IXXAT®

# VCI: C++

## Software Version 4

## SOFTWARE DESIGN GUIDE

4.02.0250.20022 1.1 ENGLISH

HMS Connecting Devices™

# Important User Information

## Liability

Every care has been taken in the preparation of this document. Please inform HMS Industrial Networks of any inaccuracies or omissions. The data and illustrations found in this document are not binding. We, HMS Industrial Networks, reserve the right to modify our products in line with our policy of continuous product development. The information in this document is subject to change without notice and therefore should not be considered as a binding description of the range of functions (neither for future product versions). HMS Industrial Networks assumes no responsibility for any errors that may appear in this document.

There are many applications of the described product. Those responsible for the use of this device must ensure that all the necessary steps have been taken to verify that the applications meet all performance and safety requirements including any applicable laws, regulations, codes, and standards.

HMS Industrial Networks will under no circumstances assume liability or responsibility for any problems that may arise as a result from improper use or use that is not in accordance with the documented features of this product.

The examples and illustrations in this document are included solely for illustrative purposes.

## Intellectual Property Rights

HMS Industrial Networks has intellectual property rights relating to technology embodied in the product described in this document. These intellectual property rights may include patents and pending patent applications in the USA and other countries.

## Trademark Acknowledgements

IXXAT® is a registered trademark of HMS Industrial Networks. All other trademarks are the property of their respective holders.



VCI: C++ Software Version 4 Software Design Guide
4.02.0250.20022 1.1

This page intentionally left blank

# 1 User Guide

Please read the manual carefully. Make sure you fully understand the manual before using the product.

## 1.1 Document History

| Version | Date | Description |
|---|---|---|
| 1.0 | June 2016 | First version |
| 1.1 | January 2017 | Minor corrections |

## 1.2 Conventions

Instructions and results are shown in the following way:

- ▶ instruction 1
- ▶ instruction 2
  - ▷ result 1
  - ▷ result 2

Lists are shown in the following way:

- item 1
- item 2

**Bold typeface** indicates interactive parts such as connectors and switches on the hardware, or menus and buttons in a graphical user interface.

```
This font is used to indicate program code and other
kinds of data input/output such as configuration scripts.
```

This is a cross-reference within this document: *Conventions, p. 5*

This is an external link (URL): www.hms-networks.com

*This is additional information which may facilitate installation and/or operation.*

**!** This instruction must be followed to avoid a risk of reduced functionality and/or damage to the equipment, or to avoid a network security risk.

## 1.3 Glossary

| Abbreviations | |
|---|---|
| **BAL** | Bus Access Layer |
| **CAN** | Controller Area Network |
| **FIFO** | First In/First Out Memory |
| **GUID** | Globally unique ID |
| **LIN** | Local Interconnect Network |
| **VCI** | Virtual Communication Interface |
| **VCIID** | VCI specific unique ID |
| **VCI server** | VCI system service |

# 2 System Overview

The VCI (Virtual Communication Interface) is a system extension, that provides common access to different devices by HMS Industrial Networks for applications.

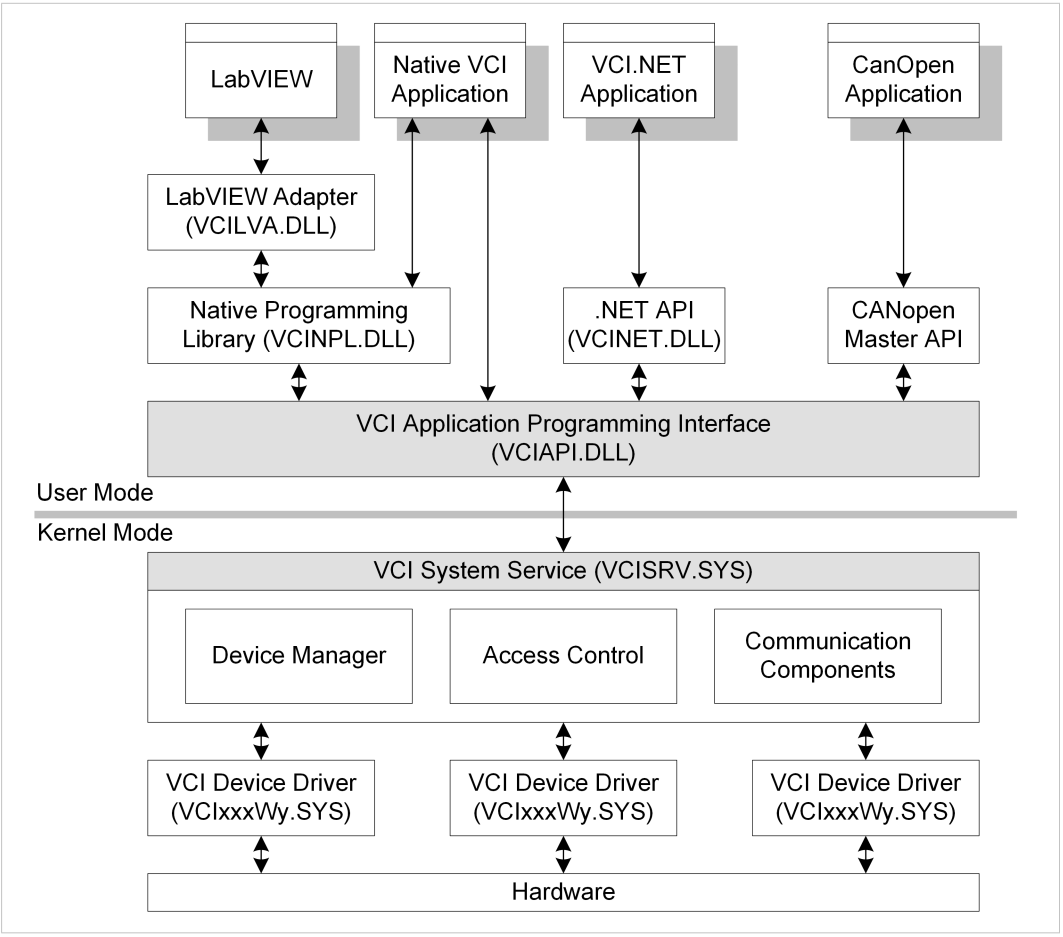


**Fig. 1 System structure and components**

The VCI consists of the following components:

- VCIAPI.DLL: user mode programming interface
- VCISRV.SYS: kernel mode system service
- VCIxxxWy.SYS: on or more device drivers

The following further components are described in other manuals:

- VCINPL.DLL: C programming interface
- VCILVA.DLL: LabVIEW adapter, allows access to VCI with LabView
- VCINET.DLL: .NET programming interface

The VCI system service (VCI server) takes over the following tasks:

- management of VCI specific device drivers
- management of the access to the device
- providing mechanisms for the exchange of data and commands between application and operating system or between user and kernel mode

The programming interfaces connect the VCI server and the application programs using predefined components, interfaces and functions.

The user mode programming interface (VCIAPI.DLL) is the basis for all superior programming interfaces and applications. The provided components implement the interface `IUnknown`, that is defined by MS-COM. The server functionality that is also specified in MS-COM is not implemented, resp. is not supported. The components don't have a COM conform fabric or automation interface, i. e. VCI specific components are not created with `IClassFactory` and do not have an `IDispatch` interface compatible to automation. They can not be used by script or .NET languages.

Regarding multi threading, simultaneous access to particular components from several threads is possible. Every thread has to open an own instance of the desired component resp. interface. The individual functions of an interface must not be called by different threads, because the implementation is not thread safe due to performance reasons. Interfaces, that have an own locking mechanism are an exception to this rule. This locking mechanism is for example provided by the interfaces *`IFifoReader`* and *`IFifoWriter`* with the functions `Lock()` and `Unlock()`.

The components don't have to be assigned to an apartment, as usual in COM. If the VCIAPI is used exclusively, without any other COM components the particular threads of an application don't have to be assigned to an apartment nor create an apartment and therefore don't have to call the function `CoInitialize()`.

# 3 Device Management and Device Access

The device management provides listing of and access to devices logged into the VCI server.

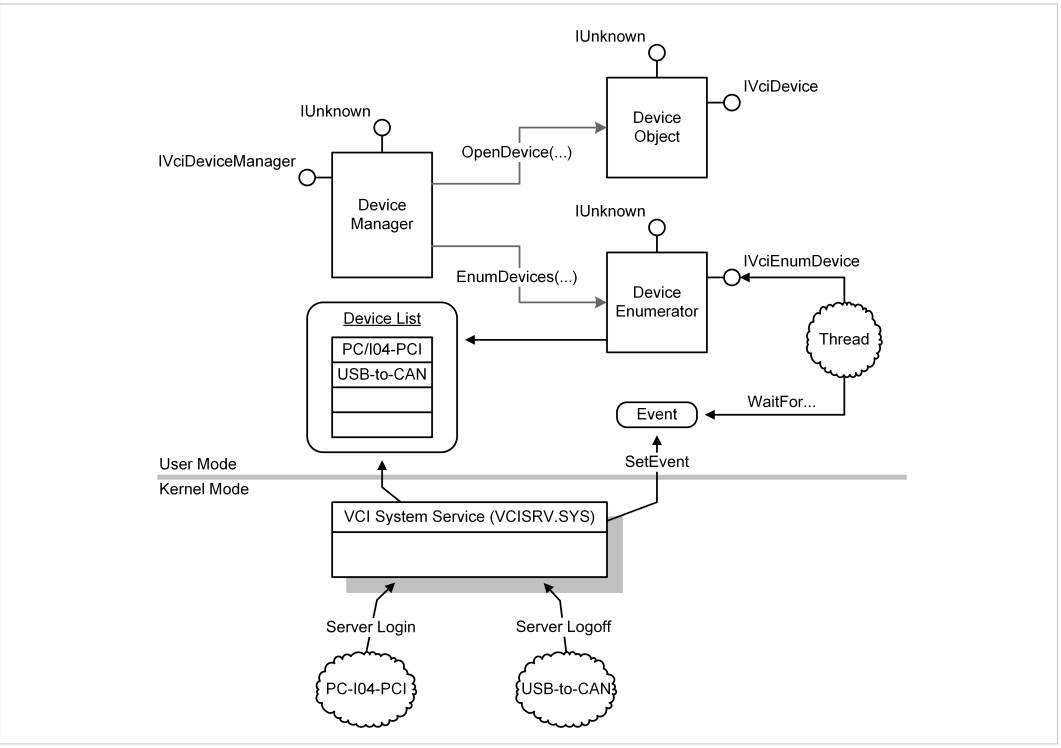


**Fig. 2 Device management components**

The VCI server manages all devices in a system-wide global device list. When the computer is booted or a connection between device and computer is established the device is automatically logged into the server.
If a device is no longer available e. g. because the connection is interrupted, the device is automatically removed from the device list.

The logged in devices are accessed via the VCI device manager or its interface *IVciDeviceManager*. A pointer to this interface is provided by the exported function *VciGetDeviceManager*.

