Manual

XL Driver Library

API Description

Version 6.7

English

Imprint

Vector Informatik GmbH
Ingersheimer Straße 24
D-70499 Stuttgart

Table of contents

1 Introduction

In this chapter you find the following information:

1.1 About this user manual

1.1.1 Access helps and conventions

To find information quickly

The user manual provides you the following access helps:

- At the beginning of each chapter you will find a summary of the contents,
- In the header you can see in which chapter and paragraph you are ((situated)),
- In the footer you can see to which version the user manual replies,
- At the end of the user manual you will find an index, with whose help you will quickly find information,
- Also at the end of the user manual on page 11 you will find a glossary in which you can look up an explanation of used technical terms.

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.

1.1.2 Certification

Certified Quality Management System

Vector Informatik GmbH has ISO 9001:2000 certification. The ISO standard is a globally recognized standard.

1.1.3 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.

1.1.4 Support

You need support?

You can get through to our support at the phone number

+49 711 80670-200 or by fax

+49 711 80670-555

E-Mail: support@vector-informatik.de

1.1.5 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 2000 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's XL interfaces like CANcardXL, CANcaseXL, CANcaseXL log, CANboardXL, CANboardXL PCIe, CANboardXL pxi, CANcardX, VN26X0 and VN3X00.

Info: The library does not support CANAC2 PCI, CANAC2 ISA and CANpari. For CANcardX there is no LIN or digital/analog I/O support.

XL Driver Library

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

- Win2000
- WinXP

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

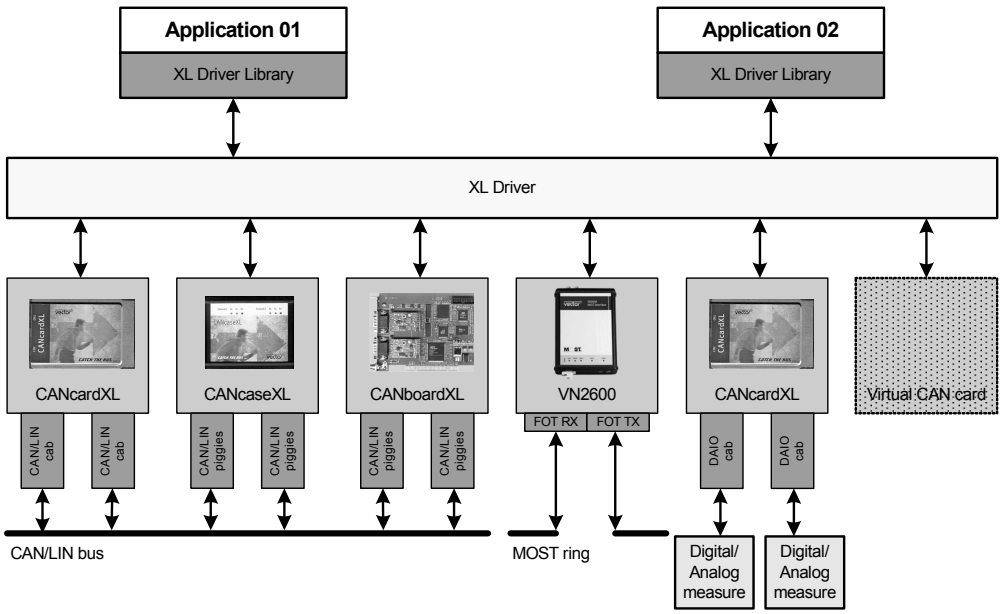


Figure 1: Possible applications with the XL Driver.

Hardware installation

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

Multi application

The driver is designed for Windows 2000/XP 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, FlexRay or DAIO, a channel can only be used by one application at the same time.

Principle structure for CAN applications

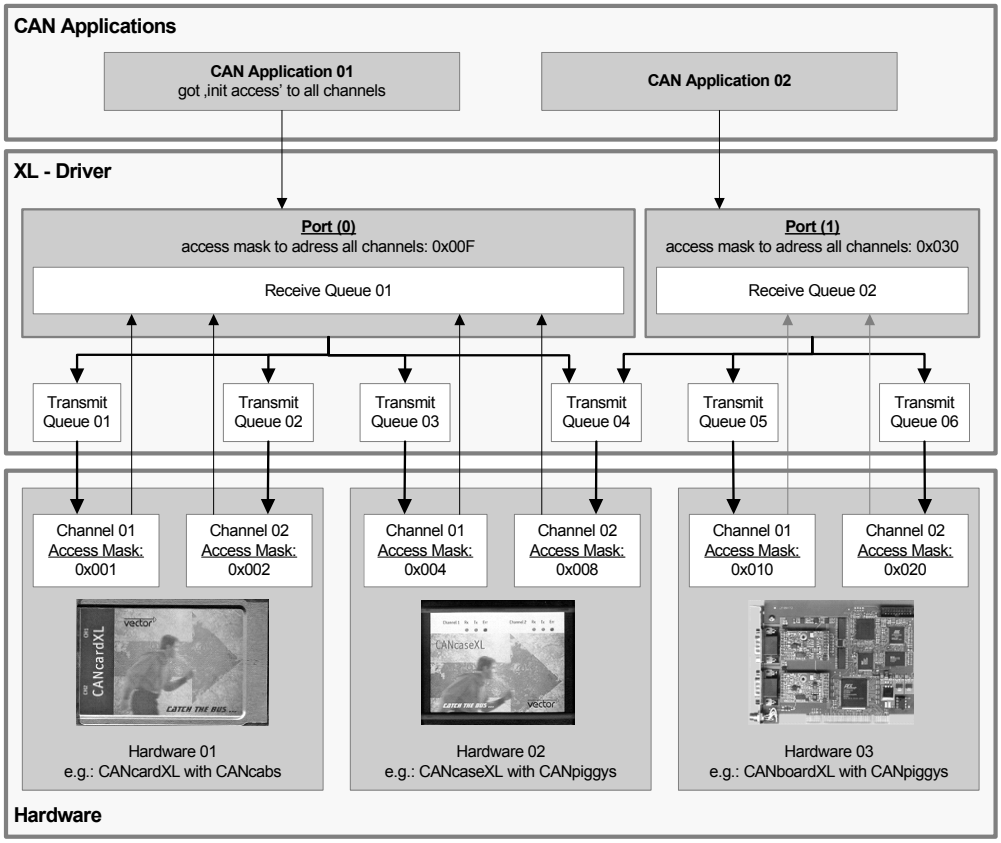


Figure 2: Accessing XL Interfaces

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 14.

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 15.

MOST

The MOST implementation supports currently 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 a 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 mulit-application functionality. For further information see chapter: “General information - Multiapplication support”

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 16.

