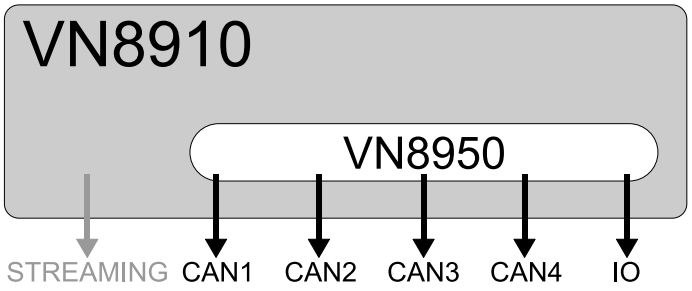
```
XLstatus xlGetRemoteDriverConfig(XLdriverConfig *pDriverConfig)
```

Description

This function is similar to `xlGetDriverConfig()`.

While `xlGetDriverConfig()` returns the driver configuration of a directly connected (host) device, `xlGetRemoteDriverConfig()` returns the driver configuration of the installed slide-in module (client) in a VN8910 device.

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



xlGetDriverConfig() for host:

channelCount	6
STREAMING internal use	channelIndex = 0 hwType = VN8910 hwChannel = 0 hwIndex = 0
CAN1 VN8950	channelIndex = 1 hwType = VN8910 hwChannel = 1 hwIndex = 0
CAN2 VN8950	channelIndex = 2 hwType = VN8910 hwChannel = 2 hwIndex = 0
CAN3 VN8950	channelIndex = 3 hwType = VN8910 hwChannel = 3 hwIndex = 0
CAN4 VN8950	channelIndex = 4 hwType = VN8910 hwChannel = 4 hwIndex = 0
IO VN8950	channelIndex = 5 hwType = VN8910 hwChannel = 5 hwIndex = 0

xlGetRemoteDriverConfig() for client:

channelCount	5
CAN1 VN8950	channelIndex = 0 hwType = VN8950 hwChannel = 0 hwIndex = 0
CAN2 VN8950	channelIndex = 1 hwType = VN8950 hwChannel = 1 hwIndex = 0
CAN3 VN8950	channelIndex = 2 hwType = VN8950 hwChannel = 2 hwIndex = 0
CAN4 VN8950	channelIndex = 3 hwType = VN8950 hwChannel = 3 hwIndex = 0
IO VN8950	channelIndex = 4 hwType = VN8950 hwChannel = 4 hwIndex = 0

Input parameters

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

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

Info: It is not possible to access the DLL version of the VN8910 through the parameter `dllVersion`. This parameter always returns 0.

3.1.7 xlGetChannelIndex

Syntax

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

Description

Retrieves the channel index of a particular hardware channel.

Input parameters

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

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

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

Return value

Returns the channel index.

3.1.8 xlGetChannelMask

Syntax

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

Description

Retrieves the channel mask of a particular hardware channel.

Input parameters

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

- **hwIndex**
  Required to distinguish between two or more devices of the same hardware type (-1, 0, 1…). Parameter `-1` 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.

3.1.9 xlOpenPort

Syntax

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

Description

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

Input parameters

- **userName**
  The name of the application that is listed in the Vector Hardware Configuration tool.
- **accessMask**
  Mask specifying which channels shall be used with this port. The accessMask can be retrieved by using `xlGetChannelMask`.
- **rxQueueSize**
  CAN, LIN, DAIO
  Size of the port receive queue allocated by the driver. Specifies how many events can be stored in the queue. The value must be a power of 2 and within a range of 16…32768. The actual queue size is `rxQueueSize-1`.

  MOST, FlexRay
  Size of the port receive queue allocated by the driver in bytes.
- **xlInterfaceVersion**
  Current API version, e.g.
  `XL_INTERFACE_VERSION` for CAN, LIN, DAIO.
  `XL_INTERFACE_VERSION_V4` for MOST, CAN FD.
- **busType**
  Bus type that should be activated, e.g.
  `XL_BUS_TYPE_LIN` to initialize LIN
  `XL_BUS_TYPE_CAN` to initialize CAN
  `XL_BUS_TYPE_DAIO` to initialize DAIO
  `XL_BUS_TYPE_MOST` to initialize MOST
  `XL_BUS_TYPE_FLEXRAY` to initialize FlexRay

Output parameters

- **portHandle**
  Pointer to a variable, where the `portHandle` is returned. This handle must be used for any further calls to the port. If `-1` is returned, the port was neither created nor opened.

Input/Output Parameters

- **permissionMask**
  on output
  Pointer to a variable where the mask is returned for the channel for which init access is granted.

  on input
  As input there must be the channel mask where **init access** is requested.

A LIN channel needs init access.

Return value

Returns an error code. For LIN (`busType = XL_BUS_TYPE_LIN`) **init access** is **needed**. If the channel gets no **init access** the function returns `XL_ERR_INVALID_ACCESS`.
See section Error Codes on page 124 for further details.

Example: Access Mask

This example should help to understand the meanings of channel index and channel mask (access mask). Channels are identified by their channel index. Most functions expect a bit mask (called access mask) to identify multiple channels. The bit mask is constructed by: `access mask = 1<<channel index`

To get access to more than one channel, it is needed to merge (add) all wanted channels: $\sum$*wanted_access_masks*

The following example is a possible configuration.

Hardware	Hardware Channel	Channel Index	Access Mask (hex)	Access Mask (bin)
CANcardXL	Channel 01	0	0x01	00000**1**
	Channel 02	1	0x02	00001**0**
CANcaseXL	Channel 01	2	0x04	000**1**00
	Channel 02	3	0x08	00**1**000
CANboardXL	Channel 01	4	0x10	0**1**0000
	Channel 02	5	0x20	**1**00000
All above-mentioned	All above-mentioned	All above-mentioned	0x3F	**111111**

Example: Select CANcardXL channel 1

```
m_xlChannelMask = xlGetChannelMask(XL_HWTYPE_CANCARDXL,-1,  0);
if(!m_xlChannelMask) return XL_ERR_HW_NOT_PRESENT;
xlPermissionMask = m_xlChannelMask;

xlStatus = xlOpenPort(&m_XLportHandle, "xlCANdemo",
                      m_xlChannelMask, &xlPermissionMask,
                      1024, XL_INTERFACE_VERSION,
                      XL_BUS_TYPE_CAN);
```

Example: Open port with two channels with 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);
```

3.1.10 xlClosePort

Syntax

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

Description

The port is closed and the channels are deactivated.

Input parameters

→ **portHandle**
The port handle retrieved by `xlOpenPort`.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.11 xlSetTimerRate

Syntax

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

Description

This call sets the rate for the port's cyclic timer events.

The resolution of `timeRate` is 10 µs, but the internal step width is 1000 µs. Values less than multiples of 1000 µs will be rounded down (truncated) to the next closest value.

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

The minimum timer rate value is 1000 µs (`timerRate = 100`).

If more than one application uses the timer events the lowest value will be used for both.

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

Info: 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 124 for further details.

3.1.12 xlSetTimerRateAndChannel

Syntax

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

Description

This call sets the rate for the port's cyclic timer events. The resolution is 10 µs (`timerRate` of 1 means 10 µs, a `timerRate` of 10 means 100 µs). The minimum and maximum `timerRate` values depend on the hardware. If a value is outside of the allowable range the limit value is used. Only deterministic values according to the following list can be used. Other values will be rounded to the next faster timerrate.

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

Info: 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.
If the function call succeeds, XL_SUCCESS will be returned. Otherwise XL_ERROR, XL_ERR_INVALID_HANDLE or XL_ERR_INVALID_ACCESS.

3.1.13 xlResetClock

Syntax

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

Description

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

Input parameters

➔ **portHandle**
The port handle retrieved by `xlOpenPort`.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.14 xlSetNotification

Syntax

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

Description

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

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

Input parameters

➔ **portHandle**
The port handle retrieved by `xlOpenPort`.

➔ **queueLevel**
Queue level that triggers this event. For LIN it is fixed to ‘1’.

Output parameters

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

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

Example: Setup the notification for a CAN application

```
Xlhandle h;
xlStatus = xlSetNotification (gPortHandle, &h, 1);

// Wait for event
while (WaitForSingleObject(h,1000) == WAIT_TIMEOUT);
do {
  // Get the event
  xlStatus = xlReceive(gPortHandle, 1, &pEvent);
} while (xlErr == 0);
```

3.1.15 xlFlushReceiveQueue

Syntax

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

Description

The 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 124 for further details.

3.1.16 xlGetReceiveQueueLevel

Syntax

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

Description

The function returns the count of events in the port’s receive queue.

Input parameters

→ **portHandle**
The port handle retrieved by `xlOpenPort`.

Output parameters

→ **level**
Pointer to an int where the actual count of events in the receive queue is returned.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.17 xlActivateChannel

Syntax

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

Description

Goes ‚on bus’ for the selected port and channels. (Starts the measurement). At this point the user can transmit and receive messages on the bus. For LIN the **master**/**slave** must be parameterized before.

Input parameters

→ **portHandle**
The port handle retrieved by `xlOpenPort`.

→ **accessMask**
The access mask must contain the mask of channels to be activated.

→ **busType**
Bus type that should be activated. e.g.
`XL_BUS_TYPE_LIN` to initialize LIN
`XL_BUS_TYPE_CAN` to initialize CAN, …

➔ **flags**
Additional flags for activating the channels.
`XL_ACTIVATE_RESET_CLOCK`
reset the internal clock after activating the channel.

`XL_ACTIVATE_NONE`

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

Example: Channel Activation

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

3.1.18 xlReceive

Syntax

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

Description

Reads the received events from the message queue. 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_CAN_MSG_FLAG_OVERRUN` in `XLevent.tagData.msg.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. The buffer must be big enough to hold the requested messages (pEventCount).

Return value

`XL_ERR_QUEUE_IS_EMPTY`: No event is available.
See section Error Codes on page 124 for further details.

Example: Read each message from the message queue

```
XLhandle          h;
unsigned int      msgsrx = 1;
XLevent           xlEvent;

vErr = xlSetNotification(XLportHandle, &h, 1);