**Main information about a device:**

| Interface | Type | Description |
|---|---|---|
| *VciObjectId* | Unique ID of device | When a device logs in it is allocated a system-wide unique ID (VCIID). This ID is required for later access to the device. |
| *DeviceClass* | Device class | All device drivers identify their supported device class by a worldwide unique ID (GUID). Different devices belong to different device classes, e. g. the USB-to-CAN belongs to a different device class as PC-I04/PCI. |
| *UniqueHardwareId* | Hardware ID | All devices have a unique hardware ID. The ID can be used to differentiate between to interfaces or to search for a device with a certain hardware ID. Remains after restart of the system. Because of that it can be stored in the configuration file and enables automatic configuration of the application after program and system start. |

# 3.1 Listing Available Devices

- To access global device list, call function `IVciDeviceManager::EnumDevices`.
  - Returns pointer to interface `IVciEnumDevice` of device list.

Information about available devices can be accessed and changes in the device list can be monitored. There are different possibilities to navigate in the device list.

### Requesting Information About Devices in Device List

Application must provide necessary memory as a structure of type `VCIDEVICEINFO`.

- Call function `IVciEnumDevice::Next`.
  - Returns description of a device in the device list.
  - With each call the internal index is incremented.
- To get information about the next device in the device list, call function `IVciEnumDevice::Next` again.

### Reset Internal List Index

- Call function `IVciEnumDevice::Reset`.
  - Subsequent call of function `vciEnumDevice::Next` provides information about the first device in the device list again.

### Skip Defined Number of Elements in Device List

- Call function `IVciEnumDevice::Skip`.
- Use of function exclusively makes sense in systems with unchangeable device list, because exclusively here the sequence of the devices is known and fix.

Hot plug-in devices like USB devices are logged in with connecting and logged out with disconnecting. The devices are also logged in or off when the operation system activates or deactivates a device driver in the device manager.

### Monitoring Changes in the Device List

- Call function `IVciEnumDevice::AssignEvent`.
  - An event object is created and assigned to the device list.
  - If a device or a driver logs in or off the VCI server the event object is automatically signalized.

## 3.2 Accessing Individual Devices

Access individual devices with function *IVciDeviceManager::OpenDevice*.

- Specify device ID in (VCIID) of the device to be opened in parameter (to determine device ID see *Listing Available Devices, p. 10*).
  - ▷ Returns pointer to interface `IVciDevice` of device list.

### Requesting Information About an Open Device

- Call function *IVciDevice::GetDeviceInfo*.
  - ▷ Necessary memory is provided by the application as structure of the type `VCIDEVICEINFO`.
  - ▷ Returns information about the device in device list (see *Main information about a device:, p. 9*).

### Requesting Information About Technical Features of a Device

- Call function *IVciDevice::GetDeviceCaps*.

  Parameter is pointer to structure of type `VCIDEVICECAPS` .
  - ▷ Function saves information about technical features of the device in the specified area.
  - ▷ Returned information inform how many bus controllers are available on a device.
  - ▷ Structure `VCIDEVICECAPS` contains a table with up to 32 entries, that address the respective bus connection resp. controller. Entry 0 describes the bus connection 1, entry 1 bus connection 2 etc.

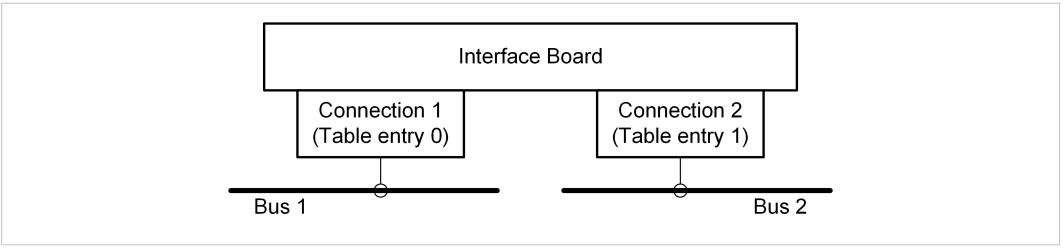


**Fig. 3 Interface with two bus controllers**

### Open Individual Layer Components

- With function *IVciDevice::OpenComponent* open individual layer components, which are used by different applications from different areas of applications to access the device (for further information see *Accessing the Bus Controller, p. 20*).

# 4 Communication Components

The applications communicate with the drivers resp. with the firmware running on the device with special communication components. The VCI provides diverse components for different requirements. For bus specific applications the First In/First Out memory (FIFO) is important.

Each memory used by the communication components comes from the *non-paged* memory, a part of the main memory which is not released from the operation system. This memory pool which is also used by other device drivers and by the operating system has a limited size, dependent on the version of the operating system and the available physical memory.

The 32 bit Windows variant reserves for the *non-paged* pool approx. 1/4 of the available main memory, maximum 256 MB (also in systems with more than 1 GB main memory.) If the 3GB boot option is active, maximum 128 are available. The 64 bit Windows variant reserves for the *non-paged* pool approx. 400 KB per MB available main memory, maximum 128 GB.

**Size of memory reserved for the *non-paged* non-paged pool of different Windows versions**

| | **32-bit Systems** | **64-bit Systems** |
|---|---|---|
| Windows XP,<br>Server 2003 | up to 1,2 GB RAM: 32-256 MB<br>more than 1,2 GB: 256 MB | approx. 400 KB per MB RAM,<br>max 128 GB |
| Windows Vista,<br>Server 2008,<br>Windows 7,<br>Server 2008R2 | Dynamically assigned, up to approx. 75% of RAM,<br>max 2 GB | Dynamically assigned, up to approx. 75% of RAM,<br>max 128 GB |

To determine the size of the memory pool via the registry the value of *NonPagedPoolSize* has to be adjusted. This value is in:

```
HKEY_LOCAL_MACHINE
  \SYSTEM
    \CurrentControlSet
      \Control
        \Session Manager
          \Memory Management\
```

*Take care of an amount and/or a size of the FIFOs as small as possible because of the limited size of the pool in 32 bit systems.*

The memory actually occupied by the FIFO is dependent on the requested dimensions, but always contains at least one physical memory site, that contains 4 KB in 32 bit systems and 8 KB in 64 bit systems. Individual FIFOs can be bigger than requested. For example the calculated required memory of a FIFO with 32 elements with each 16 byte per unit is 512 byte. For the user invisible control fields are added, which in this case need additionally 24 bytes.

If such a FIFO is created in a 32 bit system, the system reserves a memory site with 4 KB. The FIFO only needs 512+24 bytes and the unused range is not used for other components due to security reasons. 3560 bytes are wasted. In FIFOs this unused range is used to increase the number of elements available to the maximum number of elements allowed for the allocated range. If a FIFO with the above stated dimensions is for example created on a 32 bit system, the FIFO has 222 additional elements, in all 254 instead of the requested 32 elements.

## 4.1 First In/First Out Memory (FIFO)

The VCI contains an implementation for First-In/First-Out memory objects.

FIFO features:

- Dual-port memory, in which data is written on the input side and read on the output side.
- Chronological sequence is preserved, i. e. data that is written in the FIFO at first is also read at first.
- Similar to the functionality of a pipe connection and therefore also named pipe.
- Used to transfer data from a transmitter to the parallel receiver. Agreement with a lock mechanism, who has access to the common memory area at a certain point of time is not necessary.
- No locking, possible to be overcrowded, if receiver does not manage to read the data in time.
- Transmitter writes the messages to transmit with interface *IFifoWriter* in the FIFO. Receiver parallel reads the data with interface *IFifoReader*.

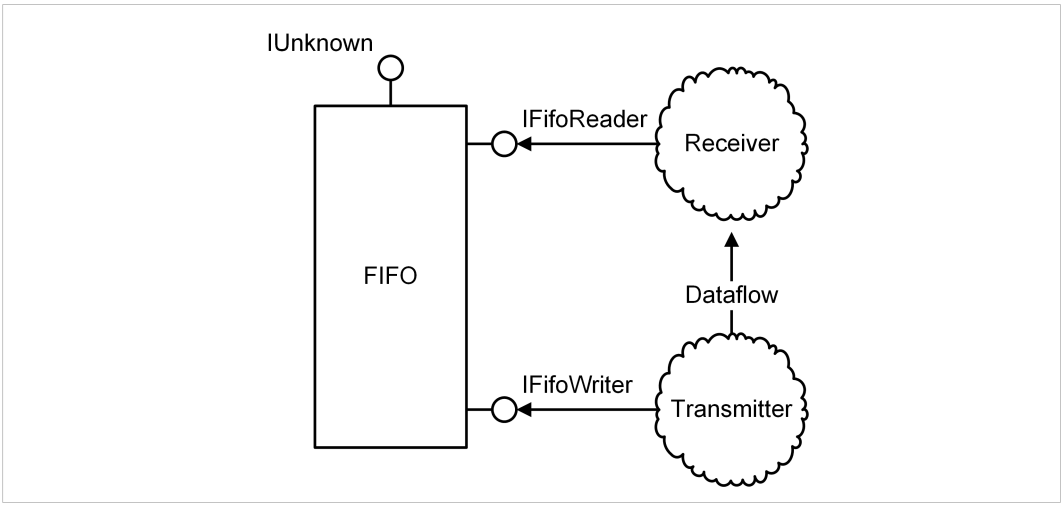


**Fig. 4 FIFO data flow**

Access:

- Writing and reading access to a FIFO is possible simultaneously, a receiver is able to read data while a transmitter writes new data to the FIFO.
- Simultaneous access of several transmitters resp. receiver to the FIFO is not possible.
- Multiple access to interfaces *IFifoReader* and *IFifoWriter* is prevented, because the respective interface of the FIFO can only be opened once, i. e. not until the interface is released can it be opened again.
- To prevent simultaneous access to one interface by different threads of an application:
  - Make sure functions of an interface can only be called by one thread of the application.

    or
  - Synchronize access to an interface with respective thread: Call function `Lock` before every accessing the FIFO and after accessing call function `Unlock` of the respective interface.