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 connected to the port. Afterwards the application calls the driver. 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.

When a hardware channel receives a message, it is passed 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 allows only 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 framework 1.1 and framework 2.0 in order to use the XL API in any .NET language. See `XL Driver Library - .NET Wrapper Description.pdf` for detailed information.

Files

File name	Description
vxlapi.dll	32 bit DLL for Windows 2000 and XP
vxlapi.h	C header
vxlapi_NET11.dll	.NET1.1 wrapper
vxlapi_NET11.xml	Wrapper documentation, used by IntelliSense function
vxlapi_NET20.dll	.NET2.0 wrapper
vxlapi_NET20.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 the file `xlLoadlib.cpp` in 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.

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 flags** 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 application. To view the debug prints the freeware tool **DebugView** from http://www.sysinternals.com (now Microsoft) can be used.

Vector Hardware Config

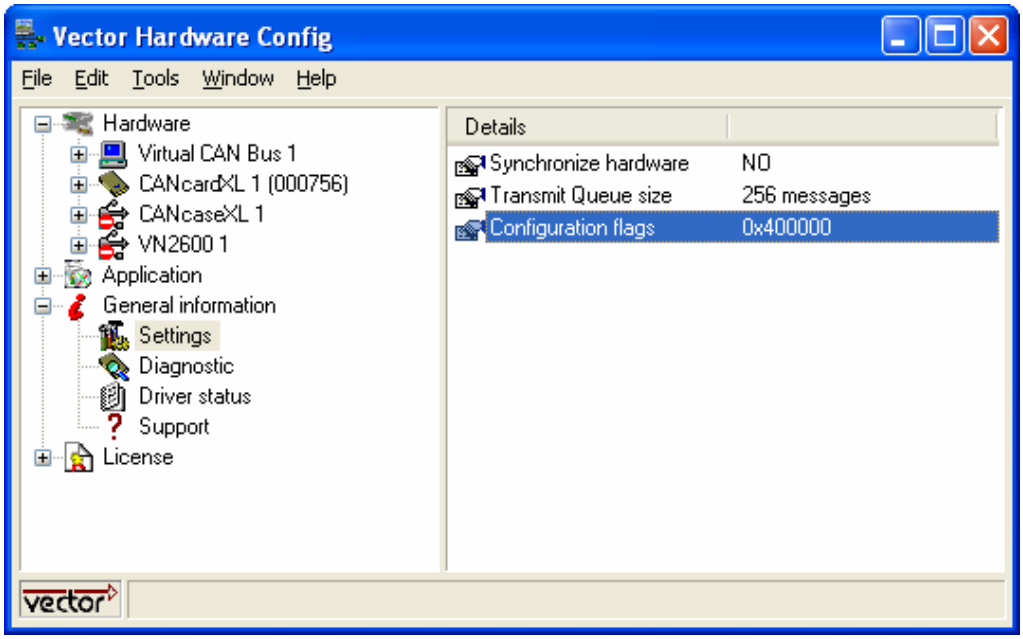


Figure 3: Hardware configuration

2.3 Flowcharts

2.3.1 CAN application

Calling sequence

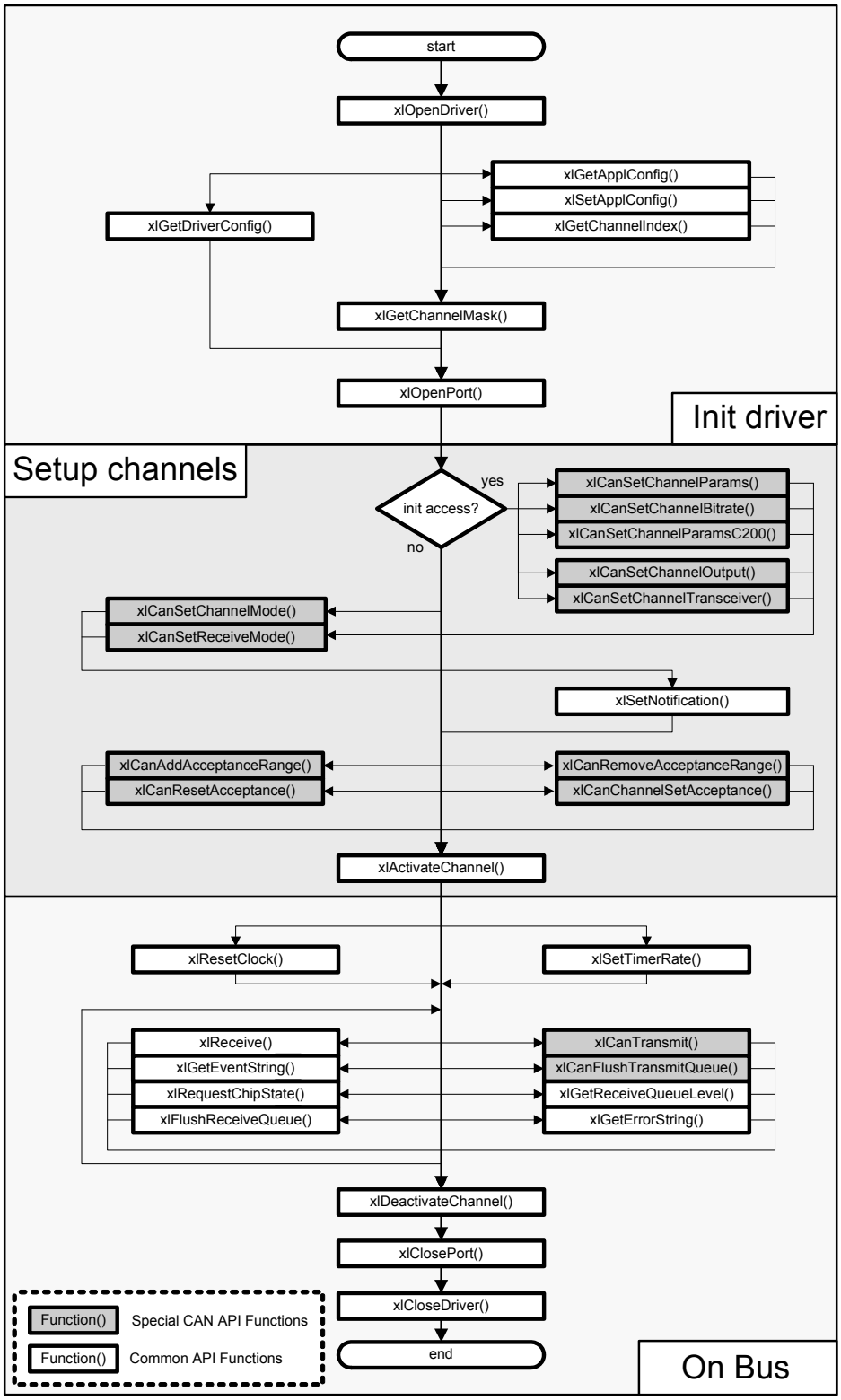


Figure 4: Function calls for CAN applications

2.3.2 LIN application

Calling sequence

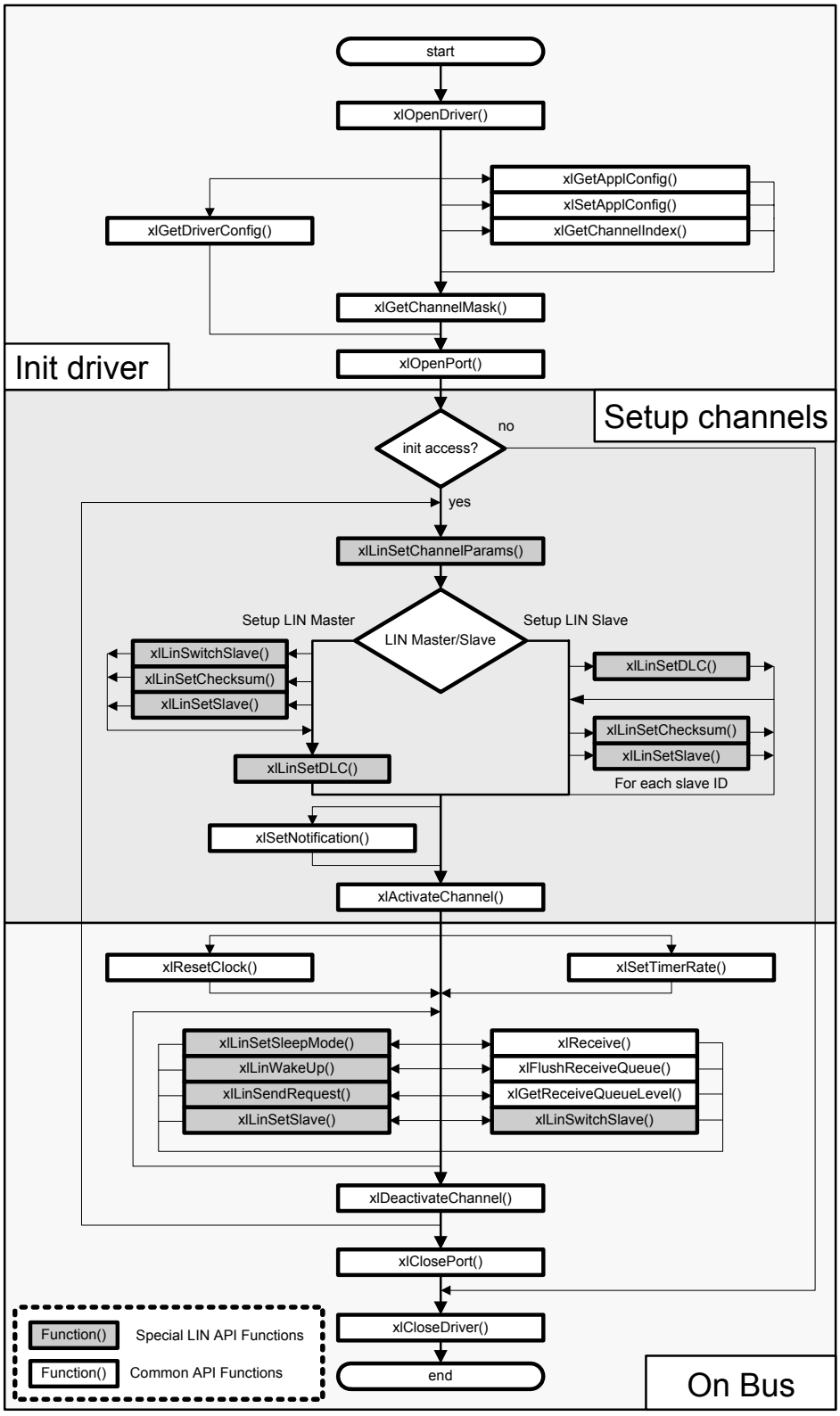


Figure 5: Function calls for LIN applications

2.3.3 DAIO application

Calling sequence

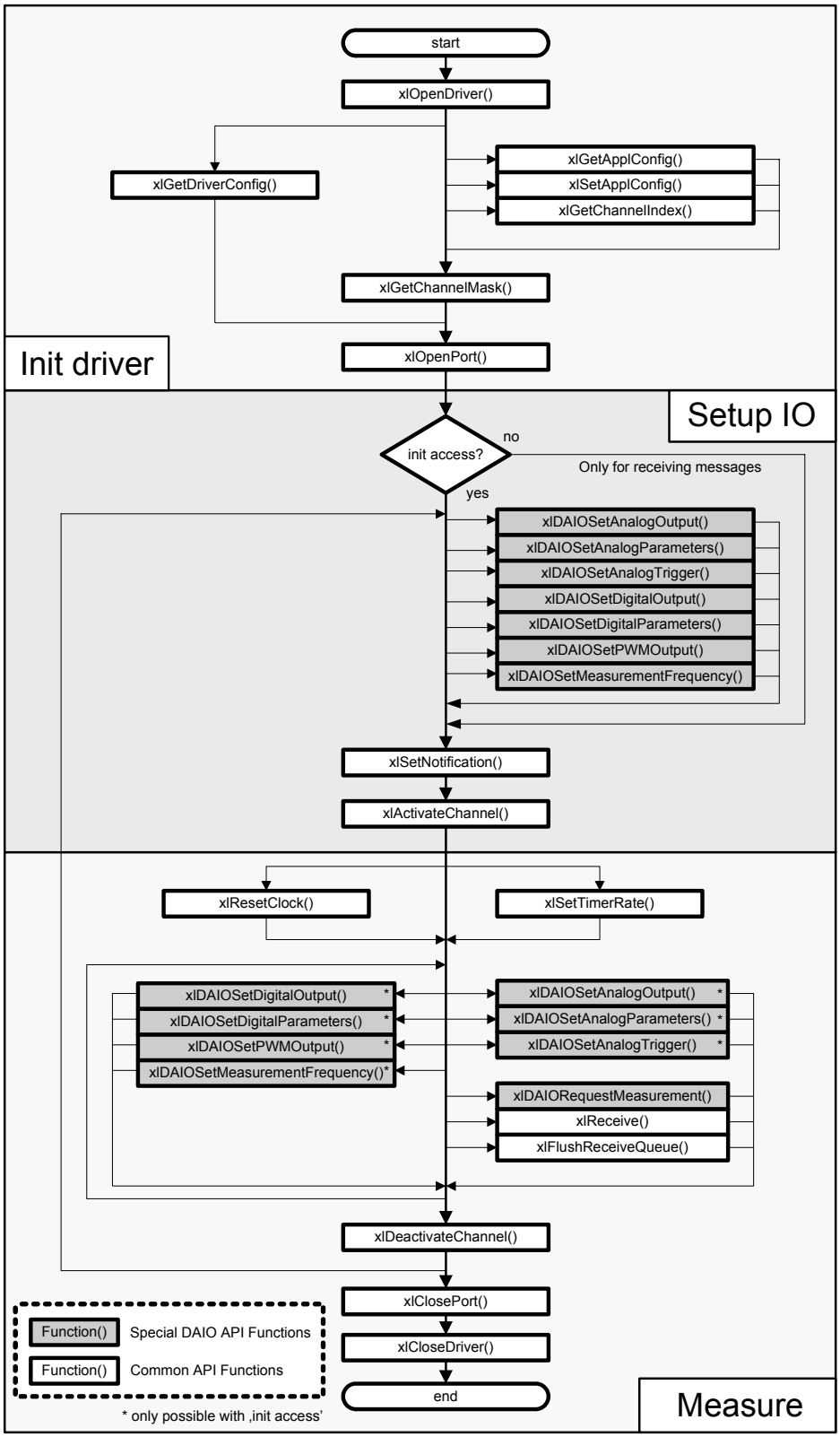


Figure 6: Function calls for DAIO applications

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.
Zero means success. See section Error codes on page 81 for further details.

3.1.2 xlCloseDriver

Syntax

```
XLstatus xlCloseDriver(void)
```

Description

This function closes the driver.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 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 done in the **Vector Hardware Configuration** tool.

Input Parameters

- **appName**
  Name of application to be read. Application names can be found in the Vector Hardware Configuration tool.
- **appChannel**
  Application channel (0,1, …) to be used. An application can offer several channels, which are assigned to physical channels (e.g. “CANdemo CAN1” to CANcardXL Channel 1, “CANdemo CAN2” to CANcardXL Channel 2). This assignment has to be done in Vector Hardware Config.
- **busType**
  Specifying 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.
Zero means success. See page 81 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 of an exiting application.

Input Parameters

- **appName**
  Name of 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`
- **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.
Zero means success. See section Error codes on page 81 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`. After each call the result describes the current state of the driver configuration.

Input Parameters

- **XLdriverConfig**
  Points to a user buffer for the information, which the driver returns. See details below for further information.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 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 field for future use.
- **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 of same hardware types (0, 1, ...),
  e.g. CANcardXL