// Wait for event
while (g_RXThreadRun) {
```

```
    WaitForSingleObject(g_hMsgEvent,10);
    msgsrx    = RECEIVE_EVENT_SIZE;
    xlStatus = xlReceive(g_XLportHandle, &msgsrx, &xlEvent);
    while (!xlStatus) {
      if (xlStatus != XL_ERR_QUEUE_IS_EMPTY ) {
        printf("%s\n", xlGetEventString(&xlEvent));
        msgsrx    = 1;
        xlStatus = xlReceive(g_XLportHandle,
                             &msgsrx,
                             &xlEvent);
      }
    }
  }
```

3.1.19 xlGetEventString

Syntax

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

Description

Returns a textual description of the given event.

Input parameters

→ **ev**
Points to the event.

Return value

Text string.

Example: Received 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 : with a timestamp of 794034375ns,
id=004 : the ID=4
l=8 : a DLC of 8 and
0000000000000: D0 to D7 are set to 0.
TX tid=CC : TX flag, message was transmitted successfully by the CAN controller.

3.1.20 xlGetErrorString

Syntax

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

Description

Returns a textual description of the given error.

Input parameters

→ **err**
Error code. See section Error Codes on page 124 for further details.

Return value

Error code as plain text string.

3.1.21 xlGetSyncTime

Syntax

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

Description

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

Note: If the software time synchronization is active, the event timestamp is synchronized to the PC time. If the XL API function `xlResetClock()` was not called, the event timestamp 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, where the sync time is received.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.22 xlGetChannelTime

Syntax

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

Description

This function is available only on VN89xx devices and returns the 64 bit PC-based card time.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.

Output parameters

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

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.23 xlGenerateSyncPulse

Syntax

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

Description

This function generates a sync pulse at the hardware sync 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 must contain the mask of channels at which the sync pulse shall be generated.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.24 xlPopupHwConfig

Syntax

```
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 124 for further details.

3.1.25 xlDeactivateChannel

Syntax

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

Description

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

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be deactivated.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.1.26 xlGetLicenseInfo

Syntax

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

Description

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

Input parameters

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

Output parameters

- **pLicInfoArray**
  Pointer to array to be returned.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

Syntax

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

Parameters

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

Example: Retrieving licenses, check if available

```
XLstatus xlStatus;
char licAvail[2048];
char strtmp[512];

XLlicenseInfo licenseArray[1024];
unsigned int  licArraySize = 1024;

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


if (xlStatus == XL_SUCCESS) {
    strcpy(licAvail, "Licenses found:\n\n");
    for (unsigned int i = 0; i < licArraySize; i++) {

      if (licenseArray[i].bAvailable) {

        sprintf(strtmp,
                "ID 0x%03x: %s\n", i,
                licenseArray[i].licName);

        if ((strlen(licAvail) + strlen(strtmp)) <
             sizeof(licAvail)) {

          strcat(licAvail, strtmp);

        }

        else {

          sprintf(licAvail, "Error: String size too small!");
          xlStatus = XL_ERROR;

        }

      }

    }

  }

  else {

    sprintf(licAvail, "Error: %d", xlStatus);

  }
```

3.1.27 xlSetGlobalTimeSync

Syntax

```
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.

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 124 for further details.

3.2 CAN Commands

3.2.1 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's 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.

Info: To call this function the port must have **init access** (see `xlOpenPort`) for the specified channels, and the channels must be deactivated.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **mode**
  Specifies the output mode of the CAN chip.
  `XL_OUTPUT_MODE_SILENT`
  No acknowledge will be generated on receive (silent mode).
  Note: With driver version V5.5 the silent mode has been changed. Now the TX pin is switched off. (The 'SJA1000 silent mode' is not used anymore).

  `XL_OUTPUT_MODE_NORMAL`
  Acknowledge (normal mode)

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.2.2 xlCanSetChannelMode

Syntax

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

Description

This sets 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 must contain the mask of channels to be accessed.

- **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 124 for further details.

3.2.3 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

- **Port**
  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 124 for further details.

3.2.4 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 `xlOpenPort`) to the channels.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **type**
  Lowspeed (252/1053/1054)
  `XL_TRANSCEIVER_TYPE_CAN_252`

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

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

  Truck & Trailer
  `XL_TRANSCEIVER_TYPE_CAN_B10011S`
  `XL_TRANSCEIVER_TYPE_PB_CAN_TT_OPTO`

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

  `XL_TRANSCEIVER_LINEMODE_NORMAL`
  Enables normal operation.

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

  `XL_TRANSCEIVER_LINEMODE_NORMAL`
  Enables normal operation.

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

  `XL_TRANSCEIVER_LINEMODE_SWC_SLEEP`
  Switches to sleep mode.

  `XL_TRANSCEIVER_LINEMODE_SWC_NORMAL`
  Switches to normal operation.

  `XL_TRANSCEIVER_LINEMODE_SWC_FAST`
  Switches transceiver to fast mode.

  Truck & Trailer
  `XL_TRANSCEIVER_LINEMODE_NORMAL`
  Normal operation on CAN high and CAN low.

  `XL_TRANSCEIVER_LINEMODE_TT_CAN_H`
  One wire mode on CAN high.

`XL_TRANSCEIVER_LINEMODE_TT_CAN_L`
One wire mode on CAN low.

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

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.2.5 xlCanSetChannelParams

Syntax

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

Description

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

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **pChipParams**
  Pointer to an array of chip parameters. See below for further details.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLchipParams

The structure for the chip parameters is defined as follows:

Syntax

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

Parameters

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

Info: For more information about the bit timing of the CAN controller please refer to some of the CAN literature or CAN controller data sheets.

Example: Calculation of baudrate

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

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

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

3.2.6 xlCanSetChannelParamsC200

Syntax

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

Description

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

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 124 for further details.

3.2.7 xlCanSetChannelBitrate

Syntax

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

Description

`xlCanSetChannelBitrate` provides a simple way to specify the bit rate. The sample point is about 65%.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **bitrate**
  Bit rate in BPS. May be in the range 15000 … 1000000.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.2.8 xlCanSetChannelAcceptance

Syntax

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

Description

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

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

Info: The acceptance filters are open after an `xlOpenPort` by default.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 124 for further details.

Example: Several acceptance filter settings

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

Example: Open filter for all standard message IDs

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

Example: Set acceptance filter for several IDs (formula)

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

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

3.2.9 xlCanAddAcceptanceRange

Syntax

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

Description

The filters are opened (all messages are received) by default. `xlCanAddAccep-tanceRange` opens the filters for the specified range of standard IDs. The function 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.

Info: The acceptance filters are **open** after `xlOpenPort` by default. This function is for **standard IDs** only. For selecting an ID range maybe the filters must be closed before.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **first_id**
  First ID to pass acceptance filter.
- **last_id**
  Last ID to pass acceptance filter.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

Example: Receive ID between 10…17 and 22…33

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

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

3.2.10 xlCanRemoveAcceptanceRange

Syntax

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

Description

The specified IDs will not pass the acceptance filter. `xlCanRemove-AcceptanceRange` is only implemented for standard identifier. 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.

Info: The acceptance filters are **open** after `xlOpenPort` by default. This function is for **standard IDs** only.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.

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

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

Example: Remove range between 10…13 and 27…30

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

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

3.2.11 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 must contain the mask of channels to be accessed.
- **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 124 for further details.

Example: Open filter for all messages with extended IDs

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

3.2.12 xlCanRequestChipState

Syntax

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

Description

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

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.2.13 xlCanTransmit

Syntax

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

Description

The function transmits CAN messages on the selected channels. It is possible to transmit more messages with one `xlCanTransmit` call (see the following example).

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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`.

Info: 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 124 for further details.

Example: Transmit 100 CAN messages with the ID = 4

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

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

```
  xlEvent[i].tagData.msg.dlc     = 8;
}
xlStatus = xlCanTransmit(portHandle, accessMask,
                         &nCount, xlEvent);
```

3.2.14 xlCanFlushTransmitQueue

Syntax

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

Description

The function flushes the transmit queues of the selected channels.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  Mask specifying which channels shall be used with this port.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.3 CAN FD Commands

3.3.1 General Information

CAN FD support

The XL API offers specific CAN FD functions for use with your Vector device.

Note: It is not possible to mix standard CAN and CAN FD settings for the same channel.

To figure out which CAN channel supports CAN FD please call the `xlGetDriverConfig()` function first. Use the `channelCapabilities` flags to determine the corresponding CAN channel (`XL_CHANNEL_FLAG_CANFD_SUPPORT`).

Example: Searching a CAN FD channel after calling `xlGetDriverConfig()`:

```
if (g_xlDrvConfig.channel[i].channelBusCapabilities &
    XL_BUS_ACTIVE_CAP_CAN) {


  // check if CAN FD is supported
  if (g_xlDrvConfig.channel[i].channelCapabilities &
      XL_CHANNEL_FLAG_CANFD_SUPPORT) {