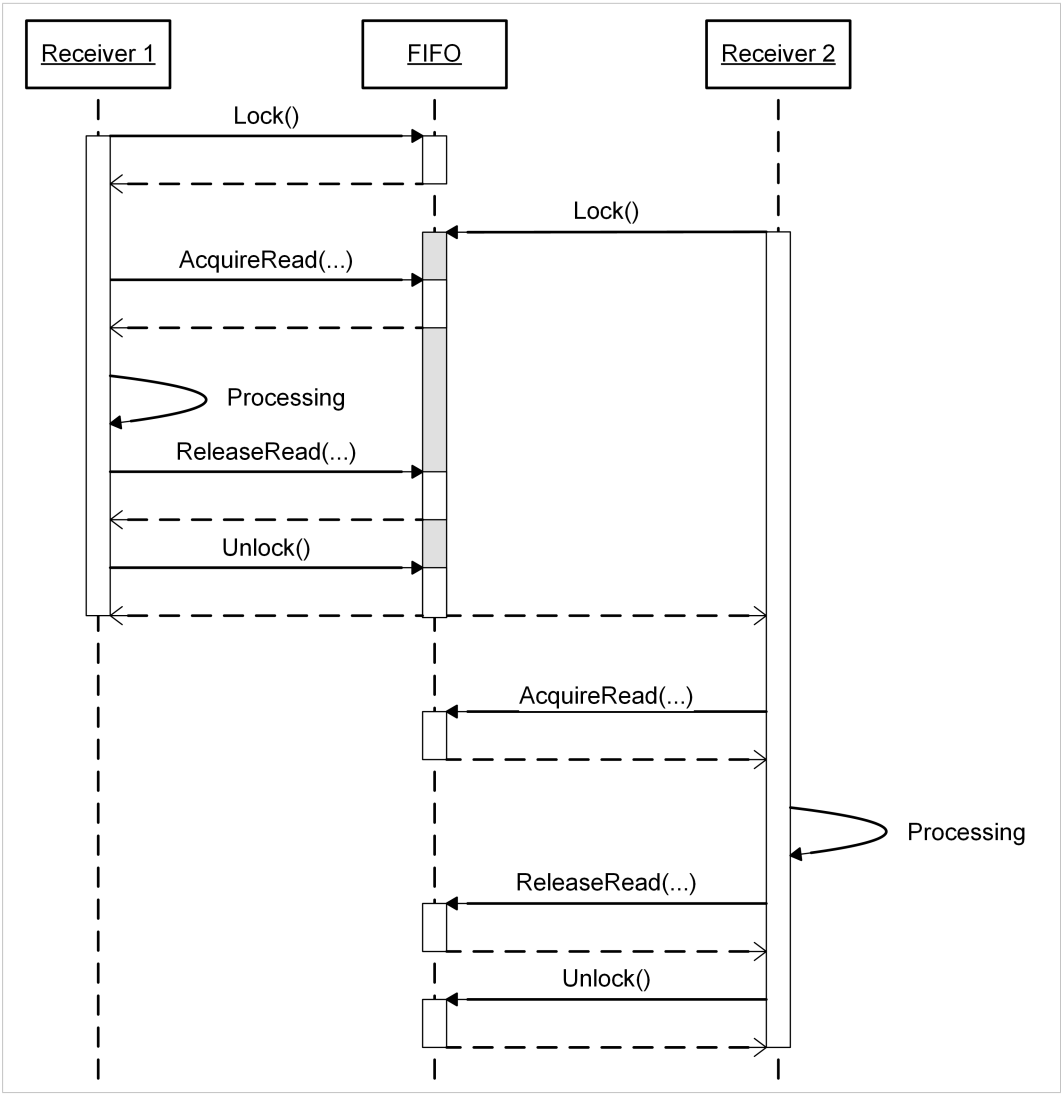


**Fig. 5 FIFO locking mechanism**

Receiver 1 calls function *Lock* and gains access to the FIFO. The following call of *Lock* by receiver 2 is blocked until receiver 1 releases the FIFO with calling function *Unlock*. Now receiver 2 can start processing. In the same way two transmitters that access the FIFO with the interface *IFifoWriter* can be synchronized.

The FIFOs provided by the VCI also allow the exchange of data between two processes, i. e. over the boundaries of the process.

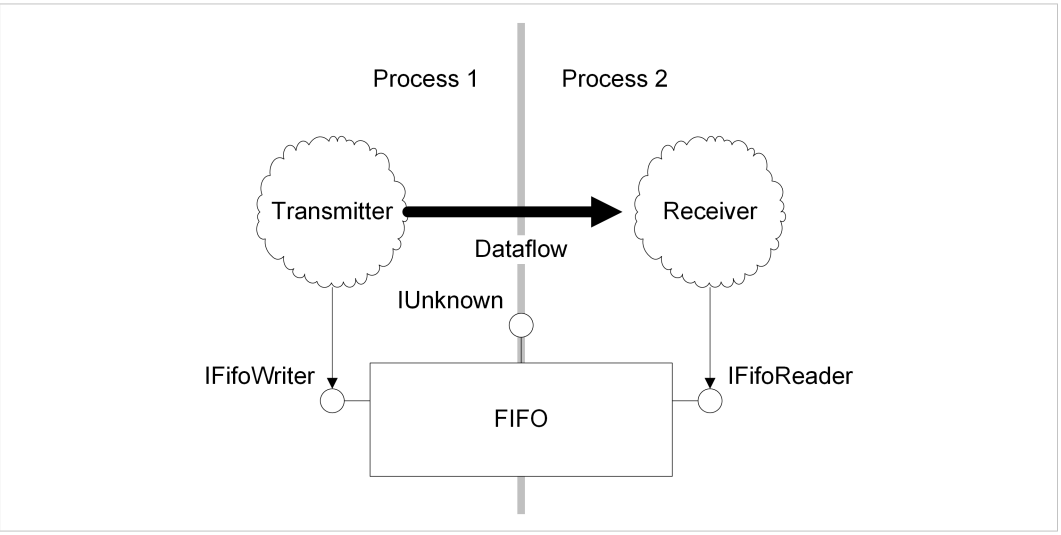


**Fig. 6 FIFO for data exchange between two processes**

FIFOs are also used to exchange data between components running in the kernel mode and programs running in the user mode.

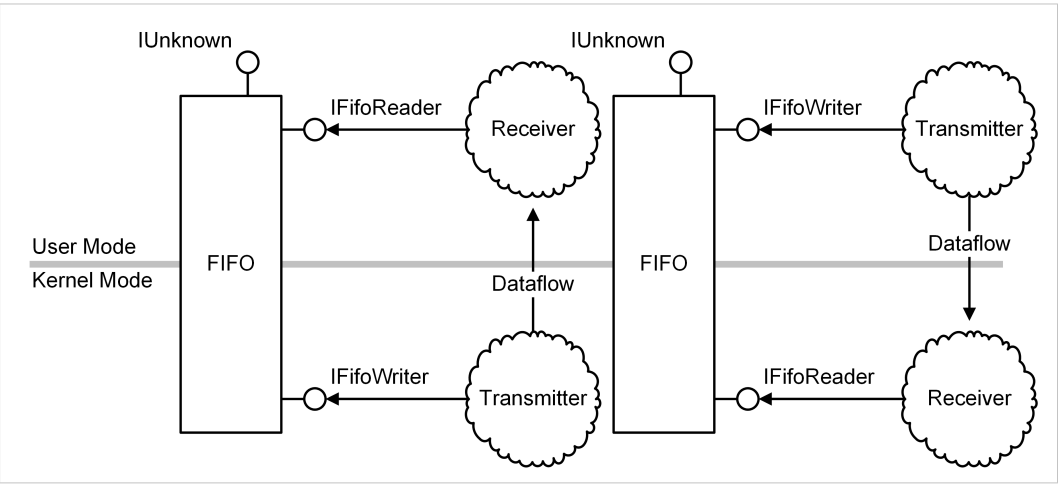


**Fig. 7 Possible combination of a FIFO for data exchange between user and kernel mode**

Applications can establish data channels with the functions *VciCreateFifo* resp. *VciAccessFifo* and are not, like *Pipes* are, dependent on operating system specific mechanisms, .

### 4.1.1 Functionality Receiving FIFO

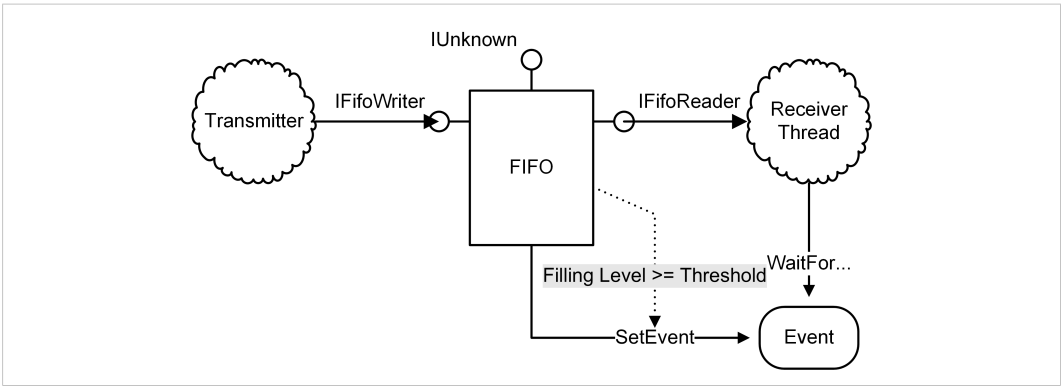


**Fig. 8 Functionality receiving FIFO**

At the receiving side FIFOs are addressed via the interface *IFifoReader*.

Access files to read:

- Call function *GetDataEntry*.
  - ▷ Next valid data element in the FIFO is read and released.

  or
- Call function *AcquireRead*.
  - ▷ Pointer to next valid element and number of valid elements that can be read sequentially from this position onward is determined.
- To release one or more read and processed elements call function *ReleaseRead*.

Because FIFOs reserve a sequential memory area it is possible *AcquireRead* returns less valid entries than are actually available.

- Repeat calling functions *AcquireRead* and *ReleaseRead* in a loop until no more valid elements are available.

The address returned when calling *AcquireRead* points directly to the memory used by the FIFO.

- Make sure that no element outside the valid area is called during access.

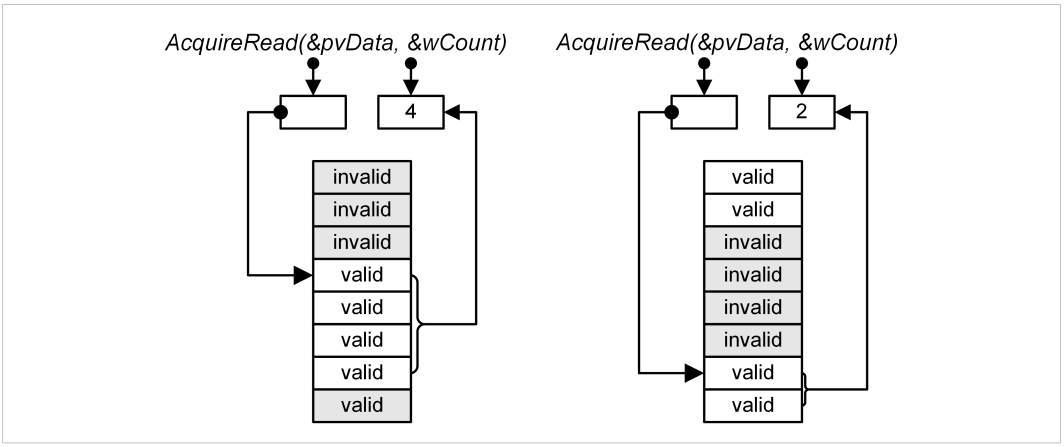


**Fig. 9 Functionality of `AcquireRead`**

### Event Object

It is possible to assign an event object to the FIFO to prevent that the receiver has to ask wether new data is available for reading. The event object is set to a signaled status if a certain threshold is reached or exceeded.

- Create `CreateEvent` with Windows API function.
  - Returned Handle is assigned to FIFO with function *`AssignEvent`*.
- Set threshold resp. filling level that triggers the event with function *`SetThreshold`*.

Afterwards the application is able to wait for the event and to read the received data with one of the Windows API functions `WaitForSingleObject`, `WaitForMultipleObjects` or one of the functions `MsgWaitFor…`.