  - 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 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

  On this channel there is a connected cab or piggy that 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.

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 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.

3.1.7 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.

3.1.8 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 (like 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 the channels, which gets **init access**.

Input Parameters

- **userName**
  The name of the application that is shown 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
  Size of the port receive queue allocated by the driver in bytes.
- **xlInterfaceVersion**
  Current API version,
  e.g.
  - use `XL_INTERFACE_VERSION` to activate the XL interface (CAN, LIN, DAIO).
  - use `XL_INTERFACE_VERSION_V4` for MOST.
- **busType**
  Bus type that should be activated,
  e.g.
  - use `XL_BUS_TYPE_LIN` to initialize LIN
  - use `XL_BUS_TYPE_CAN` to initialize CAN
  - use `XL_BUS_TYPE_DAIO` to initialize DAIO
  - use `XL_BUS_TYPE_MOST` to initialize MOST
  - use `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 is the **init access** 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`.
Zero means success. See section Error codes on page 81 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 together: $\sum wanted\_access\_masks$

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	0000**1**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);
```

3.1.9 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.
Zero means success. See section Error codes on page 81 for further details.

3.1.10 xlSetTimerRate

Syntax

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

Description

This call sets up the rate for the port's cyclic timer events. The resolution is 1000µs. The minimum and maximum `timerRate` values depend on the hardware. If a value is outside of the allowable range the limit value is used.

Info: Timer events will only be generated if no other event occurred during the timer interval and might be dropped if other events occur.

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.
Zero means success. See section Error codes on page 81 for further details.

3.1.11 xlResetClock

Syntax

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

Description

Resets the time stamps for the specified port.

Input Parameters

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

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.1.12 xlSetNotification

Syntax

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

Description

The function returns the notification handle to notify the application if there are messages within 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.
Zero means success. See section Error codes on page 81 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);
```

3.1.13 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.
Zero means success. See section Error codes on page 81 for further details.

3.1.14 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 a long, where the actual count of events in the receive queue is returned.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.1.15 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). From now 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.
- use `XL_BUS_TYPE_LIN` to initialize LIN