    xlChannelMaskFd |= g_xlDrvConfig.channel[i].channelMask;
  }

  else {
    g_xlChannelMask |= g_xlDrvConfig.channel[i].channelMask;
  }
}
```

RX queue

CAN FD uses the Rx queue version 4 (`XL_INTERFACE_VERSION_V4`) of the XL API. The size can be in a range of `RX_FIFO_CANFD_QUEUE_SIZE_MIN` and `RX_FIFO_CANFD_QUEUE_SIZE_MAX`.

The queue version has to be set in `xlOpenPort()`.

Supported functions

CAN FD uses the following functions:

- `xlGetDriverConfig()`
  General XL API function. Finds a CAN FD channel.
- `xlOpenPort()`
  General XL API function. Opens a port with the V4 queue.
- `xlCanFdSetConfiguration()`
  Specific CAN FD function. Sets the CAN FD channel.
- `xlCanTransmitEx()`
  Specific CAN FD function. Sends a CAN FD message.
- `xlCanReceive()`
  Specific CAN FD function. Receives a CAN FD message.
- `xlCanGetEventString()`
  Specific CAN FD function. Builds an RX event string for CAN FD messages.

3.3.2 xlCanFdSetConfiguration

Syntax

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

Description

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

Info: To call this function the port must have **init access** (see `xlOpenPort`) for the specified channels.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **pCanFdConf**
  Points to the CAN FD configuration structure to set up a CAN FD channel.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLcanFdConf

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

Input parameters

- **arbitrationBitRate**
  Arbitration CAN bus timing for nominal / arbitration bit rate.
- **sjwAbr**
  Arbitration CAN bus timing value (sample jump width).
- **tseg1Abr**
  Arbitration CAN bus timing tseg1.

  tseg2Abr
  Arbitration CAN bus timing tseg2.
- **dataBitRate**
  CAN bus timing for data bit rate.
- **sjwDbr**
  CAN bus timing value (sample jump width).
- **tseg1Dbr**
  CAN bus timing for data tseg1.

- **tseg2Dbr**
  CAN bus timing for data tseg2.
- **reserved**
  Reserved for future use. Set to 0.

3.3.3 xlCanTransmitEx

Syntax

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

Description

The function transmits CAN FD messages on the selected channels. It is possible to send multiple messages in a row (with a single call). See section xlCanTransmit on page 56.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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.
  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 124 for further details.

3.3.4 xlCanReceive

Syntax

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

Description

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

Input parameters

➔ **portHandle**
The port handle retrieved by `xlOpenPort`.

Input/ output parameters

➔ **pXLCanRxEvt**
Pointer to the application allocated receive event buffer.

Return value

`XL_ERR_QUEUE_IS_EMPTY`: No event is available.
See section Error Codes on page 124 for further details.

3.3.5 xlCanGetEventString

Syntax

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

Description

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

Input parameters

➔ **pEv**
Points the CAN Rx event buffer to be parsed.

3.4 LIN Commands

3.4.1 xlLinSetChannelParams

Syntax

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

Description

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

Info: 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 must contain the mask of channels to be accessed.
- **statPar**
  Defines the mode of the LIN channel and the baud rate.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLlinStatPar

The following structure is used in function `xlLinSetChannelParams`:

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

Parameters

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

  `XL_LIN_SLAVE`
  Set channel to LIN slave.
- **baudrate**
  Set the baud rate. e.g. 9600, 19200, ...
  The baud rate range is 200 … 30.000 Bd. Please note that the functionality of the XL API is guaranteed for 200 … 20.000 Bd according to the LIN specification. Higher values should be used with care.
- **LINVersion**
  `XL_LIN_VERSION_1_3`
  Use LIN 1.3 protocol

  `XL_LIN_VERSION_2_0`
  Use LIN 2.0 protocol

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

Example: Channel setup as a SLAVE to 9k6 and LIN 1.3

```
XLlinStatPar xlStatPar;

xlStatPar.LINMode    = XL_LIN_SLAVE;
xlStatPar.baud rate  = 9600;

// use LIN 1.3
xlStatPar.LINVersion = XL_LIN_VERSION_1_3;

xlStatus = xlLinSetChannelParams(m_XLportHandle,
                                 m_xlChannelMask[SLAVE],
                                 xlStatPar);
```

3.4.2 xlLinSetDLC

Syntax

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

Description

Defines the data length for all requested messages. This is needed for the LIN master (and recommended for LIN slave) and must be called **before** activating a channel.

Input parameters

➔ **portHandle**
The port handle retrieved by `xlOpenPort`.

➔ **accessMask**
The access mask must contain the mask of channels to be accessed.

➔ **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 124 for further details.

Example: Set DLC for LIN message with ID 0x04 to 8 and for all other IDS to undefined.

```
unsigned char DLC[64];
for (int i=0;i<64;i++) DLC[i] = XL_LIN_UNDEFINED_DLC;
DLC[4] = 8;

xlStatus = xlLinSetDLC(m_XLportHandle, m_xlChannelMask[MASTER],
DLC);
```

3.4.3 xlLinSetChecksum

Syntax

```
XLstatus xlLinSetChecksum (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned char   checksum[60])
```

Description

This function is only for a LIN 2.0 node and must be called before activating a channel. The checksum calculation can be changed here from the classic to enhanced model for the LIN IDs 0..59. The LIN ID 60..63 range is fixed to the classic model and cannot be changed. The classic model is always set for all IDs by default. There are no changes when it is called for a LIN 1.3 node.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 124 for further details.

Example: Set the checksum for a LIN message with the ID 0x04 to “enhanced” and for all other IDs to “undefined”.

```
unsigned char checksum[60];
for (int i = 0; i < 60; i++)
  checksum[i] = XL_LIN_CHECKSUM_UNDEFINED;
checksum[4] = XL_LIN_CHECKSUM_ENHANCED;
xlStatus =
xlLinSetChecksum(m_XLportHandle,
                 m_xlChannelMask[MASTER],
                 checksum);
```

3.4.4 xlLinSetSlave

Syntax

```
XLstatus xlLinSetSlave (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned char   linId,
  unsigned char   data[8],
  unsigned char   dlc,
  unsigned short  checksum)
```

Description

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

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

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 124 for further details.

Example: Setup 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);
```

3.4.5 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 must contain the mask of channels to be accessed.
- **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 124 for further details.

3.4.6 xlLinSendRequest

Syntax

```
XLstatus xlLinSendRequest (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned char   linId,
  unsigned int    flags)
```

Description

Sends a master LIN request to the slave(s).
After a 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 must contain the mask of channels to be accessed.
- **linID**
  Contains the master request LIN ID.
- **flags**
  For future use. Set to 0.

Return value

Returns an error code.
Returns `XL_ERR_INVALID_ACCESS` if it is done on a LIN slave. See section Error Codes on page 124 for further details.

3.4.7 xlLinWakeUp

Syntax

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

Description

Transmits a wake-up signal.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.4.8 xlLinSetSleepMode

Syntax

```
XLstatus xlLinSetSleepMode (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    flags,
  unsigned char   linId)
```

Description

Activates the sleep mode.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 wakeup 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 124 for further details.

3.5 Digital/Analog Input/Output Commands for CANcab

3.5.1 xlDAIOSetAnalogParameters

Syntax

```
XLstatus xlDAIOSetAnalogParameters (
  XLportHandle  portHandle,
  XLaccess      accessMask,
  unsigned int  inputMask,
  unsigned int  outputMask,
  unsigned int  highRangeMask)
```

Description

Configures the analog lines. All lines are set to input by default. The bit sequence to access the physical pins on the D-SUB15 connector is as follows:

- AIO0 = 0001 (0x01)
- AIO1 = 0010 (0x02)
- AIO2 = 0100 (0x04)
- AIO3 = 1000 (0x08)

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 … 8.192V (3.1kHz)
  - High range 0 … 32.768V (6.4kHz)
  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 124 for further details.

Example: Setup 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)

3.5.2 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 must contain the mask of channels to be accessed.
- **analogLine1**
  Voltage level for AIO0.
- **analogLine2**
  Voltage level for AIO1.
- **analogLine3**
  Voltage level for AIO2.
- **analogLine4**
  Voltage level for AIO3.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.5.3 xlDAIOSetAnalogTrigger

Syntax

```
XLstatus xlDAIOSetAnalogTrigger (
  XLportHandle portHandle,
  XLaccess     accessMask,
  unsigned int triggerMask,
  unsigned int triggerLevel,
  unsigned int triggerEventMode)
```

Description

Configures analog trigger functionality.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 124 for further details.

3.5.4 xlDAIOSetDigitalParameters

Syntax

```
XLstatus xlDAIOSetDigitalParameters (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    inputMask,
  unsigned int    outputMask)
```

Description

Configures the digital lines. All lines are set to input by default. The bit sequence to access the physical pins on the D-SUB15 connector is as follows:

- DAIO0: `0b0000000`**`1`**
- DAIO1: `0b000000`**`1`**`0`
- DAIO2: `0b00000`**`1`**`00`
- DAIO3: `0b0000`**`1`**`000`
- DAIO4: `0b000`**`1`**`0000`
- DAIO5: `0b00`**`1`**`00000`

- DAIO6: 0b01000000
- DAIO7: 0b10000000

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **inputMask**
  Mask for lines to be configured as input. Generally the inverted value of the output mask will be used.
- **outputMask**
  Mask for lines to be configured as output. A set output line affects always a defined second digital line.

Caution: The digital outputs consist internally of electronic switches (photo MOS relays) and need always two digital lines of the IOcab 8444opto: a general output line and a line for external supply. In other words: When the switch is closed (by software), the applied voltage can be measured at the second output line, otherwise not. The line pairs are defined as follows: DIO0/DIO1, DIO2/DIO3, DIO4/DIO5 and DIO6/DIO7.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.5.5 xlDAIOSetDigitalOutput

Syntax

```
XLstatus xlDAIOSetDigitalOutput (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    outputMask,
  unsigned int    valuePattern)
```

Description

Sets digital output line to desired logical level.

Input parameters

- **portHandle**
  The port handle retrieved by xlOpenPort.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **outputMask**
  Switches to be changed:
  - DAIO0/DAIO1: 0b0001
  - DAIO2/DAIO3: 0b0010
  - DAIO4/DAIO5: 0b0100
  - DAIO6/DAIO7: 0b1000
- **valuePattern**
  Mask specifying the switch state for digital output.
  - DAIO0/DAIO1: 0b000x
  - DAIO2/DAIO3: 0b00x0
  - DAIO4/DAIO5: 0b0x00
  - DAIO6/DAIO7: 0bx000

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

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

Example: Setup the IOcab8444

```
outputMask   = 0x05(0b0101) Update digital output DIO0/DIO1 and DIO4/DIO5
valuePattern = 0x01(0b0001) Close relay DIO0/DIO1
                            Open relay DIO4/DIO5
```

3.5.6 xlDAIOSetPWMOutput

Syntax

```
XLstatus xlDAIOSetPWMOutput (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    frequency,
  unsigned int    value)
```

Description

Changes PWM output to defined frequency and value.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **frequency**
  Set PWM frequency to specified value in Hertz.
  Allowed values: 40…500 Hertz and 2.4kHz…100kHz
- **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 124 for further details.