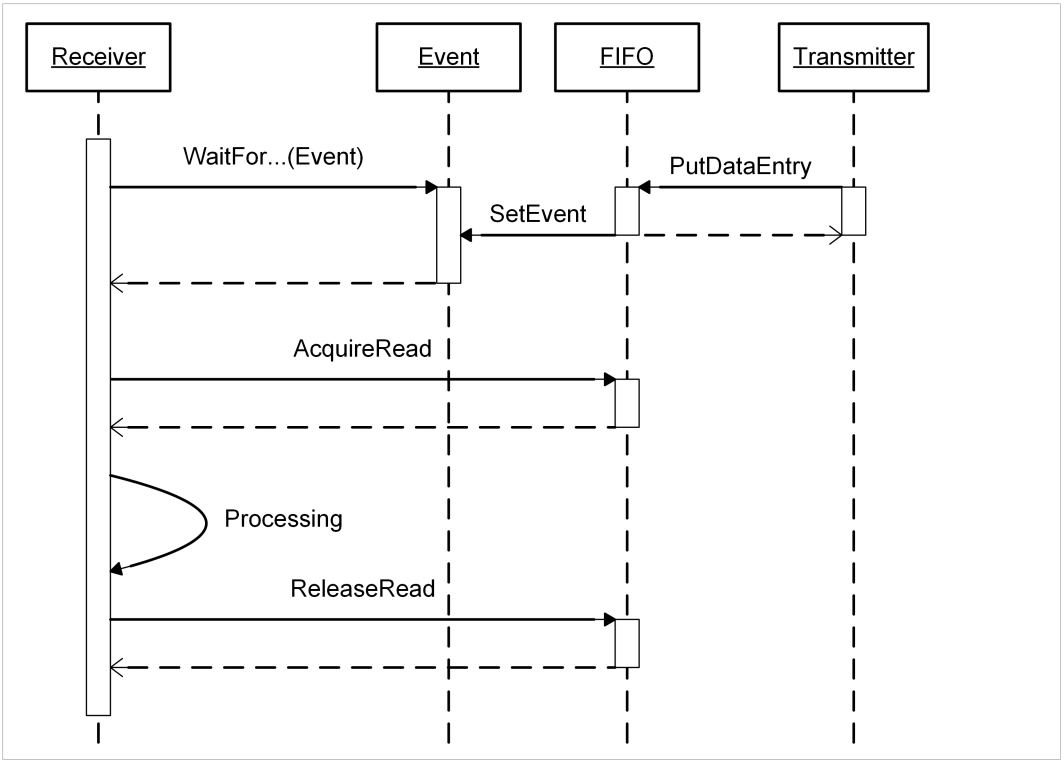


**Fig. 10 Receiving sequence event-driven reading of data from the FIFO**

*Since the event is exclusively triggered with the exceedance of the set threshold, make sure that all entries of the FIFO are read in case of event-driven reading. If the threshold is set e. g. 1 and already 10 elements are in the FIFO when the event happens and only one is read, a following event will not be triggered until the next write-access. If no further write-access follows by the transmitter 9 unread elements are in the FIFO that aren't shown as event anymore.*

### 4.1.2 Functionality Transmitting FIFO

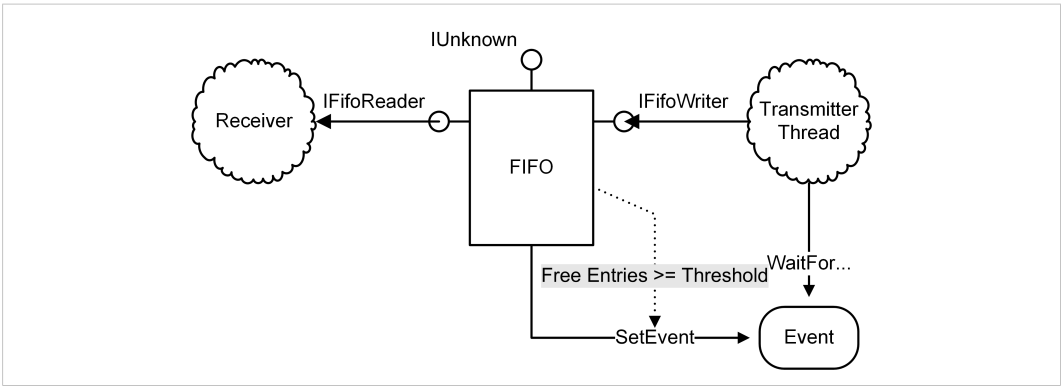


**Fig. 11 Functionality Transmitting FIFO**

At the transmitting side FIFOS are addressed via the interface *IFifoWriter*.

Write data to be transmitted in the FIFO:

- To write individual data element to the FIFO call function *PutDataEntry*.
  - Data element is marked valid and can be read by the receiver.

  or
- Call function *AcquireWrite*.
  - Pointer to next free element and number of free elements that can be addressed sequentially from this position onward is determined.
- Declare one or more addressed elements in the FIFO valid with function *ReleaseWrite*.
  - New elements are visible for the receiver and can be read.

Because FIFOs reserve a sequential memory area it is possible that *AcquireWrite* returns less free entries than are actually available.

- Repeat calling functions *AcquireWrite* and *ReleaseWrite* in a loop until no more free elements are available.

The address returned when calling *AcquireWrite* points directly to the memory used by the FIFO.

- Make sure that no element outside the valid area is addressed during access.

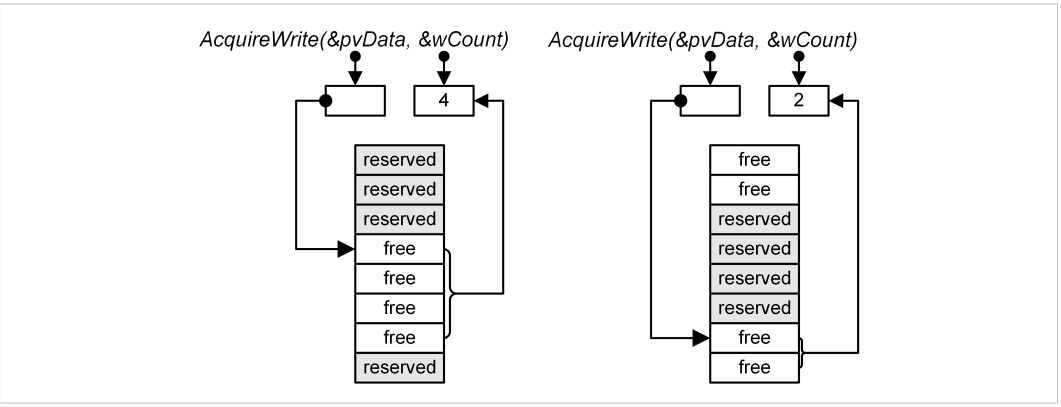


**Fig. 12 Functionality of AcquireWrite**

### Event Object

It is possible to assign an event object to the FIFO to prevent that the transmitter must check if free elements are available. The event object is set to a signaled status if the number of free elements has reached or exceeded a certain value.

- Create `CreateEvent` with Windows API function.
  - Returned Handle is assigned to FIFO with function *`AssignEvent`*.
- Set threshold resp. number of free elements that triggers the event with function *`SetThreshold`*.

Afterwards the application is able to wait for the event and to write the new data in the FIFO with one of the Windows API functions `WaitForSingleObject`, `WaitForMultipleObjects` or one of the functions `MsgWaitFor`….

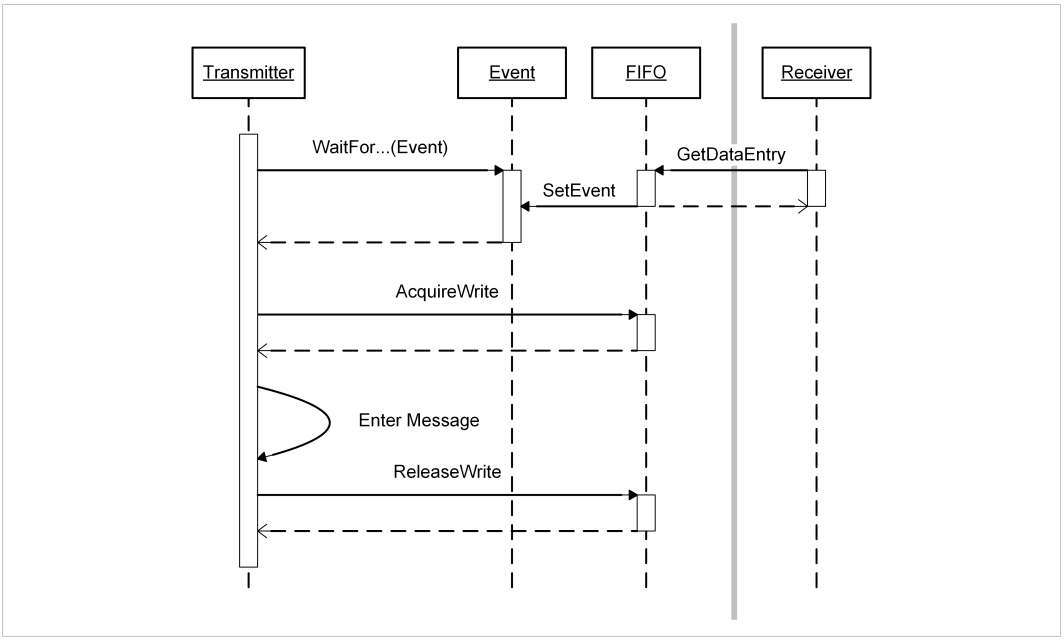


**Fig. 13 Transmitting sequence event-driven writing of data to FIFO**

# 5 Accessing the Bus Controller

### Open Individual Layer Components

With function *IVciDevice::OpenComponent* open individual layer components, which are used by different applications from different areas of applications to access the device.

- ▶ In first parameter determine which layer is opened (for further information see *OpenComponent*).
- ▶ In second Parameter determine the interface to access.

The different layers are locked against each other and can not be opened simultaneously. If e. g. an application opens another layer of the BAL, no component of the BAL can be opened until all components of the other layer resp. the layer itself is closed.

## 5.1 BAL

The data buses that are connected via a bus adapter are accessed with the Bus Access Layer (BAL).

- Provides components and interfaces for the access to available bus controller and direct communication with the connected bus system.
- Interfaces abstract and encapsulate the communication with the controller hardware in such a way that applications can mostly be implemented independently of the special features of different bus controllers.
- The BAL can be opened several times simultaneously (not secured against multiple opening). Different applications can access different bus connections simultaneously.