- use `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.
Zero means success. See section Error codes on page 81 for further details.

Example: Channel Activation

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

3.1.16 xlReceive

Syntax

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

Description

Reads the received events out of the message queue. An application should read all available messages to be sure to re-enable the event.

Input Parameters

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

Input/ Output Parameters

→ **pEventCount**
Pointer to event counter. On input the variable must be set to the size (in messages) of received buffer. On output the variable contains the number of received messaged.

→ **pEventList**
Pointer to 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.
Zero means success. See section Error codes on page 81 for further details.

Example: Read each message out of 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);
      }
    }
  }
```

3.1.17 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.

3.1.18 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 81 for further details.

Return Value

Error code as plain text string.

3.1.19 xlGetSyncTime

Syntax

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

Description

Current high precision PC time comparable with the synchronized time stamps (1ns resolution)

Input Parameters

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

Output Parameters

→ **time**
Points to variable, where the sync time is received.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.1.20 xlGenerateSyncPulse

Syntax

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

Description

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

Input Parameters

- **portHandle**
  The port handle retrieved by `xlOpenPort`.
- **accessMask**
  The access mask must contain the mask of channels on which the sync pulse shall be generated.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.1.21 xlPopupHwConfig

Syntax

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

Description

Call this function to popup the Vector Hardware Config tool.

Input Parameters

- **callSign**
  Reserved type.
- **wairForFinish**
  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.
Zero means success. See section Error codes on page 81 for further details.

3.1.22 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 activated 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.
Zero means success. See section Error codes on page 81 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 an acknowledge 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 the function is not called.

Info: To call this function the port must have **init access (**see `xlOpenPort`) to 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: From driver version V5.5 the silent mode is changed. Now the TX pin is switched off. (The 'SJA1000 silent mode' is no more used).

  - `XL_OUTPUT_MODE_NORMAL`
  Acknowledge (normal mode)

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.2.2 xlCanSetChannelMode

Syntax

```
Xlstatus xlCanSetChannelMode (
   XLportHandle  portHandle,
   XLaccess      accessMask,
   unsigned      long   tx,
   unsigned      long   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 defaults are 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.
Zero means success. See section Error codes on page 81 for further details.