Example: Setup the IOcab8444

`frequency`	`= 2500`	PWM frequency is now 2500 Hz
`value`	`= 2500`	PWM ratio is now 25% (75% pulse low, 25% pulse high)

3.5.7 xlDAIOSetMeasurementFrequency

Syntax

```
XLstatus xlDAIOSetMeasurementFrequency (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    measurementInterval)
```

Description

Sets the measurement frequency. xlEvents will be automatically triggered, which can be received by `xlReceive`. For manual trigger see chapter xlDAIORequestMeasurement on page 76.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **measurementInterval**
  Measurement frequency in ms.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.5.8 xlDAIORequestMeasurement

Syntax

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

Description

Forces manual measurement of DAIO values.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.6 Digital/Analog Input/Output Commands for VN1630A/VN1640A

3.6.1 xlIoSetTriggerMode

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 must contain the mask of channels to be accessed.
- **pxlDaioTriggerMode**
  Use this structure to define the trigger type (see structure definition in section xlIoSetTriggerMode on page 77).
  Note: Currently only `XL_DAIO_TRIGGER_TYPE_CYCLIC` is supported.

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

3.6.2 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 must contain the mask of channels to be accessed.
- **pxlDaioDigitalParams**
  Use this structure to set the value of the digital out pin (see below).

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLdaioDigitalParams

```
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

3.7 Digital/Analog Input/Output Commands for IOpiggy

3.7.1 xlIoSetTriggerMode

Syntax

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

Description

Sets the DAIO trigger mode for the analog and digital ports.

Note: This command can be called only once per port type (analog and digital) and only when the channel is deactivated (see flowchart example in section DAIO IOpiggy Application on page 20).

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **pxlDaioTriggerMode**
  Use this structure to define the trigger type (see below).

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLdaioTriggerMode

```
typedef struct s_xl_daio_trigger_mode {
  unsigned int portTypeMask;
  unsigned int triggerType;

  union triggerTypeParams {
    unsigned int cycleTime;
    struct {
      unsigned int portMask;
      unsigned int type;
    } digital;
  } param;

} XLdaioTriggerMode;
```

Parameters

- **portTypeMask**
  Defines the port type:
  `XL_DAIO_PORT_TYPE_MASK_ANALOG`
  `XL_DAIO_PORT_TYPE_MASK_DIGITAL`
- **triggerType**
  Defines the trigger type:
  `XL_DAIO_TRIGGER_TYPE_CYCLIC` (for analog and digital port type)
  `XL_DAIO_TRIGGER_TYPE_PORT` (for digital port type)
- **cycleTime**
  For use with `XL_DAIO_TRIGGER_TYPE_CYCLIC`.
  Cyclic trigger time in µs (`1000...1048575`).

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

fired. During a cycle additional events may also be fired, e.g. if the digital IO pin toggles.

➔ **portMask**
For use with `XL_DAIO_TRIGGER_TYPE_PORT`.
Specifies the digital port (D0…D07):
`XL_DAIO_PORT_MASK_DIGITAL_D0`
`XL_DAIO_PORT_MASK_DIGITAL_D1`
`XL_DAIO_PORT_MASK_DIGITAL_D2`
`XL_DAIO_PORT_MASK_DIGITAL_D3`
`XL_DAIO_PORT_MASK_DIGITAL_D4`
`XL_DAIO_PORT_MASK_DIGITAL_D5`
`XL_DAIO_PORT_MASK_DIGITAL_D6`
`XL_DAIO_PORT_MASK_DIGITAL_D7`

➔ **type**
For use with `XL_DAIO_TRIGGER_TYPE_PORT`.
`XL_DAIO_TRIGGER_TYPE_RISING`
`XL_DAIO_TRIGGER_TYPE_FALLING`
`XL_DAIO_TRIGGER_TYPE_BOTH`

Example:

```
XLstatus            xlStatus;
XLportHandle        portHandle = ...;
XLaccess            mask = ...;
XLdaioTriggerMode   xlDaioTmAna;

memset(&xlDaioTmAna, 0x00, sizeof(xlDaioTmAna));
xlDaioTmAna.triggerType      = XL_DAIO_TRIGGER_TYPE_CYCLIC;
xlDaioTmAna.portTypeMask     = XL_DAIO_PORT_TYPE_MASK_ANALOG;
xlDaioTmAna.param.cycleTime  = 50000; // in us
xlStatus = xlIoSetTriggerMode(portHandle, mask, &xlDaioTmAna);
```

Example:

```
XLstatus            xlStatus;
XLportHandle        portHandle = ...;
XLaccess            mask = ...;
XLdaioTriggerMode   xlDaioTmDig;

memset(&xlDaioTmDig, 0x00, sizeof(xlDaioTmDig));
xlDaioTmDig.triggerType = XL_DAIO_TRIGGER_TYPE_PORT;

xlDaioTmDig.portTypeMask
 = XL_DAIO_PORT_TYPE_MASK_DIGITAL;

xlDaioTmDig.param.digital.portMask
= XL_DAIO_PORT_MASK_DIGITAL_D4 | XL_DAIO_PORT_MASK_DIGITAL_D5;

xlDaioTmDig.param.digital.type = XL_DAIO_TRIGGER_TYPE_BOTH;
xlStatus = xlIoSetTriggerMode(portHandle, mask, &xlDaioTmDig);
```

3.7.2 xlIoConfigurePorts

Syntax

```
XLstatus xlIoConfigurePorts (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  XLdaioSetPort *pxlDaioSetPort)
```

Description

Configures the DAIO ports.

Note: This command can be called only once.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **pxlDaioTriggerMode**
  Use this structure to configure the port (see below).

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLdaioSetPort

```
struct xl_daio_set_port{
      unsigned int portType;
      unsigned int portMask;
      unsigned int portFunction[8];
      unsigned int reserved[8];
} XLdaioSetPort;
```

Parameters

- **portType**
  `XL_DAIO_PORT_TYPE_MASK_ANALOG`
  `XL_DAIO_PORT_TYPE_MASK_DIGITAL`
- **portMask**
  Specifies the digital port (D0…D07):
  `XL_DAIO_PORT_MASK_DIGITAL_D0`
  `XL_DAIO_PORT_MASK_DIGITAL_D1`
  `XL_DAIO_PORT_MASK_DIGITAL_D2`
  `XL_DAIO_PORT_MASK_DIGITAL_D3`
  `XL_DAIO_PORT_MASK_DIGITAL_D4`
  `XL_DAIO_PORT_MASK_DIGITAL_D5`
  `XL_DAIO_PORT_MASK_DIGITAL_D6`
  `XL_DAIO_PORT_MASK_DIGITAL_D7`

  Specifies the analog port (A0…A3):
  `XL_DAIO_PORT_MASK_ANALOG_A0`
  `XL_DAIO_PORT_MASK_ANALOG_A1`
  `XL_DAIO_PORT_MASK_ANALOG_A2`
  `XL_DAIO_PORT_MASK_ANALOG_A3`
- **portFunction**
  For digital ports:
  `XL_DAIO_PORT_DIGITAL_OPENDRAIN`

XL_DAIO_PORT_DIGITAL_PUSHPULL
XL_DAIO_PORT_DIGITAL_IN

For analog ports:
XL_DAIO_PORT_ANALOG_IN
XL_DAIO_PORT_ANALOG_OUT
XL_DAIO_PORT_ANALOG_DIFF
XL_DAIO_PORT_ANALOG_OFF

XL_DAIO_PORT_ANALOG_IN and XL_DAIO_PORT_ANALOG_OUT can be defined at the same time.

➔ **reserved**
Set to 0.

Example:

```
XLstatus            xlStatus;
XLportHandle        portHandle = ...;
XLaccess            mask = ...;
XLdaioSetPort       confDaioPortsDig;

memset(&confDaioPortsDig, 0x00, sizeof(confDaioPortsDig));
confDaioPortsDig.portType = XL_DAIO_PORT_TYPE_MASK_DIGITAL;
confDaioPortsDig.portMask = (XL_DAIO_PORT_MASK_DIGITAL_D0 |
                             XL_DAIO_PORT_MASK_DIGITAL_D1 |
                             XL_DAIO_PORT_MASK_DIGITAL_D2 |
                             XL_DAIO_PORT_MASK_DIGITAL_D3 |
                             XL_DAIO_PORT_MASK_DIGITAL_D4 |
                             XL_DAIO_PORT_MASK_DIGITAL_D5 |
                             XL_DAIO_PORT_MASK_DIGITAL_D6 |
                             XL_DAIO_PORT_MASK_DIGITAL_D7);

confDaioPortsDig.portFunction[0] =
XL_DAIO_PORT_DIGITAL_PUSHPULL;

confDaioPortsDig.portFunction[1] =
XL_DAIO_PORT_DIGITAL_PUSHPULL;

confDaioPortsDig.portFunction[2] =
XL_DAIO_PORT_DIGITAL_OPENDRAIN;

confDaioPortsDig.portFunction[3] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[4] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[5] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[6] = XL_DAIO_PORT_DIGITAL_IN;
confDaioPortsDig.portFunction[7] = XL_DAIO_PORT_DIGITAL_IN;

xlStatus = xlIoConfigurePorts(portHandle, mask,
&confDaioPortsDig);
```

3.7.3 xlIoSetDigInThreshold

Syntax

```
XLstatus xlIoSetDigInThreshold (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    level)
```

Description

Defines the voltage level for logical high and logical low (digital input).

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 124 for further details.

3.7.4 xlIoSetDigOutLevel

Syntax

```
XLstatus xlIoSetDigOutLevel(
  XLportHandle    portHandle,
  XLaccess        accessMask,
  unsigned int    level)
```

Description

Defines the voltage level for logical high (digital output).

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **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 124 for further details.

3.7.5 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 must contain the mask of channels to be accessed.
- **pxlDaioDigitalParams**
  Use this structure to set the value of the digital out pin (see below).

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLdaioDigitalParams

```
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`

3.7.6 xlIoSetAnalogOutput

Syntax

```
XLstatus xlIoSetAnalogOutput (
  XLportHandle    portHandle,
  XLaccess        accessMask,
  XLdaioDigitalParams* pxlDaioAnalogParams)
```

Description

Configures the analog output.

Input parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels to be accessed.
- **pxlDaioAnalogParams**
  Use this structure to set the value of the analog out pin (see below).

Return value

Returns an error code.
See section Error Codes on page 124 for further details.