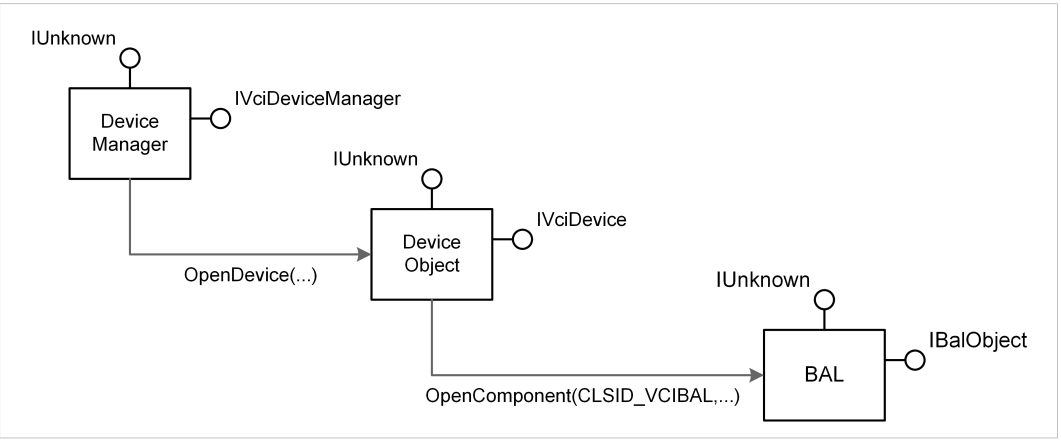


**Fig. 14 Components for accessing the bus**

- ▶ Search adapter in device list and open with `IVciDeviceManager::OpenDevice`.
- ▶ Open BAL components with function `IVciDevice::OpenComponent`.
- ▶ In first parameter specify value `CLSID_VCIBAL`.
- ▶ In second parameter specify value `IID_IBalObject`, to specify desired interface to access (BAL exclusively supports interface `IBalObject` ).
- ▶ Call function.
  - ▷ Returns pointer to interfaces `IBalObject` in third parameter.
  - ▷ If an error occurs, the function returns error code unlike `VCI_OK`.

- After opening release references to the device manager resp. the device object that are no longer needed with `Release`.

For further work with the adapter only the BAL object resp. its interface *`IBalObject`* is necessary. The interface *`IBalObject`* can be opened by several programs simultaneously.

The BAL object supports several types of controllers and bus connections.

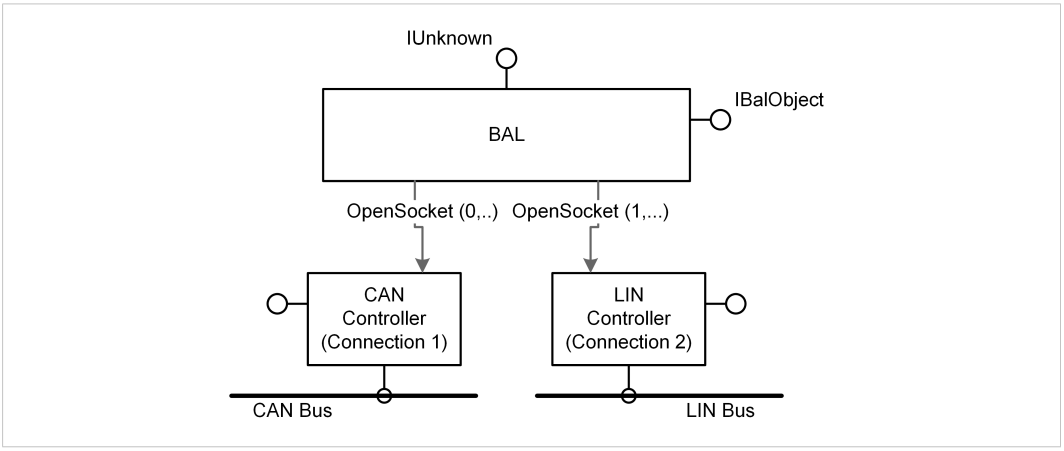


**Fig. 15 BAL with CAN and LIN controller**

### Determine Number and Type of Provided Connections

- Call function `IBalObject::`*`GetFeatures`*.
  - Returns information as structure of type *`BALFEATURES`*.

### Access Connection or Interface of Connection

Access connection with `IBalObject::`*`OpenSocket`*.

- In first parameter determine number of connection to be opened. The value must be in the range 0 to *BusSocketCount*-1. To open connection 1 enter value 0, for connection 2 value 1 etc.
- In second parameter determine the ID of the interface to access the controller.
- Call function.
  - Returns in the variable the third parameter is pointing to the address of the desired interface.
  - Possibilities resp. interfaces of a connection are dependent on the supported bus.

*Certain interfaces of a connection can only be accessed by one program, others can be accessed by any number of programs simultaneously. The rules of accessing the particular interfaces are dependent on the type of the connection and are described in detail in the following chapters.*

## 5.2 CAN Controller

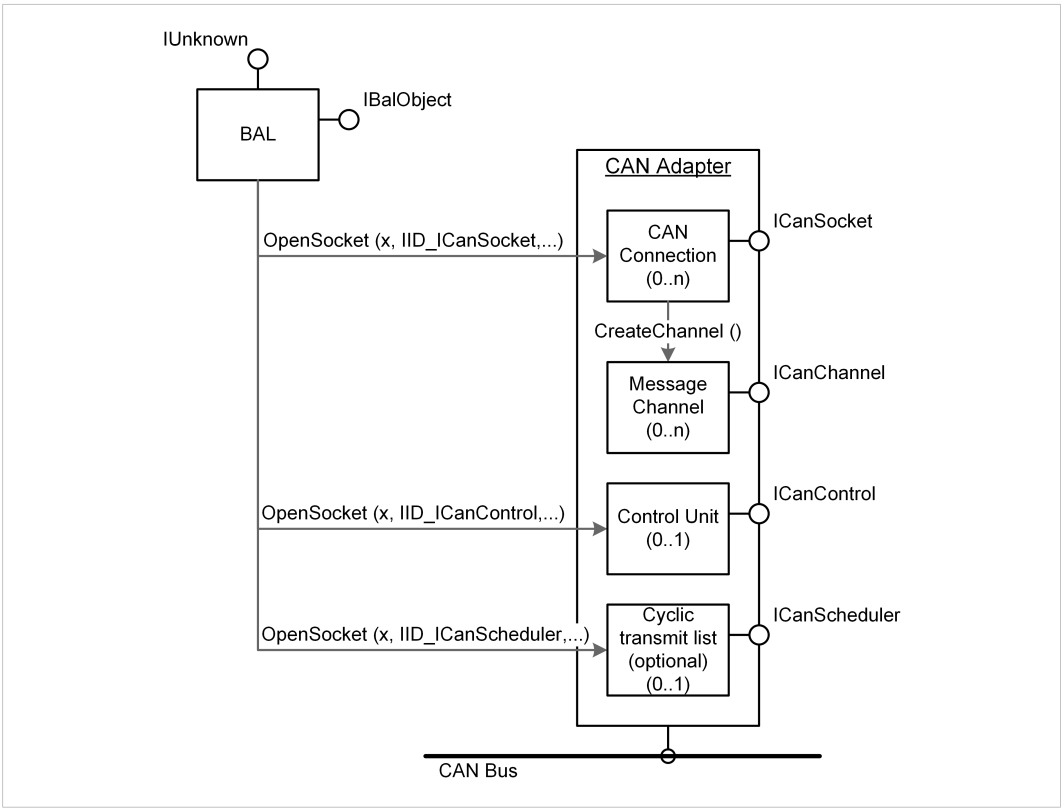


**Fig. 16 Components CAN controller and interface IDs**

Access individual components resp. interfaces of CAN controller with function `IBalObject::OpenSocket`. For a complete description of all interfaces and the IDs that are necessary for opening see *CAN Specific Interfaces, p. 73*.

Supported interfaces of the components:

- `ICanSocket`, `ICanSocket2` (CAN controller), see *Socket Interface, p. 22*.
- `ICanControl`, `ICanControl2` (control unit), see *Control Unit, p. 31*
- `ICanChannel`, `ICanChannel2` (message channel), see *Message Channels, p. 23*.
- `ICanScheduler`, `ICanScheduler2` (cyclic transmitting list), see *Cyclic Transmitting List, p. 44*

  Optional, exclusively with devices with their own microprocessor

The extended interfaces `ICanSocket2`, `ICanControl2`, `ICanChannel2` and `ICanScheduler2` allow the access to the new functions of CAN FD controller. With standard controllers they can be used for further filter possibilities

### 5.2.1 Socket Interface

The socket interface *`ICanSocket`* resp. *`ICanSocket2`* is used to request features, possibilities and operating status of the CAN controller The interface is not subjected to any access restrictions and can be opened by multiple applications simultaneously.

Open with function `IBalObject::OpenSocket`.

▶ In parameter *riid* specify value `IID_ICanSocket` or `IID_ICanSocket2` depending on the functionality.

- Call function.
- To request features of the connection, like the controller type in use, the type of bus coupling and the supported features, call function `GetCapabilities` (for further information about returned data see *CANCAPABILITIES* and *CANCAPABILITIES2*).
- To determine current operating state of controller call function `GetLineStatus` (for further information see *CANLINESTATUS* and *CANLINESTATUS2*).
- Create message channels with function `CreateChannel`.

## 5.2.2 Message Channels

Message channels consist of a receiving and an optional transmitting FIFO.

Message channels with extended functionality (CAN FD) contain an additional, optional input filter.

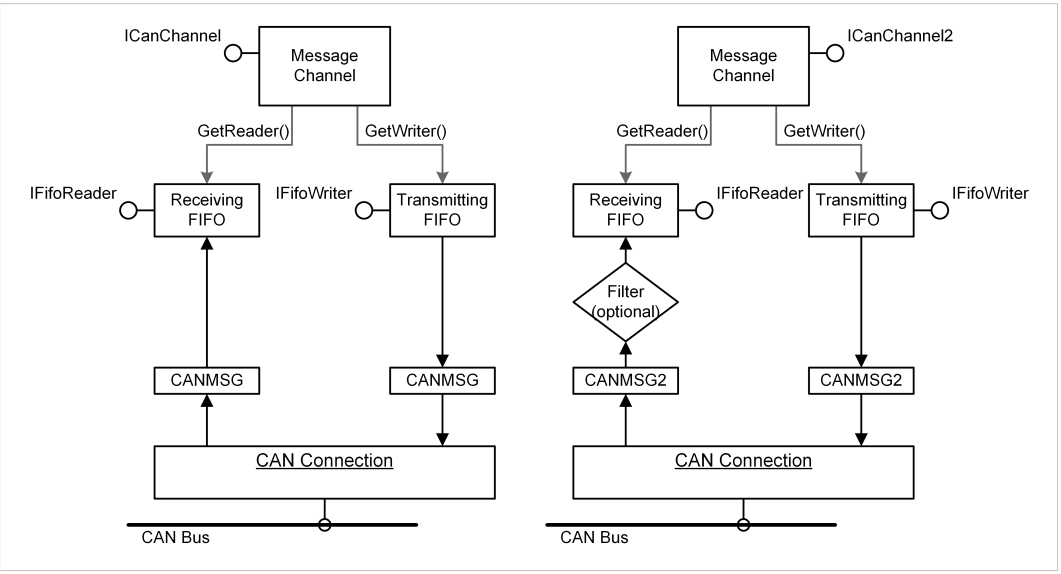


**Fig. 17 Components and interfaces of a message channel**

The size of the data elements in the FIFO corresponds to the size of the structure *CANMSG*, or with message channels with extended functionality the size of the structure *CANMSG2*. All functions to access the data elements of the FIFO attend resp. return a pointer to structures of type *CANMSG* resp. *CANMSG2* (description see *First In/First Out Memory (FIFO), p. 13*).

All CAN connections support message channels of the type `ICanChannel` and `ICanChannel2`. If the extended functionality of a message channel of type `ICanChannel2` is usable, is depending on the CAN controller of the connection. If the connection provides e. g. only a standard CAN controller, the extended functionality can not be used. With a message channel of type `ICanChannel` the extended functionality of a CAN FD neither can be used.