3.2.3 xlCanSetReceiveMode

Syntax

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

Description

Suppress' error frames and chipstate events with '1' and allows them with '0'. Default is to allow error frames and chipstate events.

Input Parameters

- **Port**
  The port handle retrieved by `xlOpenPort`.
- **ErrorFrame**
  Suppress error frames.
- **ChipState**
  Suppress chipstate events.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.2.4 xlCanSetChannelTransceiver

Syntax

```
XLstatus xlCanSetChannelTransceiver(
  XLportHandle  portHandle,
  XLaccess      accessMask,
  int           type,
  int           lineMod
  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`

➔ **lineMod**
- Lowspeed (252/1053/1054)
`XL_TRANSCEIVER_LINEMODE_SLEEP`
Put CANcab into sleep mode

`XL_TRANSCEIVER_LINEMODE_NORMAL`
Enable normal operation

- Highspeed (1041 and 1041opto)
`XL_TRANSCEIVER_LINEMODE_SLEEP`
Put CANcab into sleep mode

`XL_TRANSCEIVER_LINEMODE_NORMAL`
Enable normal operation

- Single Wire (AU5790)
`XL_TRANSCEIVER_LINEMODE_NORMAL`
Enable normal operation

`XL_TRANSCEIVER_LINEMODE_SWC_SLEEP`
Switch to sleep mode

`XL_TRANSCEIVER_LINEMODE_SWC_NORMAL`
Switch to normal operation

`XL_TRANSCEIVER_LINEMODE_SWC_FAST`
Switch transceiver to fast mode

➔ **resNet**
- Lowspeed (252/1053/1054)
`XL_TRANSCEIVER_RESNET_NA`

- Highspeed (1041 and 1041opto)
`XL_TRANSCEIVER_RESNET_NA`

- Single Wire (AU5790)
`XL_TRANSCEIVER_RESNET_NA`

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 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.
Zero means success. See section Error codes on page 81 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 sam. 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.
Zero means success. See section Error codes on page 81 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.
Zero means success. See section Error codes on page 81 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 can not implement the filter, the driver virtualizes filtering.

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

Info: As the default the acceptance filters are open after a xlOpenPort.

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`
  Means the filter is set for standard message IDs.

  - `XL_CAN_EXT`
  Means the filter is set for extended message IDs

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 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_EXT);
```

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

By default the filters are opened (all messages are received). `xlCanAddAcceptanceRange` 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 can not implement the filter, the driver virtualizes filtering.

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

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.
Zero means success. See section Error codes on page 81 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 can not implement the filter, the driver virtualizes filtering.

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

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.
Zero means success. See section Error codes on page 81 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.
Zero means success. See section Error codes on page 81 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.
Zero means success. See section Error codes on page 81 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, which should be transmit and returns the number of transmitted messages.
- **pMessages**
  Points to user buffer with messages to be transmit. At least the buffer must have the size of `messageCount`.

Return Value

Returns an error code.
Zero means success. `XL_ERR_QUEUE_IS_FULL` means the channel‘s transmit-queue is full. See section Error codes on page 81 for further details.

Example: Transmit 100 CAN messages with the ID = 4

```
XLevent xlEvent[100];
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.
Zero means success. See section Error codes on page 81 for further details.

3.3 LIN commands

3.3.1 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. What means 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.
Zero means success. See section Error codes on page 81 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, ...
- **LINVersion**
  - `XL_LIN_VERSION_1_3`
  Use LIN 1.3 protocol

  - `XL_LIN_VERSION_2_0`
  Use LIN 2.0 protocol
- **reserved**
  For future use.

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);
```

3.3.2 xlLinSetDLC

Syntax

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

Description

Defines the data length for all requested messages. It is needed for 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.
Zero means success. See section Error codes on page 81 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);
```

3.3.3 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. Here the checksum calculation can be changed 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 can not be changed. Per default always the classic model is set for all IDs. 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.
Zero means success. See section Error codes on page 81 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);
```

3.3.4 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. 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.
Zero means success. See section Error codes on page 81 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);
```

3.3.5 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.
Zero means success. See section Error codes on page 81 for further details.

3.3.6 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 that 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. At the moment set to ‚0’.

Return Value

Returns an error code.
Zero means success. Returns `XL_ERR_INVALID_ACCESS` if it is done on a LIN slave. See section Error codes on page 81 for further details.

3.3.7 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.
Zero means success. See section Error codes on page 81 for further details.

3.3.8 xlLinSetSleepMode

Syntax

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

Description

Activate 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`
  Set hardware into sleep mode (transmits no ‘Sleep-Mode’ frame).

  - `XL_LIN_SET_WAKEUPID`
  Transmits the indicated linID at wakeup and set hardware into sleep mode. It is only possible on a LIN master.
- **linID**
  Defines the linID that is transmited at wake-up.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.4 Digital/analog input/output commands

3.4.1 xlDAIOSetAnalogParameters

Syntax

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

Description

Configures the analog lines. By default all lines are set to input. 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.
Zero means success. See section Error codes on page 81 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)

3.4.2 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 to not 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.
Zero means success. See section Error codes on page 81 for further details.

3.4.3 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.
Zero means success. See section Error codes on page 81 for further details.

3.4.4 xlDAIOSetDigitalParameters

Syntax

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

Description

Configures the digital lines. By default all lines are set to input. 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: `0b0`**`1`**`000000`
- DAIO7: `0b`**`1`**`0000000`

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 needs always two digital lines of the IOcab 8444opto: a general output line and a line for external supply. This means, 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.
Zero means success See section Error codes on page 81 for further details.

3.4.5 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.
Zero means success. See section Error codes on page 81 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
```

3.4.6 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.
Zero means success. See section Error codes on page 81 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)
```

3.4.7 xlDAIOSetMeasurementFrequency

Syntax

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

Description

Sets the measurement frequency. xlEvents will be triggered automatically, which can be received by `xlReceive`. For manual trigger see chapter xlDAIORequestMeasurement 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.
- **measurementInterval**
  Measurement frequency in ms.

Return Value

Returns an error code.
Zero means success. See section Error codes on page 81 for further details.

3.4.8 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.
Zero means success. See section Error codes on page 81 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 short        reserved;
  XLuint64              timeStamp;
  union s_xl_tag_data   tagData;
};
```