XLdaioAnalog-Params

```
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).

3.7.7 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 must contain the mask of channels to be accessed.
- **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 124 for further details.

4 Event Structures

In this chapter you find the following information:

4.1 Basic Events

4.1.1 XL Event

Syntax

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

Parameters

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

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

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

4.1.2 XL Tag Data

Syntax

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

Parameters

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

4.2 CAN Event

4.2.1 XL CAN Message

Description

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

Syntax

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

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 chapter XL Event, **tag** on page 88.

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.

4.3 Chip State Event

4.3.1 XL Chip State

Syntax

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

Tag

`XL_CHIP_STATE` (see chapter XL Event, **tag** on page 88).

Description

This event occurs after calling `xlCanRequestChipState`.

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.

4.4 Timer Events

4.4.1 Timer

Tag `XL_TIMER` (see chapter XL Event, **tag** on page 88)

Description A timer event can be generated cyclically by the driver to keep the application alive. The timer event occurs after init of the timer with `xlSetTimerRate`.

4.5 LIN Events

4.5.1 LIN Message API

Syntax

```
union  s_xl_lin_msg_api {
  struct s_xl_lin_msg          linMsg;
  struct s_xl_lin_no_ans       linNoAns;
  struct s_xl_lin_wake_up      linWakeUp;
  struct s_xl_lin_sleep        linSleep;
  struct s_xl_lin_crc_info     linCRCinfo;
};
```

Parameters

- **linMsg**
  Structure for the LIN messages.
- **linNoAns**
  Structure for the LIN message that gets no answer.
- **linWakeUp**
  Structure for the wake events.
- **linSleep**
  Structure for the sleep events.
- **linCRCino**
  Structure for the CRC info events.

4.5.2 LIN Message

Syntax

```
struct s_xl_lin_msg {
  unsigned char  id;
  unsigned char  dlc;
  unsigned short flags;
  unsigned char  data[8];
  unsigned char  crc;
};
```

Tag

`XL_LIN_MSG` (see chapter XL Event, **tag** on page 88)

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.

4.5.3 LIN Error Message

Tag `XL_LIN_ERRMSG` (see chapter XL Event, **tag** on page 88)

4.5.4 LIN Sync Error

Tag `XL_LIN_SYNC_ERR` (see chapter XL Event, **tag** on page 88)

Description Notifies an error in analyzing the sync field.

4.5.5 LIN No Answer

Syntax

```
struct s_lin_NoAns {
  unsigned char id;
}
```

Tag `XL_LIN_NOANS` (see chapter XL Event, **tag** on page 88)

Description If a LIN **master** request gets no **slave** response a `linNoAns` event is received.

Parameters

- **id**
  The LIN ID on which was the master request.

4.5.6 LIN Wake Up

Syntax

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

Tag `XL_LIN_WAKEUP` (see chapter XL Event, **tag** on page 88)

Description When a channel wakes up (comes out of the sleep mode) a `linWakeUp` event is received.

Parameters

- **flag**
  If the wake-up signal comes from the internal hardware, the flag is set to `XL_LIN_WAKUP_INTERNAL` otherwise it is not set (external wake-up).

4.5.7 LIN Sleep

Syntax

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

Tag `XL_LIN_SLEEP` (see chapter XL Event, **tag** on page 88)

Description For this event there can be different reasons:

- After `xlActivatechannel` a linSleep event is received (only for a LIN application).

- After `xlLinWakeUp` (e.g. an internal wake-up).
- After receiving a LIN message the master goes back into sleep mode.

Parameters

- **flag**
  The flags describe if the hardware comes from the sleep-mode or is set into the sleep mode.
  `XL_LIN_SET_SLEEPMODE`
  The hardware is set into sleep-mode.

  `XL_LIN_COMESFROM_SLEEPMODE`
  The hardware wakes up.

  `XL_LIN_STAYALIVE`
  There is no change in the hardware state.

4.5.8 LIN CRC Info

Syntax

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

Tag

`XL_LIN_CRCINFO` (see chapter XL Event, **tag** on page 88)

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.

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.

4.6 Sync Pulse Events

4.6.1 Sync Pulse

Syntax

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

Tag

`XL_SYNC_PULSE` (see chapter XL Event, **tag** on page 88).

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.

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.

4.7 DAIO Events for CANcab

4.7.1 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 chapter XL Event, **tag** on page 88)

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 timestamp in this event (in order to get real time of measurement). In order to get real time of measurement substract this value from event's timestamp. 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.

4.8 DAIO Events for VN1630A/VN1640A/IOpiggy

4.8.1 DAIO Piggy Data

Syntax

```
struct s_xl_daio_piggy_data {
  unsigned int daioEvtTag;
  unsigned int triggerType;

  union {
    XL_IO_DIGITAL_DATA  digital;
    XL_IO_ANALOG_DATA   analog;
  } data;
};
```

Description

The event is fired as configured via `xlIoSetTriggerMode` (for VN1630A/VN1640A see section xlIoSetTriggerMode on page 77, for IOpiggy see section xlIoSetTriggerMode on page 79). 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**
  See section IO Digital Data on page 101 and section IO Analog Data on page 101.

Example: Checking digital port D0

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

4.8.2 IO Digital Data

Syntax

```
typedef struct s_xl_io_digital_data {
  unsigned int digitalInputData;
} XL_IO_DIGITAL_DATA;
```

Parameters

- **digitalInputData**
  Contains the data of port 0 ..7. It is independent of the port function.

4.8.3 IO Analog Data

Syntax

```
typedef struct s_xl_io_analog_data {
  unsigned int measuredAnalogData0;
  unsigned int measuredAnalogData1;
  unsigned int measuredAnalogData2;
  unsigned int measuredAnalogData3;
} XL_IO_ANALOG_DATA;
```

Parameters

- **measuredAnalogData0**
  First analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData1**
  Second analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData0**
  Third analog port that is defined as an input.
  This value is 0 for differential input.
- **measuredAnalogData0**
  Fourth analog port that is defined as an input.
  This value is 0 for differential input.

4.9 Transceiver Events

4.9.1 Transceiver

Syntax

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

Tag

`XL_TRANSCEIVER` (see chapter XL Event, **tag** on page 88)

Parameters

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

5 CAN FD Event Structures

In this chapter you find the following information:

5.1 Tx Event

5.1.1 CAN FD Event

Description

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

XLcanTxEvent

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

Parameters

- **tag**
  Event type.
- **transId**
  Internal use.
- **channelIndex**
  Channel index of the hardware (see section xlGetChannelIndex on page 30).
- **reserved.**
  Internal use.
- **tagData**
  Tag Data. See section CAN FD Tag Data Tx Message on page 105 for further information.

5.1.2 CAN FD Tag Data Tx Message

Tag data

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

Parameters

- **canId**
  CAN ID (11 or 29 bits).
- **msgFlags**
  `XL_CAN_TXMSG_FLAG_EDL`
  Extended data length.

  `XL_CAN_TXMSG_FLAG_BRS`
  Baudrate switch.

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

  `XL_CAN_TXMSG_FLAG_WAKEUP`
  Generates a wakeup message.
- **dlc**
  Data length code.

Format	Number of Data Bytes	DLC3	DLC2	DLC1	DLC0	DLC
CAN/CAN FD	0	0	0	0	0	0
CAN/CAN FD	1	0	0	0	1	1
CAN/CAN FD	2	0	0	1	0	2
CAN/CAN FD	3	0	0	1	1	3
CAN/CAN FD	4	0	1	0	0	4
CAN/CAN FD	5	0	1	0	1	5
CAN/CAN FD	6	0	1	1	0	6
CAN/CAN FD	7	0	1	1	1	7
CAN/CAN FD	8	1	0	0	0	8
CAN FD	12	1	0	0	1	9
CAN FD	16	1	0	1	0	10
CAN FD	20	1	0	1	1	11
CAN FD	24	1	1	0	0	12
CAN FD	32	1	1	0	1	13
CAN FD	48	1	1	1	0	14
CAN FD	64	1	1	1	1	15

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

5.2 Rx Event

5.2.1 CAN FD Event

Description

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

XLcanRxEvent

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

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

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_STATISTIC`
  `XL_CAN_EV_TAG_CHIP_STATE`
- **channelIndex**
  Channel index of the hardware (see section xlGetChannelIndex on page 30).
- **reserved**
  Internal use.
- **userHandle**
  Internal use.
- **flagsChip**
  Queue overflow (upper 8bit), `XL_CAN_QUEUE_OVERFLOW`.
- **reserved0**
  Internal use.
- **reserved1**
  Internal use.
- **timeStamp**
  Timestamp which is synchronized by the driver.

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

5.2.2 CAN FD Tag Data Rx Message

Tag data RX message

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

Parameters

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

  `XL_CAN_RXMSG_FLAG_BRS`
  Baud rate switch.

  `XL_CAN_RXMSG_FLAG_ESI`
  Error state indicator.

  `XL_CAN_RXMSG_FLAG_EF`
  Error frame.

  `XL_CAN_RXMSG_FLAG_ARB_LOST`
  Arbitration lost.
- **crc**
  Crc of the CAN message.
- **totalBitCnt**
  Number of received bits including stuff bit.
- **dlc**
  Data length code.

Format	Number of Data Bytes	DLC3	DLC2	DLC1	DLC0	DLC
CAN/CAN FD	0	0	0	0	0	0
CAN/CAN FD	1	0	0	0	1	1
CAN/CAN FD	2	0	0	1	0	2
CAN/CAN FD	3	0	0	1	1	3
CAN/CAN FD	4	0	1	0	0	4
CAN/CAN FD	5	0	1	0	1	5
CAN/CAN FD	6	0	1	1	0	6
CAN/CAN FD	7	0	1	1	1	7
CAN/CAN FD	8	1	0	0	0	8
CAN FD	12	1	0	0	1	9
CAN FD	16	1	0	1	0	10
CAN FD	20	1	0	1	1	11

CAN FD	24	1	1	0	0	12
CAN FD	32	1	1	0	1	13
CAN FD	48	1	1	1	0	14
CAN FD	64	1	1	1	1	15

- **reserved**
  Internal use.
- **data**
  Data that was received.

5.2.3 CAN FD Tag Data Tx Request

Tag data
TX request

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

Parameters

- **canId**
  CAN ID.
- **msgFlags**
  `XL_CAN_RXMSG_FLAG_EDL`
  Extended data length.
  `XL_CAN_RXMSG_FLAG_BRS`
  Baud rate switch.
  `XL_CAN_RXMSG_FLAG_ESI`
  Error state indicator.
  `XL_CAN_RXMSG_FLAG_EF`
  Error frame.
  `XL_CAN_RXMSG_FLAG_ARB_LOST`
  Arbitration lost.
- **dlc**
  Data length code.

Format	Number of Data Bytes	DLC3	DLC2	DLC1	DLC0	DLC
CAN/CAN FD	0	0	0	0	0	0
CAN/CAN FD	1	0	0	0	1	1
CAN/CAN FD	2	0	0	1	0	2
CAN/CAN FD	3	0	0	1	1	3
CAN/CAN FD	4	0	1	0	0	4
CAN/CAN FD	5	0	1	0	1	5
CAN/CAN FD	6	0	1	1	0	6
CAN/CAN FD	7	0	1	1	1	7
CAN/CAN FD	8	1	0	0	0	8
CAN FD	12	1	0	0	1	9

CAN FD	16	1	0	1	0	10
CAN FD	20	1	0	1	1	11
CAN FD	24	1	1	0	0	12
CAN FD	32	1	1	0	1	13
CAN FD	48	1	1	1	0	14
CAN FD	64	1	1	1	1	15

- **txAttemptConf**
  Reserved.
- **reserved**
  Internal use.
- **data**
  Data that was receive.

5.2.4 CAN FD Tag Data Chip State

Tag data chip state

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

Parameters

- **busStatus**
  Returns the state of the CAN controller. The following codes are possible:
  `XL_CHIPSTAT_BUSOFF`
  The bus is offline.

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

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

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

5.2.5 CAN FD Tag Data Event Error

Tag data event error

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

Parameters

- **errorCode**
  `XL_CAN_ERRC_NACK_ERROR`
- **reserved**
  Internal use.

5.2.6 CAN FD Tag Data Sync Pulse

Tag data sync pulse