The basically functionality of a message channel is the same, irrespective whether the connection is used exclusively or not. In case of exclusive usage the message channel is directly connected to the CAN controller.

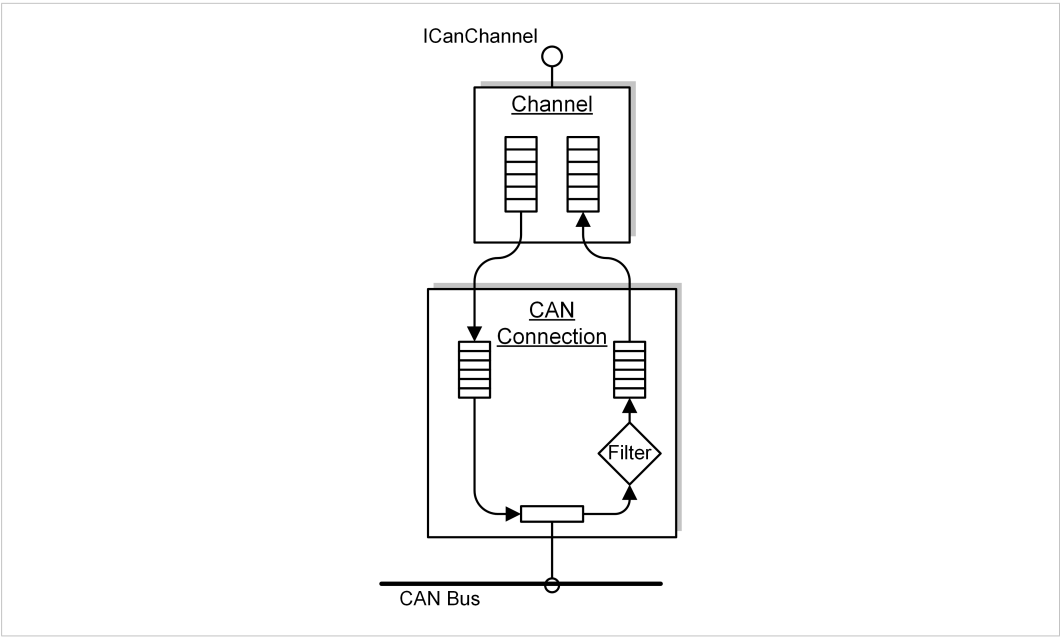


**Fig. 18 Exclusive use of a message channel**

In case of non-exclusive usage the individual message channels are connected to the controller via a distributor.

The distributor transfers all received messages to all active channels and parallel the transmitted messages to the controller. No channel is prioritized i. e. the algorithm used by the distributor is designed to treat all channels as equal as possible.

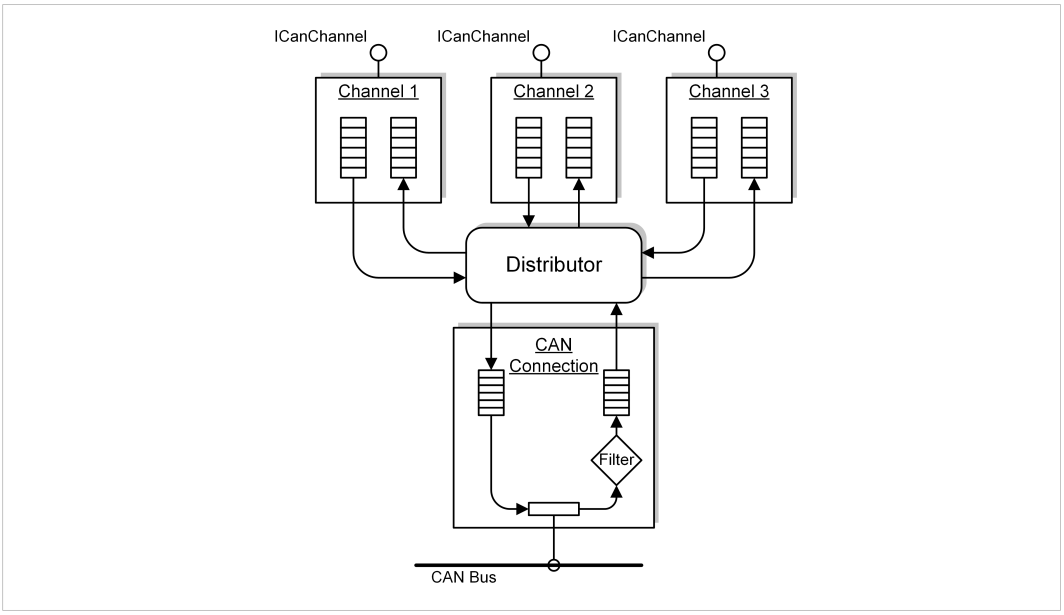


**Fig. 19 CAN message distributor: possible configuration with three channels**

### Creating a Message Channel

Create message channel with function `ICanSocket::CreateChannel` resp. for channels with extended functionality with `ICanSocket2::CreateChannel`.

- If controller is used exclusively (exclusively with the first message channel) enter in parameter `fExclusive` value `TRUE`.

  or

  If controller is used non-exclusively (further message channels can be opened and controller can be used by other applications) enter in parameter `fExclusive` value `FALSE`.

### Initializing the Message Channel

A newly generated message channel contains neither a receive nor a transmitting FIFO. Before using an initialization is necessary.

*The random access memory required for the FIFOs (see Communication Components, p. 12) limits the possible number of channels.*

Initialize with function `ICanChannel::Initialize` resp. with a controller with extended functionality with `ICanChannel2::Initialize`.

- Specify size of receiving FIFO in parameter *wRxFifoSize* resp. *dwRxFifoSize*.
- Make sure that value in parameter *wRxFifoSize* is higher than 0.
- Specify size of transmitting FIFO in parameter *wTxFifoSize* resp. *dwTxFifoSize*.

  The size determines the number of messages that the respective FIFO must record at the mininum.
- If no sending FIFO is required, set value in *wTxFifoSize* resp. *dwTxFifoSize* to 0.

In case of usage of message channels with extended functionality an additional optional receive filter can be created.

- In case of a 29 bit filter specify size of filter table in number of IDs in parameter *dwFilterSize*.

  In case of 11 bit filter size of filter table is specified to 2048 and can not be changed.
- If no receiving filter is needed set *dwFilterSize* to 0.
- Specify functionality for 11 bit and 29 bit filter in parameter *bFilterMode*.
- Call function.

*Initially specified functionality can be changed later for both filters separately with the function `SetFilterMode` in inactive message channels.*

### Activating the Message Channel

A new message channel is inactive. Messages can't be transmitted nor received before activation.

- To connect channel to controller and to activate the message transport call function `Activate`.
- To activate message transport between channel and bus, start controller with control unit.

  Which messages are received by the bus, is dependent on the settings of the message filter in the controller (further information about the control unit and message filters see *Control Unit, p. 31* and *Message Filter, p. 40*).

- Disconnect active channel with function `Deactivate`.

  To change the filter settings of a message channel the message channel must be inactive, i. e. disconnected from the controller.

## Receiving CAN Messages

The messages received on the bus and accepted by the filter are written to the receiving FIFO by the distributor.

- To access FIFO call function `ICanChannel::GetReader` resp. with a controller with extended functionality `ICanChannel2::GetReader`.
  - ▷ Pointer to interface *`IFifoReader`* is returned.
  - ▷ Make sure that in all functions that return a pointer to FIFO elements the elements in the receiving FIFO are always of the type *`CANMSG`* resp. with a controller with extended functionality of the type *`CANMSG2`*.

Reading messages from the FIFO:

- Make sure that parameter *pvData* points to buffer of type *`CANMSG`* resp. *`CANMSG2`*.
- Call function `IFifoReader::GetDataEntry`.

  or

- Call function `IFifoReader::AcquireRead`.
  - ▷ Returns pointer to next valid message in the FIFO and the number of messages that can be read sequentially ascending from this position onward.
- After processing remove data with function `IFifoReader::ReleaseRead` from FIFO.

---

*The address returned by `AcquireRead` points directly to the memory of the FIFO. Make sure that exclusively elements of the valid range are addressed.*

---

**Possible Usage of Function `GetDataEntry`:**

```
void ReceiveMessages(IFifoReader* pReader)
{
  CANMSG sCanMsg;

  while( pReader->GetDataEntry(&sCanMsg) == VCI_OK )
  {
    // Processing of message
  }
}
```

**Possible Usage of Functions `AcquireRead` and `ReleaseRead`:**

```
void ReceiveMessages(IFifoReader* pReader)
{
  PCANMSG2 pCanMsg2;
  UINT16 wCount;

  while( pReader->AcquireRead((PVOID*) &pCanMsg2, &wCount) == VCI_OK )
  {
    for( UINT16 i = 0; i < wCount; i++ )
    {
      // processing of message
      .
      .
      .
      // set pointer ahead to next message
      pCanMsg2++;
    }
    // release read message
    pReader->ReleaseRead(wCount);
  }
}
```

**Advantages of Usage of `AcquireRead` and `ReleaseRead`:**

- Application decides when data is copied or not.
- Application decides how many messages are removed from the FIFO.
- Useful when applications processes messages exclusively selective.

Example:
If the application detects that at a moment only two of five incoming messages can be processed, because otherwise there is an overflow somewhere else, *`ReleaseRead`* can be called with value 2 instead of value 5. A subsequent calling of *`AcquireRead`* returns a pointer to the three messages not yet processed.

### Receiving Time of a Message

The receiving time of a message is noted in the field *dwTime* of structure *`CANMSG`* resp. *`CANMSG2`*. The field contains the number of timer ticks that elapsed since the start of the controller resp. the hardware or since the last overrun of the counter.

**Calculation of the length of a tick resp. the resolution of a time stamp in seconds: ($t_{tsc}$):**

- $t_{tsc}$ [s] = *dwTscDivisor* / *dwClockFreq*

  Fields *dwClockFreq* and *dwTscDivisor*, see *`CANCAPABILITIES`*
- Channels with extended functionality:

  $t_{tsc}$ [s] = *dwTscDivisor* / *dwTscClockFreq*

  Fields *dwTscClockFreq* and *dwTscDivisor*, see *`CANCAPABILITIES2`*

**Calculation of the relative receiving time ($T_{rx}$):**

- $T_{rx}$ [s] = *dwTime* * $t_{tsc}$

When the control unit is started a message of type `CAN_MSGTYPE_INFO` is written in the receiving FIFOs of all active message channels. The time stamp of this message contains the relative starting point of the controller (for further information see *CANMSGINFO*).

### Transmitting CAN Messages

Messages are transmitted via the transmitting FIFO of the message channel.

- To access FIFO call interface *IFifoWriter* with function `ICanChannel::GetWriter` resp. with a controller with extended functionality with function `ICanChannel2::GetWriter`.

Write messages to the FIFO:

- Make sure that parameter *pvData* points to buffer of type *CANMSG* resp. *CANMSG2*.
- Make sure that buffer is initialized with valid values.
- Call function `IFifoWriter::PutDataEntry`.