Input 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.
- **reserved**
  Reserved for future use.
- **timestamp**
  Actual timestamp generated by the hardware with 8µs resolution.
- **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;
};
```

Input 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

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/XL_TRANSMIT_MSG` (see chapter XL event, **tag** on page 58)

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

  - `XL_CAN_MSG_FLAG_OVERRUN`
  An overrun occurred in the CAN controller

  - `XL_CAN_MSG_FLAG_REMOTE_FRAME`
  The event is a remote frame

  - `XL_CAN_MSG_FLAG_TX_COMPLETED`
  Notification of successful transmission of a message

  - `XL_CAN_MSG_FLAG_TX_REQUEST`
  Notification of request for transmission of a message

  - `XL_CAN_MSG_FLAG_NERR`
  The transceiver reported a error while the message was received.

  - `XL_CAN_MSG_FLAG_WAKEUP`
  high voltage message for Single Wire.
  To flush the queue and transmit a high voltage message
  make an „OR“ combination between the `XL_CAN_MSG_FLAG_WAKEUP` and `XL_CAN_MSG_FLAG_OVERRUN`.
- **dlc**
  The length of a message
- **res1**
  Reserved for future use.
- **data**
  Array containing the data.
- **res2**
  Reserved for future use.

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 58)

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 58)

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 58)

Input 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 58)

4.5.4 LIN sync error

Tag `XL_LIN_SYNC_ERR` (see chapter XL event, **tag** on page 58)

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 58)

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 58)

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 58)

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 58)

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 58)

Description

Input Parameters

➔ **pusleCode**

- `XL_SYNC_PULSE_EXTERNAL`
The sync event comes from an external device

- `XL_SYNC_PULSE_OUR`
The sync pulse event occurs after a `xlGenerateSyncPulse`.

- `XL_SYNC_PULSE_OUR_SHARED`
The sync pulse comes from the same hardware but from another channel.

➔ **time**
Recalculated high resolution card timestamp with 1ns resolution.

4.7 DAIO events

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 58)

Input 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.
- **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.
- **Reserved2**
  Reserved for future use.

4.8 Transceiver events

4.8.1 Transceiver

Syntax

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

Tag

`XL_TRANSCEIVER` (see chapter XL event, **tag** on page 58)

Parameters

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

5 Examples

In this chapter you find the following information:

5.1 Overview

Available examples

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

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

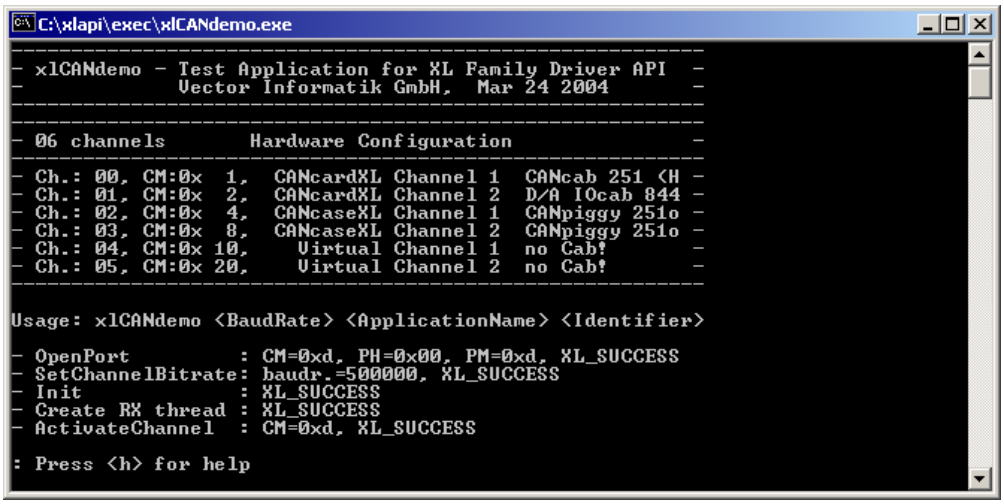
Caution: THE INCLUDED EXAMPLES ARE PROVIDED “AS-IS”. NO LIABILITY OR RESPONSIBILITY FOR ANY ERRORS OR DAMAGES.