```
typedef XL_SYNC_PULSE_EV XL_CAN_EV_SYNC_PULSE;

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

Parameters

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

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

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

6 Examples

In this chapter you find the following information:

6.1 Overview

Available examples

In order to show the functionality of the XL Family Driver Library, a couple of examples are included:

- **xlCANdemo**
  Demonstrates the CAN implementation.
- **xlCANcontrol**
  An example GUI applicaton for CAN.
- **xlLINExample**
  Shows how to setup a LIN master/slave.
- **xlDAIOexamples**
  Detailed example for IOcab 8444opto.
- **xlDAIOdemo**
  Demo program for the IOcab 8444opto.
- **.NET examples**
  See `XL Driver Library - .NET Wrapper Description.pdf` for detailed information.

Note: To run and compile the examples in VS2013, the MFC libraries for multibyte character encoding has to be installed (MFC MBCS DLL Add-on). The download is available on the Microsoft website.

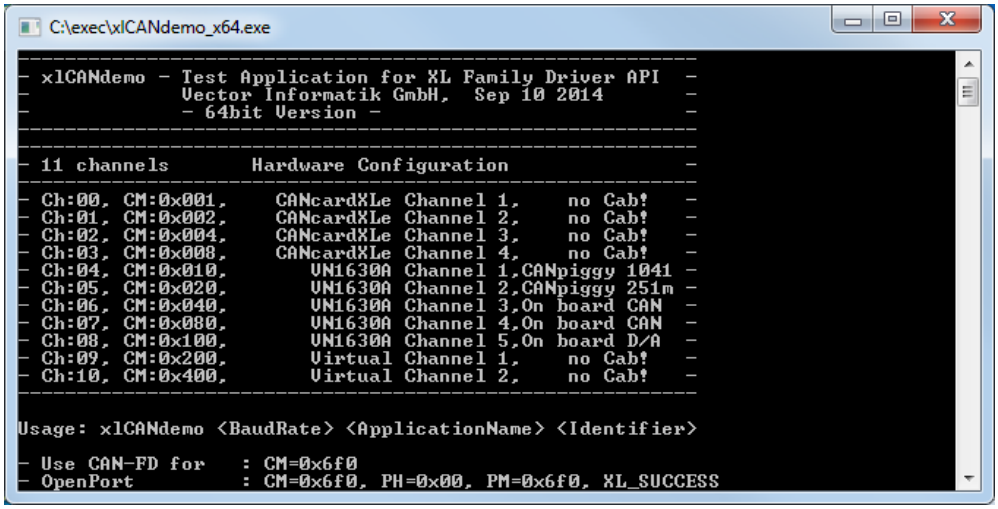
Caution: THE INCLUDED EXAMPLES ARE PROVIDED “AS-IS”. NO LIABILITY OR RESPONSIBILITY FOR ANY ERRORS OR DAMAGES.

6.2 xlCANdemo

Description

xlCANdemo is the replacement for the old CANdemo. It shows the basic handling in a CAN and CAN FD applications. The program contains a command line interface:

xlCANdemo <Baudrate> <ApplicationName> <Identifier>



Keyboard commands

The running application can be controlled by a few keyboard commands:

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

Source code

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

Function

```
demoInitDriver()
```

Function Description

This function opens the driver and reads the actual hardware configuration. (`xlGetHardwareConfig`). A valid `channelMask` is calculated (we use only channels with CANcabs or CANpiggy's) and **one** port is opened afterwards.

Function	`demoCreateRxThread()`
Function Description	In order to read the driver message queue a thread is generated.

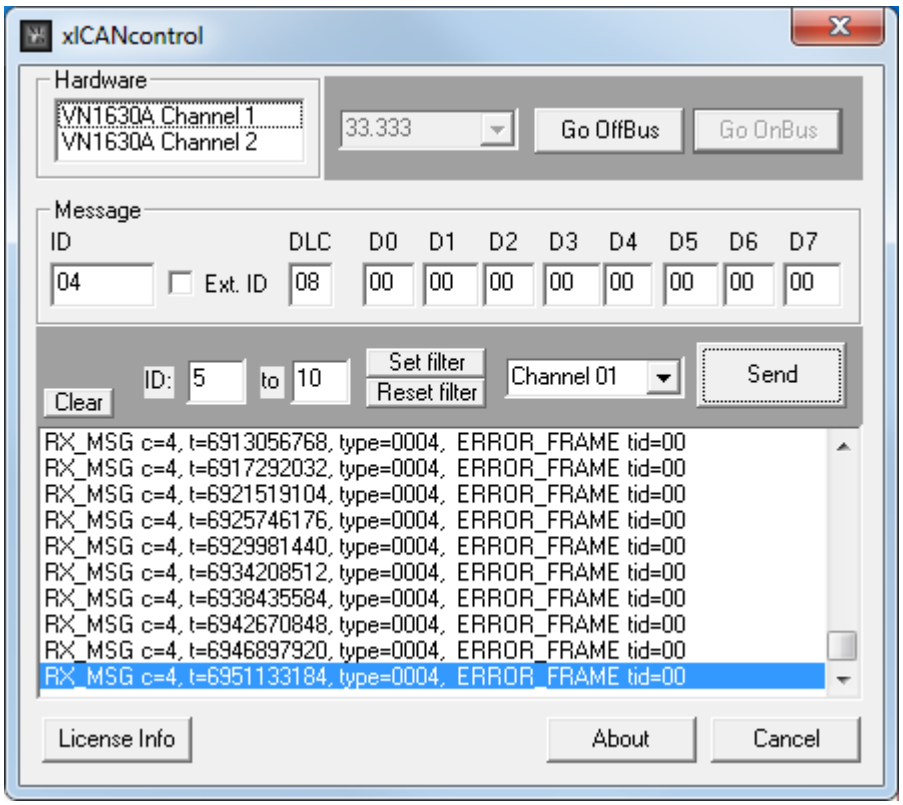
6.3 xlCANcontrol

Description

This Visual Studio project **xlCANcontrol** shows the basic CAN handling with the XL Driver Library and a simple graphical user interface. The application needs two CANcabs/CANpiggies to run. The program searches a Vector device on the first start, which supports CAN and assigns two channels within **Vector Hardware Config** (which can surely be changed to other device channels). The found device is 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, setup 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.



Class overview

The example has the following class structure:

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

Function	`CANInit`
Function Description	This function is called on application start to get the valid `channelmasks` (access masks). Afterwards one port is opened for the two channels and a thread is created to readout the message queue is started.
Function	`CANGoOnBus`
Function Description	After pressing the **[Go OnBus]** button, the CAN parameters are set and both channels are activated.
Function	`CANGoOffBus`
Function Description	After pressing the [Go OffBus] button, the channels will be deactivated.
Function	`CANSend`
Function Description	Transmits the CAN message with `xlCANtransmit`.
Function	`CANResetFilter`
Function Description	Resets (open) the acceptance filter.
Function	`CANSetFilter`
Function Description	Sets the acceptance filter range. It is needed to close the acceptance filter for every ID before.
Function	`canGetChannelMask`
Function Description	This function looks for assigned channels in **Vector Hardware Conf** with `xlGetApplConfig`. If there is no application registered, `xlCANcontrol` searchs for available CAN channels and assigns them in **Vector Hardware Conf** with `xlSetApplConfig`. The function fails, if there are no valid channels found.
Function	`canInit`
Function Description	Opens **one** port with **both** channels (`xlOpenPort`).
Function	`canCreateRxThread`
Function Description	In order to readout the driver message queue, the application uses a thread

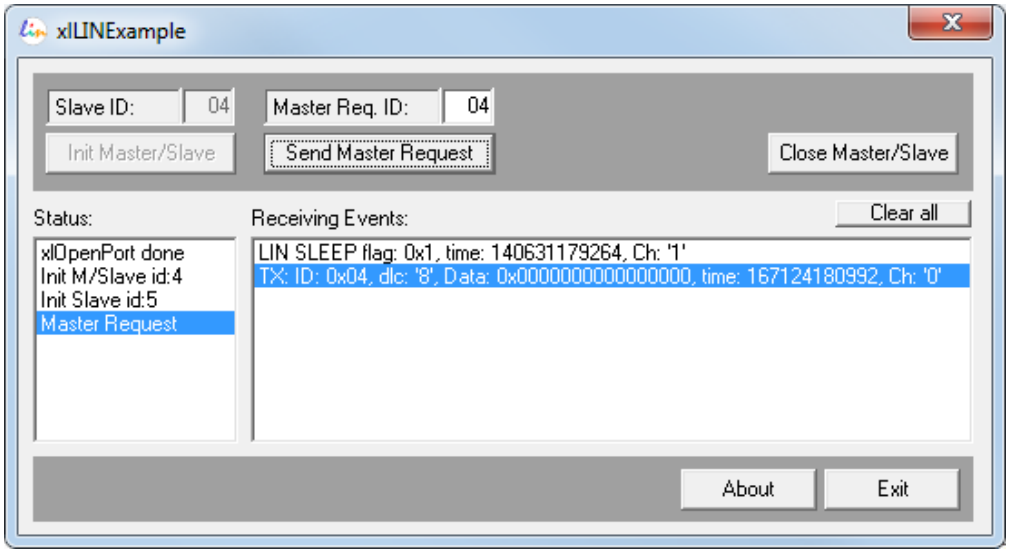
(`RxThread`). An event is created and set up with `xlSetNotification` to notify the thread.