  Exclusively messages of the type `CAN_MSGTYPE_DATA` can be transmitted. Messages with other values in the field *uMsgInfo.Bytes.bType* are ignored by the controller and automatically rejected. For detailed information about the field *uMsgInfo* of a CAN message see *CANMSGINFO*.

  or

- Call functions `IFifoWriter::AcquireWrite`.
  - ▷ Returns pointer to next free entry of the FIFO and the number of messages that can be addressed sequentially ascending from this position onward.
- Make sure that exclusively data of type *CANMSG* resp. with a controller with extended functionality of type *CANMSG2* is copied to the pointer.
- To declare the messages written in the FIFO valid call function `IFifoWriter::ReleaseWrite`.
  - ▷ Controller transmits messages to the bus.
  - ▷ Returned address points directly to the memory used by the FIFO.
- Make sure that no element outside the valid area is addressed during access.

**Possible Usage of Function `PutDataEntry`:**

```
BOOL TransmitByte(IFifoWriter* pWriter, UINT32 dwId, UINT8 bData)
{
  CANMSG sCanMsg;

  // Initialize CAN message.
  sCanMsg.dwTime = 0; // send immediately, therefore 0
  sCanMsg.dwMsgId = dwId; // message ID (CAN-ID)

  sCanMsg.uMsgInfo.Bytes.bType = CAN_MSGTYPE_DATA;
  sCanMsg.uMsgInfo.Bytes.bReserved = 0; // reserved, always 0

  sCanMsg.uMsgInfo.Bits.srr = 0; // no Self-Reception
  sCanMsg.uMsgInfo.Bits.rtr = 0; // no Remote-Request
  sCanMsg.uMsgInfo.Bits.ext = 0; // Standard Frame Format
  sCanMsg.uMsgInfo.Bits.dlc = 1; // only1 data byte

  sCanMsg.abData[0] = bData;

  // send message
  return( pWriter->PutDataEntry(&sCanMsg) == VCI_OK );
}
```

**Possible Usage of Functions `AcquireWrite` and `ReleaseWrite` with Message Channels with Extended Functionality:**

```
BOOL TransmitByte(IFifoWriter* pWriter, UINT32 dwId, UINT8 bData)
{
  PCANMSG2 pCanMsg2;

  if( pWriter->AcquireWrite((PVOID*) &pCanMsg2, NULL) == VCI_OK )
  {
    // Initialize CAN message.
    sCanMsg2.dwTime = 0; // send immediately, therefore 0
    pCanMsg2->_rsvd_ = 0; // reserved, always0
    sCanMsg2.dwMsgId = dwId; // message ID (CAN-ID)

    pCanMsg2->uMsgInfo.Bytes.bType = CAN_MSGTYPE_DATA;
    pCanMsg2->uMsgInfo.Bytes.bFlags = 0; // preinitialized with 0
    pCanMsg2->uMsgInfo.Bytes.bFlags2 = 0; // preinitialized with 0

    pCanMsg2->uMsgInfo.Bits.fdr = 1; // use Fast Data bit rate
    pCanMsg2->uMsgInfo.Bits.ext = 1; // Extended Frame Format
    sCanMsg2.uMsgInfo.Bits.dlc = 1; // only1 data byte

    pCanMsg2->abData[0] = bData;

    // and send
     pWriter->ReleaseWrite(1);
    return TRUE;
  }

  return FALSE;
}
```

### Transmitting Messages Delayed

Controller with set bit `CAN_FEATURE_DELAYEDTX` in field *dwFeatures* of structure *CANCAPABILITIES* resp. *CANCAPABILITIES2* support the possibility to transmit messages delayed, with a latency between two consecutive messages.

Delayed transmission can be used to reduce the message load on the bus. This prevents that other to the bus connected participants receive to much data in to short a time, which can cause data loss in slow nodes.

▶ In field *dwTime* of structure *CANMSG* resp. *CANMSG2* specify number of ticks that have to pass at a minimum before the next message is written in the transmitting buffer by the controller.

#### Delay Time

- Value 0 triggers no delay, that means a message is transmitted the next possible time.
- The maximal possible delay time is determined by the field *dwMaxDtxTicks* of structure *CANCAPABILITIES* resp. *CANCAPABILITIES2*, the value in *dwTime* must not exceed the value in *dwMaxDtxTicks*.

#### Calculation of the duration of a tick delay counter in seconds ($t_{dtx}$)

- $t_{dtx}$ [s] = dwDtxDivisor / dwClockFreq
- Channels with extended functionality:

  $t_{dtx}$ [s] = dwDtxDivisor / dwDtxClockFreq
- Delay time of message in seconds ($T_{delay}$):

  $T_{delay}$ [s] = dwTime * $t_{dtx}$

The specified delay time represents a minimal value as it can not be guaranteed that the message is transmitted exactly after the specified time. Also, it has to be considered that if several message channels are used simultaneously on one connection the specified value is basically exceeded because the distributor handles all channels one after the other.

▶ If an application requires a precise time sequence use connection exclusively.

### Sending Messages Uniquely

The controller tries to transmit transmitting messages with set bit `uMsgInfo.Bits.ssm` only once. If this transmitting attempt is not successful the message is rejected and there is no automatic transmitting repetition.

This happens e. g. if one or more bus participants are transmitting simultaneously. If the participant that is transmitting a message with set `uMsgInfo.Bits.ssm` bit loses the bus assignment (arbitration), the message is rejected and further transmitting is not attempted.

The functionality is exclusively available if bit `CAN_FEATURE_SINGLESHOT` in field *dwFeatures* of structure *CANCAPABILITIES* resp. *CANCAPABILITIES2* is set.

### Transmitting Messages with High Priority

Transmitting messages with set `uMsgInfo.Bits.hpm` bit are registered by the controller in a controller specific transmitting buffer that takes precedence over messages in the standard transmitting buffer and primarily transmits.

The functionality is exclusively available if the bit `CAN_FEATURE_HIGHPRIOR` in field *dwFeatures* of structure *CANCAPABILITIES* resp. *CANCAPABILITIES2* is set. If the bit is used observe that messages that are already in the transmitting FIFO can not be overtaken. The

functionality is of minor impact resp. can only be sensibly used if the controller is opened exclusive and the transmitting FIFO is empty before addressing a message with set bit `uMsgInfo.Bits.hpm`.

#### Transmitting Messages Confirmed (Self-Reception)

Transmitting messages with set `uMsgInfo.Bits.srr` bit are, after they are transmitted successfully from the controller to the bus, automatically received again and forwarded to all active message channels by the distributor. Each message channel can decide on its own how to handle this self reception messages.

**Message Channel Type `ICanChannel`**

- Write all self reception messages in the Receiving FIFO. Irrespective whether the message is transmitted on this or another channel on the same controller.
- Each active channel receives each transmitted self reception message (`uMsgInfo.Bits.srr` bit is always set).

**Message Channel Type `ICanChannel2`**

- If a self reception message is received the channel verifies if the message originates from itself.
- If the message originates from itself the message is written in the receiving FIFO with set *srr* bit irrespective of the current settings of the filter.
- If the messages originates from another channel of the same controller the following processing is dependent on the current settings of the filter:
  - If while calling the function `ICanChannel2::Initialize` the operating mode is combined with the constant `CAN_FILTER_SRRA` the channel treats all self reception messages of all channels of the same controller as if they come from the same bus. The message, as it passes the message filter of the channel, is written to the receiving FIFO with deleted *srr* bit. For the application it seems as if the message has been transmitted from another controller.
  - If the message filter is initialized without the constant `CAN_FILTER_SRRA` the channel exclusively receives self reception messages that are transmitted by itself. Self reception messages transmitted via other channels are rejected. Messages that are transmitted via a channel with deleted *srr* bit are invisible for other channels on the same controller.

### 5.2.3 Control Unit

The control unit provides the following functions via the interface `ICanControl`:

- configuration and control of the CAN controller
- configuration of the transmitting features of the CAN controller
- configuration of CAN message filters
- starting and stopping of data transmitting
- requesting of current operating state

To stop several competing applications from gaining control of the controller, the control unit can exclusively be opened once by one application at a time.

### Opening the Interface

Open with function `IBalObject::OpenSocket`.

- In parameter *riid* specify value `IID_ICanControl` resp. `IID_ICanControl2`.
  - If the function returns an error code like *access denied* the component is already used by another program.
- With `Release` close control unit and release for access by other applications.

*If other interfaces of the controller are opened when the controller is closed, the current settings remain, i. e. a started CAN controller is not stopped automatically with calling `Release` as long as an additional message channel or the cyclic transmitting list is opened.*

## Controller States

The control unit resp. the CAN controller is always in one of the following states:

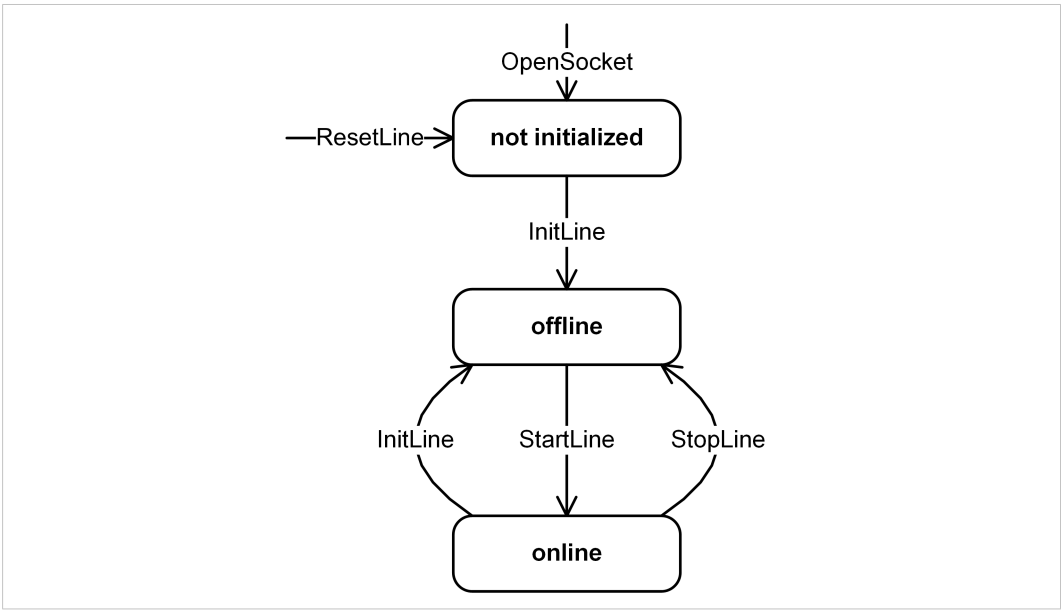


**Fig. 20 Controller states**

## Initializing the Controller

After the first opening of the control unit via the interface *ICanControl* or *ICanControl2* the controller is in a non-initialized state.

- To leave non-initialized state, call function `ICanControl::InitLine` resp. with extended functionality `ICanControl2::InitLine`.
  - Controller is in state *offline*.
- Specify system mode and transmission rate with function `ICanControl::InitLine` resp. `ICanControl2::InitLine`.
- Functions require in parameter *pInitParam* pointer to a structure of *CANINITLINE* resp. *CANINITLINE2* with initialized structure.
- Specify operating mode in field *bOpMode*.
- If a controller with extended functionality is used, activate operation mode with field *bExMode*.