5.2 xlCANdemo

Description

xlCANdemo is the replacement for the old CANdemo. It shows the basic handling to get a CAN application running. The program contains a command line interface:

xlCANdemo <Baudrate> <ApplicationName> <Identifier>



Keyboard commands

When the application is running there are couples of 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 out the actual hardware configuration. (`xlGetHardwareConfig`). Afterwards a valid `channelMask` is calculated (we use only channels with CANcabs or CANpiggy's) and **one** port is opened.

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

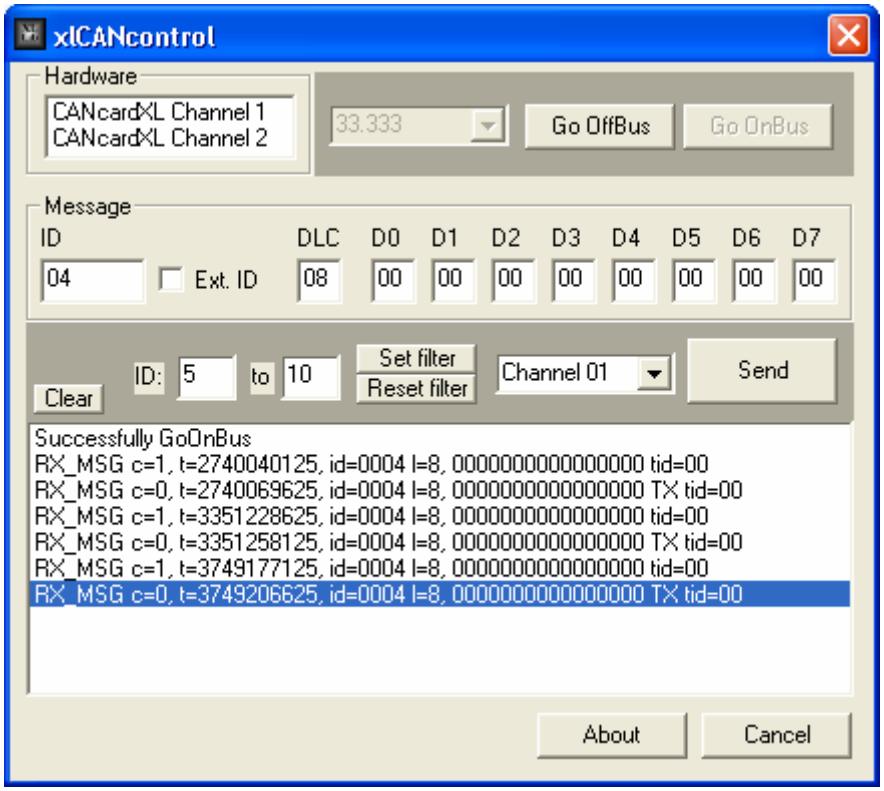
5.3 xlCANcontrol

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 hardware on the first start, which supports CAN and assigns two channels within **Vector Hardware Config** (which can surely change to other channels on another hardware). The found hardware 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 (per default there is a TX complete message from the transmit channel and the received message on the second channel).

During the measurement the acceptance filter range can 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 assign 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

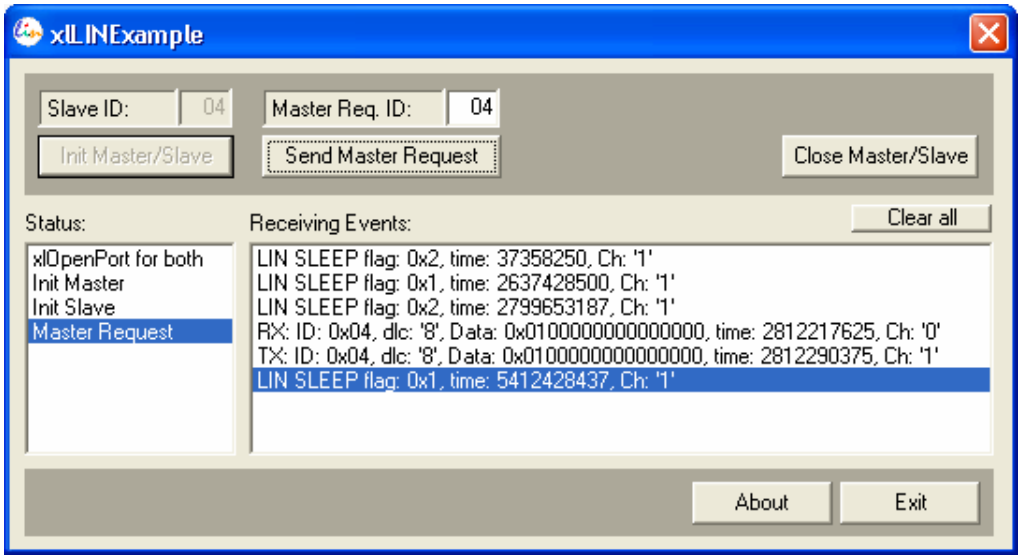
In order to readout the driver message queue the application uses a thread (`RxThread`). An event is created and set up with `xlSetNotification` to notify the thread.

5.4 xlLINExample

Description

xlLINExample is a Microsoft Visual C++ project that demonstrates the basic use of the LIN API. It sets a LIN master on one and a LIN slave to the other channel. The definition can be made within the **Vector Hardware Configuration** tool. If xlLINExample starts the first time it sets CH01 on a CANcardXL to a LIN master and 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

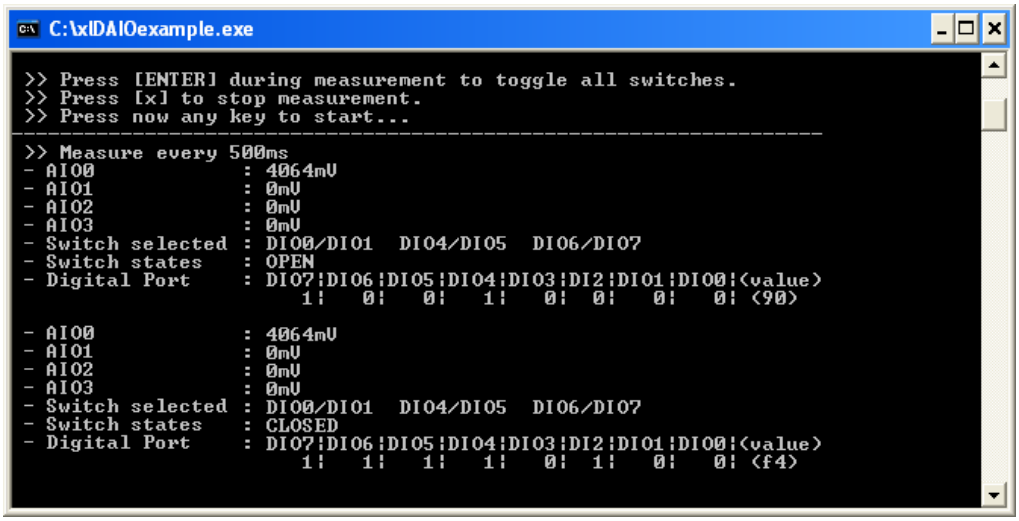
In order to get the channel mask, use `linGetChannelMask` to readout all hardware parameters. `xlGetApplConfig` checks if the application is already assigned. If not it is created a new entry with `xlSetApplConfig`.

Function	`LINInit`
Function Description	`LINInit` opens one port for both channels (CH1 and CH2). Here we use one channel as LIN master and the other one as LIN slave. After a successfully `xlOpenPort` a RX thread is created. Use `xlLinSetChannelParams` in order to initialize the channels (like master/slave and the baud rate).
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 can not 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`.

5.5 xlDAIOexample

Description

This example demonstrates how to set up a single IOcab 8444opto for a test, and how to access the inputs and outputs for cyclically measurement.

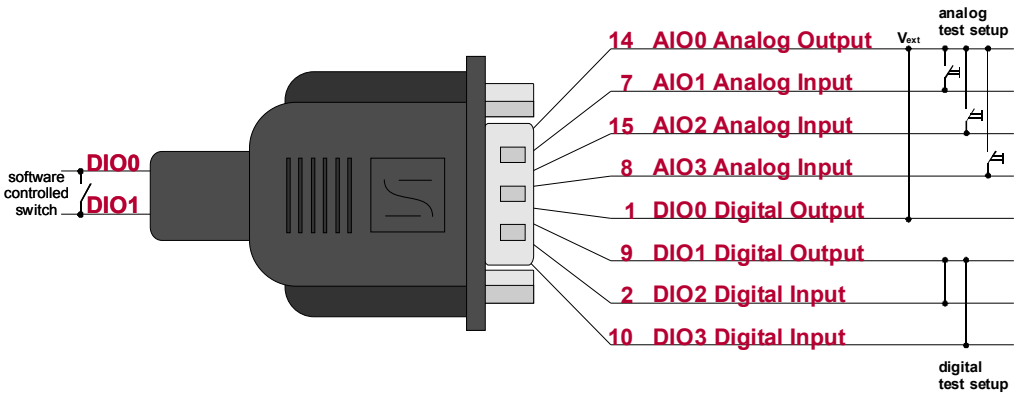


Pin definitions

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 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 out the actual hardware configuration. (`xlGetHardwareConfig`). Afterwards a valid `channelMask` is calculated and **one** port is opened.

Function