6.4 xlLINExample

Description

xlLINExample is a Microsoft Visual C++ project that 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 definition can be made within the Vector Hardware Configuration tool. If xlLINExample 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.



Class overview

The xlLINExample 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

Function

`LINGetDevice`

Function Description

In order to get the channel mask, use `linGetChannelMask` to read all hardware parameters. `xlGetApplConfig` checks whether the application has already been assigned. If not, a new entry with `xlSetApplConfig` is created.

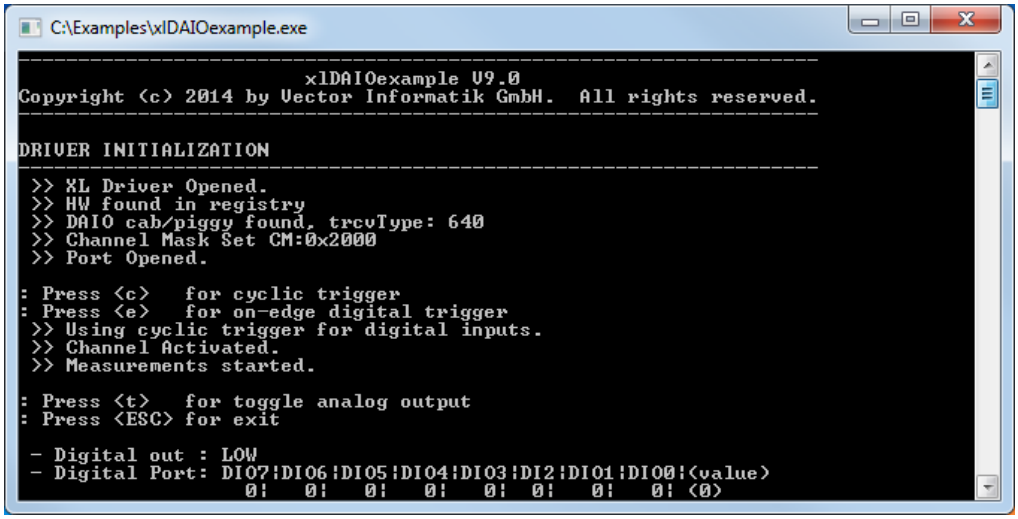
Function	`LINInit`
Function Description	`LINInit` opens one port for one channel, or if available two channels (CH1 and CH2). The first channel will be initialized as LIN master including a LIN slave (id=4) the other a LIN slave (id=5). After a successfully `xlOpenPort`, a RX thread is created. Use `xlLinSetChannelParams` 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`.
Function	`linInitMaster`
Function Description	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'.
Function	`linInitSlave`
Function Description	Use `xlLinSetSlave` to set up slave. Before you go 'onBus' it is needed to define the LIN slave ID that cannot be changed after `xlActivateChanne`. All other parameters like the `data` values or the **`DLC`** can be varied.
Function	`LINSendMasterReq`
Function Description	After the LIN network is specified and the master/slaves are 'onBus', the master can transmit master requests with `xlLinSendRequest`.
Function	`LINClose`
Function Description	When all is done, the port is closed with `xlClosePort`.

6.5 xlDAIOexample

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.

Note: This example also works with a VN1630 as well as a VN89xx with an IOpiggy. The related IO pin assignments are described in the according manuals.

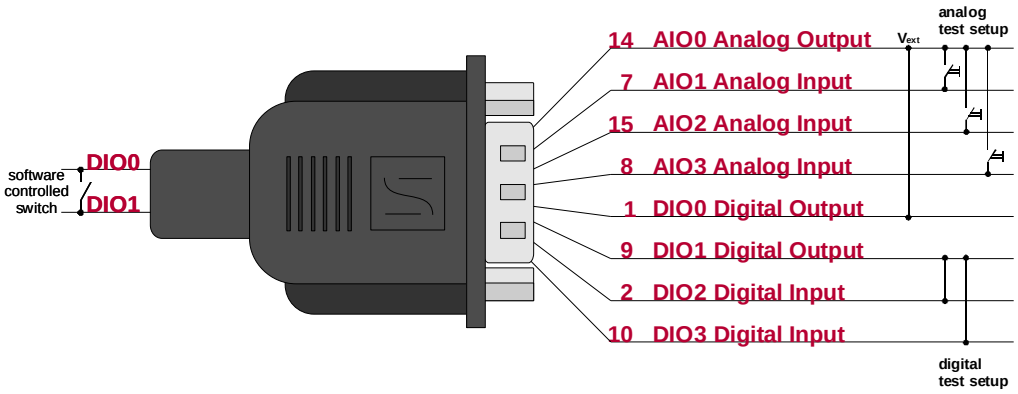


Pin definitions

The following pins of the IOcab 8444opto are used in this example:

- AIO0 (pin 14): Analog output.
- AIO1 (pin 7): Analog input.
- AIO2 (pin 15): Analog input.
- AIO3 (pin 8): Analog input.
- DIO0 (pin 1): Digital output (shared electronic switch with DIO1).
- DIO1 (pin 9): Digital output (supplied by DIO0, when switch is closed).
- DIO2 (pin 2): Digital input.
- DIO3 (pin 10): Digital input.

Setup



Info: 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.

Keyboard commands

When the application is running, there is a couple of keyboard commands:

Key	Command
ENTER	**Toggle digital output.**
x	**Closes application.**

Example: Display output of xlDAIOexample.

```
AIO0                : 4032mV
AIO1                : 0mV
AIO2                : 0mV
AIO3                : 0mV
Switch selected     : DIO0/DI01
Switch states       : OPEN
Digital Port        : DIO7 DIO6 DIO5 DIO4 DIO3 DIO2 DIO1 DIO0 val
                        0    0    0    0    0    0    0    1 (1)
```

Explanation

- ➔ “AIO0” displays 4032mV, since it is set to output with maximum output level.
- ➔ “AIO1” displays 0mV, since there is no applied voltage at this input.
- ➔ “AIO2” displays 0mV, since there is no applied voltage at this input.
- ➔ “AIO3” displays 0mV, since there is no applied voltage at this input.
- ➔ ”Switch selected” displays DIO0/DIO1 (first switch)
- ➔ ”Switch states” displays the state of switch between DIO0/DIO1
- ➔ “Digital Port” shows the single states of DIO7…DIO0:
  - DIO0: displays ‘1’ (always ‘1’, due the voltage supply)
  - DIO1: displays ‘0’ (switch is open, so voltage at DIO0 is not passed through)
  - DIO2: displays ‘0’ (output of DIO1)
  - DIO3: displays ‘0’ (output of DIO1)
  - DIO4: displays ‘0’ (n.c.)
  - DIO5: displays ‘0’ (n.c.)
  - DIO6: displays ‘0’ (n.c.)
  - DIO7: displays ‘0’ (n.c.)

Example: Display output of xlDAIOexample.

```
AIO0                : 4032mV
AIO1                : 0mV
AIO2                : 4032mV
AIO3                : 0mV
Switch selected     : DIO0/DI01
Switch state        : CLOSED
Digital Port        : DIO7 DIO6 DIO5 DIO4 DIO3 DIO2 DIO1 DIO0 val
                        0    0    0    0    1    1    1    1 (f)
```

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.)

Info: 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’.

Source code

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

Function

```
InitIOcab
```

Function Description

This function opens the driver and reads the current hardware configuration. (`xlGetHardwareConfig`). A valid `channelMask` is calculated and **one** port is opened afterwards.

Function

```
ToggleSwitch
```

Function Description

This function toggles all switches and passes through the applied voltage at DIO0 to DIO1.

Function

```
CloseExample
```

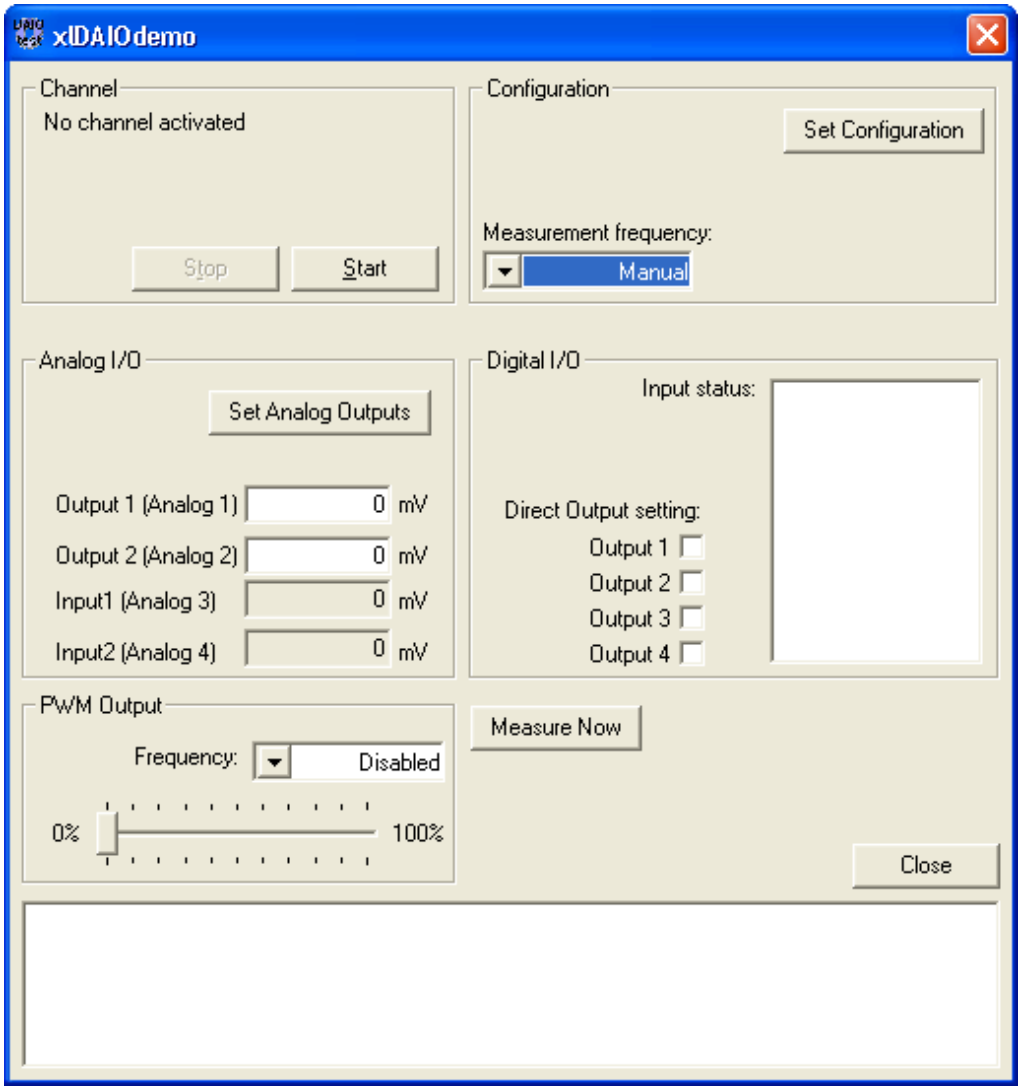
Function Description

Closes the driver and the application.

6.6 xlDAIOdemo

Description

In order to see the configuration of a digital/analog IO application, a Visual Studio Project, called ‘xlDAIOdemo’, is included in the XL API setup. To run the application, one connected IOcab 8444opto is needed.



Class overview

The xlDIAOExample has the following class structure:

- **CXlDAIOdemoApp**
  Main MFC class ⇒ xlDAIOdemo.cpp
- **CXlDAIOdemoDlg**
  Handles the window dialog messages and control the IOcab ⇒ xlDAIOdemoDlg.cpp
- **ReceiveThread**
  Thread to handle the DAIO events.

7 Error Codes

In this chapter you find the following information:

7.1 Error Code Table

XLStatus error codes

In this section all error codes are described which may be returned by a driver call.

Code	Error	Description
0	XL_SUCCESS	The driver call was successful.
10	XL_ERR_QUEUE_IS_EMPTY	The receive queue of the port is empty. The user can proceed normally.
11	XL_ERR_QUEUE_IS_FULL	The transmit queue of a channel is full. The transmit event will be lost.
12	XL_ERR_TX_NOT_POSSIBLE	The hardware is busy and not able to transmit an event at once.
14	XL_ERR_NO_LICENSE	Only used in the MOST option to differ between the free- and 'MOST Analyses' library.
101	XL_ERR_WRONG_PARAMETER	At least one parameter passed to the driver was wrong or invalid.
111	XL_ERR_INVALID_CHAN_INDEX	The driver attempted to access a channel with an invalid index.
112	XL_ERR_INVALID_ACCESS	The user made a call to a port specifying channel(s) for which he had not declared access at opening of the port.
113	XL_ERR_PORT_IS_OFFLINE	The user called a port function whose execution must be online, but the port is offline.
116	XL_ERR_CHAN_IS_ONLINE	The user called a function whose desired channels must be offline, but at least one channel is online.
117	XL_ERR_NOT_IMPLEMENTED	The user called a feature which is not implemented.
118	XL_ERR_INVALID_PORT	The driver attempted to access a port by an invalid pointer or index.
120	XL_ERR_HW_NOT_READY	The accessed hardware is not ready.
121	XL_ERR_CMD_TIMEOUT	The timeout condition occurred while waiting for the response event of a command.
129	XL_ERR_HW_NOT_PRESENT	The hardware is not present (or could not be found) at a channel. This may occur with removable hardware or faulty hardware.
158	XL_ERR_INIT_ACCESS_MISSING	Function call requires init access.
201	XL_ERR_CANNOT_OPEN_DRIVER	The attempt to load or open the driver failed. Reason could be the driver file which cannot be found, is already loaded or part of a previously unloaded driver.
202	XL_ERR_WRONG_BUS_TYPE	The user called a function with the wrong bus type. (e. g. try to activate a LIN channel for CAN).
203	XL_ERR_DLL_NOT_FOUND	The XL API dll could not be found.
204	XL_ERR_INVALID_CHANNEL_MASK	Invalid channel mask.

Code	Error	Description
205	XL_ERR_NOT_SUPPORTED	Function call not supported.
255	XL_ERROR	An unspecified error occurred.

8 Migration Guide

In this chapter you find the following information:

8.1 Overview

Migration from CAN Driver to XL Driver Library

In order to update or migrate applications, which are based on the CAN Driver library to the XL Driver Library have a look on the following table:

8.1.1 Bus Independent Function Calls

No changes

The following functions have the same calling convention:

Old Bus independent function calls	XL Bus independent function calls
ncdOpenDriver	xlOpenDriver
ncdCloseDriver	xlCloseDriver
ncdGetChannelIndex	xlGetChannelIndex
ncdGetChannelMask	xlGetChannelMask
ncdSetTimerRate	xlSetTimerRate
ncdResetClock	xlResetClock
ncdFlushReceiveQueue	xlFlushReceiveQueue
ncdGetReceiveQueueLevel	xlGetReceiveQueueLevel
ncdGetErrorString	xlGetErrorString
ncdDeactivateChannel	xlDeactivateChannel
ncdClosePort	xlClosePort

Changes

The following functions have not the same calling convention:

Old Bus independent function calls	XL Bus independent function calls
ncdGetDriverConfig	xlGetDriverConfig
ncdOpenPort	xlOpenPort
ncdActivateChannel	xlActivateChannel
ncdReceive1/ncdReceive	xlReceive
ncdGetApplConfig	xlGetApplConfig
ncdSetApplConfig	xlSetApplConfig
ncdGetEventString	xlGetEventString
n.a.	xlGetSyncTime
n.a.	xlGenerateSyncPulse
n.a.	xlPopupHwConfig
ncdGetState	removed

8.1.2 CAN Dependent Function Calls

No changes

The following functions have the same calling convention:

Old CAN functions	XL CAN functions
ncdSetChannelOutput	xlCanSetChannelOutput
ncdSetChannelMode	xlCanSetChannelMode
ncdSetReceiveMode	xlCanSetReceiveMode
ncdSetChannelTransceiver	xlCanSetChannelTransceiver
ncdSetChannelParams	xlCanSetChannelParams
ncdSetChannelParamsC200	xlCanSetChannelParamsC200
ncdSetChannelBitrate	xlCanSetChannelBitrate
ncdSetChannelAcceptance	xlCanSetChannelAcceptance
ncdAddAcceptanceRange	xlCanAddAcceptanceRange
ncdRemoveAcceptanceRange	xlCanRemoveAcceptanceRange
ncdResetAcceptance	xlCanResetAcceptance
ncdRequestChipState	xlCanRequestChipState
ncdFlushTransmitQueue	xlCanFlushTransmitQueue
ncdSetChannelAcceptance	xlCanSetChannelAcceptance
ncdTransmit	xlCanTransmit

Changes

The following functions have not the same calling convention:

Old CAN functions	XL CAN functions
ncdSetChannelAcceptance	xlCanSetChannelAcceptance
ncdTransmit	xlCanTransmit

8.1.3 LIN Dependent Function Calls

New LIN functions

The following functions have been added:

CAN Library	XLDriver Library
n.a.	xlLinSetChannelParams
n.a.	xlLinSetDLC
n.a.	xlLinSetSlave
n.a.	xlLinSetSleepMode
n.a.	xlLinWakeUp
n.a.	xlLinSendRequest
n.a.	xlLinSetSlave
n.a.	xlDAIOSetMeasurementFrequency
n.a.	xlDAIOSetAnalogParameters
n.a.	xlDAIOSetAnalogOutput
n.a.	xlDAIOSetAnalogTrigger
n.a.	xlDAIOSetDigitalParameters
n.a.	xlDAIOSetDigitalOutput
n.a.	xlDAIOSetPWMOutput
n.a.	xlDAIORequestMeasurement

8.2 Changed Calling Conventions

New conventions

New calling conventions in the XL Driver Library:

Function name	Changes
xlGetApplConfig	→ Parameter changed from **int** to **unsigned int**. → Bus type parameter added (`XL_BUSTYPE_CAN` e.g.)
xlSetApplConfig	→ Parameter changed from **int** to **unsigned int**. → Bus type parameter added (`XL_BUSTYPE_CAN` e.g.)
xlGetDriverConfig	→ Structure for return value changed. (It is not needed to malloc/alloc the structure size any more depending on the founded channels).
xlOpenPort	→ Init Mask value removed ⇒ Now it is passed in the 'permissionMask' → Interface version flag added → Bus type parameter added. → CAN: All acceptance filter are open!
xlSetNotification	→ Notification data type changed from 'unsigned long' to a windows handle (To avoid the type casts). → Now the function returns the event handle so it is not necessary to create an event before.
xlActivateChannel	→ Bus type parameter added. → Additional flags (e.g. to reset the clock after activating the channel)
xlReceive	→ Receive event structure changed. → Event counter added.
xlGetEventString	→ Event type changed.
xlCanSetChannelAcceptance	→ No structure for the code/mask needed any more. → The ID range can be changed with a separate flag.
xlCanTransmit	→ Message event type changed. → Possible to transmit more messages with one function call.

Get more Information!

Visit our Website for:

> News

> Products

> Demo Software

> Support

> Training Classes

> Addresses

www.vector.com