```
ToggleSwitch
```

Function Description

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

Function

```
CloseExample
```

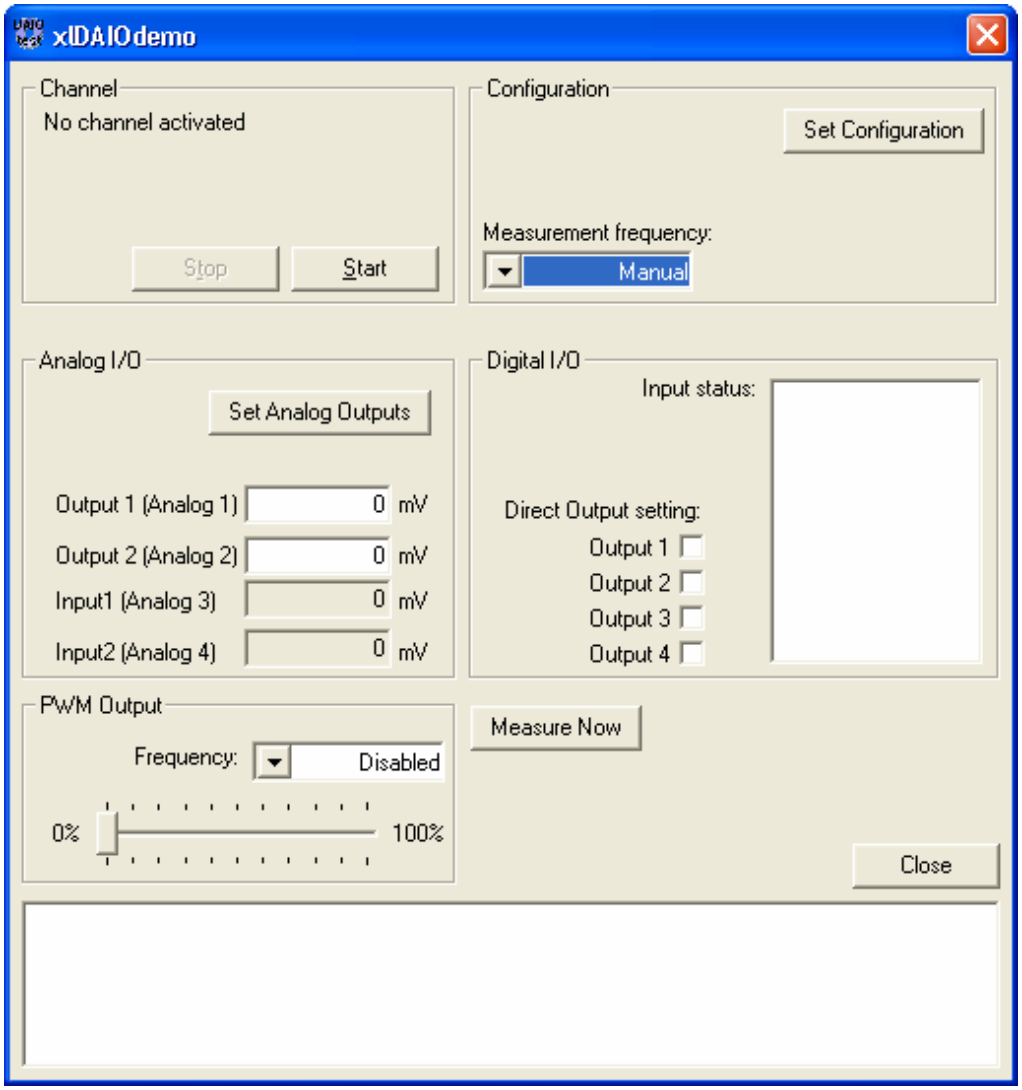
Function Description

Closes the driver and the application.

5.6 xlDAIOdemo

Description

In order to see how to configure and run a digital/analog IO application, a Visual Studio Project called ‘xlDAIOdemo’ is available. To run the application one IOcab 8444opto is needed.



Class overview

The xlDIAOExample has the following class structure:

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

6 Error codes

In this chapter you find the following information:

6.1 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.
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.
201	XL_ERR_CANNOT_OPEN_DRIVER	The attempt to load or open the driver failed. Reason could be the driver file which can not 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).
255	XL_ERROR	An unspecified error occurred.

7 Migration guide

In this chapter you find the following information:

7.1 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:

7.1.1 Bus independent function calls

No changes

The following functions have no changes within the 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 changes within the 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

7.1.2 CAN dependent function calls

No changes

The following functions have no changes within the 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 changes within the calling convention:

Old CAN functions	XL CAN functions
ncdSetChannelAcceptance	xlCanSetChannelAcceptance
ncdTransmit	xlCanTransmit

7.1.3 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

7.2 Changed calling conventions

New conventions

For the following function there is a new calling convention 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.

8 Appendix A: Address table

Vector Informatik GmbH

Vector Informatik GmbH
Ingersheimer Str. 24
D-70499 Stuttgart

Phone: +49 (711) 80670-0
Fax: +49 (711) 80670-111

mailto:info@vector-informatik.de
http://www.vector-informatik.com/

Vector CANtech, Inc.

Vector CANtech, Inc.
Suite 550
39500 Orchard Hill Place
USA-Novi, Mi 48375

Phone: +1 (248) 449 9290
Fax: +1 (248) 449 9704

mailto:info@vector-cantech.com
http://www.vector-cantech.com/

Vector Japan Co., Ltd.

Vector Japan Co., Ltd.
Seafort Square Center Bld. 18F
2-3-12, Higashi-shinagawa, Shinagawa-ku
J-140-0002 Tokyo

Phone: +81 3 (5769) 6970
Fax: +81 3 (5769) 6975

mailto:info@vector-japan.co.jp
http://www.vector-japan.co.jp/

Vector France SAS

Vector France SAS
168, Boulevard Camélinat
F-92240 Malakoff

Phone: +33 (1) 4231 4000
Fax: +33 (1) 4231 4009

mailto:information@vector-france.com
http://www.vector-france.com/

VecScan AB

VecScan AB
Theres Svenssons Gata 9
SE-41755 Göteborg

Phone: +46 (31) 76476-00
Fax: +46 (31) 76476-19

mailto:info@vecscan.com
http://www.vecscan.com/

Vector Korea IT Inc.

Vector Korea IT Inc.
Daerung Post Tower III, 508
182-4 Guro-dong, Guro-gu
Seoul 152-790

REPUBLIC OF KOREA

Phone: +82(0)2 2028 0600
Fax: +82(0)2 2028 0604

mailto:info@vector-korea.com
http://www.vector-korea.com

Get more Information!

Visit our Website for:

> News

> Products

> Demo Software

> Support

> Training Classes

> Addresses

www.vector-worldwide.com