- ▶ Specify bit rate (see *Specifying the Bit Rate, p. 34*).
- ▶ Call function.
  - ▷ Controller is initialized with specified values.

Controllers with extended functionality don't have message filters with adjustable operation mode. The default and reset values for this filter operating mode are done separately for the 11 and 29 bit filter in the fields *bSFMode* and *bEFMode* (for further information see *Message Filter, p. 40*).

### Starting the Controller

To start CAN controller and data transmission between controller and bus:

- ▶ Make sure that CAN controller is initialized (see *Initializing the Controller, p. 32*).
- ▶ Call function `StartLine`.
  - ▷ Control unit is in state *online*.
  - ▷ Incoming messages are forwarded to all active message channels.
  - ▷ Transmitting messages are transferred to the bus.

After successful start of the controller the control unit transmits an information message to all active message channels. Field *dwMsgId* of the message contains the value `CAN_MSGID_INFO`, the field *abData[0]* the value `CAN_INFO_START` and the field *dwTime* the relative starting time.

### Stopping (resp. Resetting) the Controller

- ▶ Call function `StopLine`.
  - ▷ Controller is in state `offline`.
  - ▷ Data transfer between controller and bus is stopped.
  - ▷ Transport of messages between controller and all active message channels is stopped.
  - ▷ In case of an ongoing data transfer of the controller the function waits until the message is transmitted completely over the bus, before the message transmission is stopped. No faulty telegram is on the bus.

  or

- ▶ Call function `ResetLine`.
  - ▷ Controller is in state *not initialized*.
  - ▷ Controller hardware and set message filters are reset to the predefined initial state.
  - ▷ Filter lists are deleted.
  - ▷ Transport of messages between controller and all active message channels is stopped.

*After calling the function `ResetLine` a faulty message telegram on the bus is possible, if a not completely transferred message is in the transmitting buffer of the controller.*

If `StopLine` or `ResetLine` are called the control unit transmits an information message to all active channels. The field *dwMsgId* of the message contains the value `CAN_MSGID_INFO`, the field *abData[0]* the value `CAN_INFO_STOP` resp. `CAN_INFO_RESET` and the field *dwTime* the

value 0. Neither `ResetLine` nor `StopLine` delete the content of the transmitting and receiving FIFO of the message channels.

## Specifying the Bit Rate

### Structure `CANINITLINE`

- Specify with fields *bBtReg0* and *bBtReg1*.

The values of the fields *bBtReg0* and *bBtReg1* correspond to the values of the bus timing register BTR0 and BTR1 of Philips SJA1000 CAN Controller with a clock frequency of 16 MHz.

**Values for bit timing register BTR0 and BTR1 resp. therefore defined constants of often used bit rates:**

| Bit rate (KBit) | Predefined constants for BTR0, BTR1 | BTR0 | BTR1 |
|---|---|---|---|
| 5 | CAN_BT0_5KB, CAN_BT1_5KB | 0x3F | 0x7F |
| 10 | CAN_BT0_10KB, CAN_BT1_10KB | 0x31 | 0x1C |
| 20 | CAN_BT0_20KB, CAN_BT1_20KB | 0x18 | 0x1C |
| 50 | CAN_BT0_50KB, CAN_BT1_50KB | 0x09 | 0x1C |
| 100 | CAN_BT0_100KB, CAN_BT1_100KB | 0x04 | 0x1C |
| 125 | CAN_BT0_125KB, CAN_BT1_125KB | 0x03 | 0x1C |
| 250 | CAN_BT0_250KB, CAN_BT1_250KB | 0x01 | 0x1C |
| 500 | CAN_BT0_500KB, CAN_BT1_500KB | 0x00 | 0x1C |
| 800 | CAN_BT0_800KB, CAN_BT1_800KB | 0x00 | 0x16 |
| 1000 | CAN_BT0_1000KB, CAN_BT1_1000KB | 0x00 | 0x14 |

For further information about BTR0 and BTR1 and their functionality see data sheet of Philips SJA1000.

### Structure `CANINITLINE2`

Allows a more independent setting of the bit rate and the sampling time.

- Specify with fields *sBtpSdr* and *sBtpFdr*.

The field *sBtpSdr* defines the bit timing parameters for the nominal bit rate resp. the bit rate during the arbitration period. If the controller supports fast data transfer and it is activated with the extended operating mode `CAN_EXMODE_FASTDATA` the field *sBtpFdr* determines the bit timing parameter for the fast data rate.

## Time Periods

The field *dwMode* of structure *`CANBTP`* determines how the further fields *dwBPS*, *wTS1*, *wTS2*, *wSJW* and *wTDO* are interpreted.

If the bit `CAN_BTMODE_RAW` in *dwMode* is set, all other fields contain controller specific values (see *Mode `CAN_BTMODE_RAW`, p. 38*).

If the bit `CAN_BTMODE_RAW` is not set, the field *dwBPS* contains the desired bit rate in bits per second. The fields *wTS1* and *wTS2* divide a bit in two time periods before and after the sample time resp. the time when the controller determines the value of the bit (Sample Point).

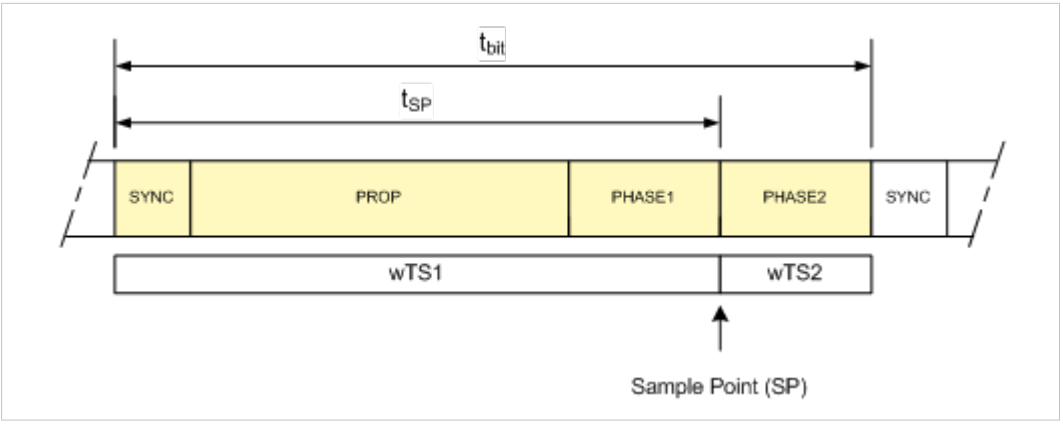


**Fig. 21 Segmentation of a bit in different time periods**

The amount of the fields *wTS1* and *wTS2* is the length of a bit $t_{bit}$ and determines the number of time quanta in which a bit is divided:

- Number of time quanta per bit: $Q_{bit}$ = *wTS1* + *wTS2*

With the highest possible values for *wTS1* and *wTS2* a bit can be divided in up to 65535 +65535=131070 time quanta.

The number of time quanta per bit $Q_{bit}$ determines together with the selected bit rate the length of an individual time quantum $t_Q$ resp. its resolution:

- $t_Q = t_{bit} / Q_{bit} = 1 / (\text{bit rate} * Q_{bit})$

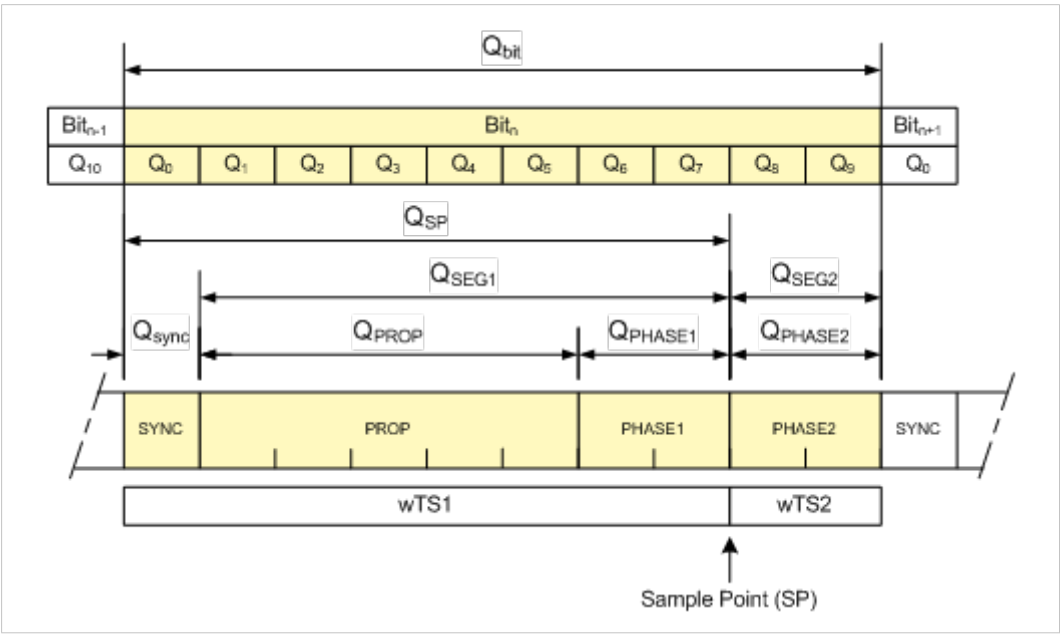


**Fig. 22 Segmentation of a bit in time quanta and segments**

The figure shows exemplary a segmentation in 10 time quanta. *wTS1*=8 and *wTS2*=2 is selected, with that the sample point is determined to 8/10 resp. 80% of a bit time.

### Segments

According to the CAN specification a bit is divided into the segments *SYNC*, *PROP* plus *PHASE1* and *PHASE2*. The beginning of a bit is expected in segment *SYNC*. The segment *PROP* serves as compensation to the cable and component caused delays. The segments *PHASE1* resp. *PHASE2* serve as compensation for the phase errors, that are caused e. g. by oscillation tolerances.

If the following recessive dominant signal flank does not occur during *SYNC* a post scoring by the controller follows. The primary scoring of the controller to the beginning of a message is always done with the starting bit of a message.

Post scoring

- Segments *PHASE1* resp. *PHASE2* are lengthened or shortened depending on the length of the phase.
- Number of time quanta ($Q_{SJW}$) necessary to compensate the phase errors is called synchronization jump width (SJW) and specified in the field *wSJW*.
- The time shifting $t_{SJW}$ that can be compensated with that can be calculated with:

  $t_{SJW} = t_Q * wSJW$

### Synchronization Jump Width

A post scoring reduces the phase error maximally by the set synchronization jump width. If the error is not completely compensated by that a remaining phase error occurs. Because a post scoring is only done after a recessive dominant signal flank in error-free transmission it lasts maximally 10 bit times (5 dominate bits followed by 5 recessive bits) until a new recessive dominate signal flank occurs. In this 10 bit times remaining phase errors can summarize and have to be corrected by the set synchronization jump width. This results in the following condition:

#### Condition 1

- $2 * \Delta F * (10 * t_{bit}) \leq t_{SJW}$ (1)

In case of an error on the bus it is possible that up to 6 bits are transmitted in a row and a stuff error occurs. The controller that recognizes that at first (and is error active) then transmits an error telegram, that consists of 6 bits. Other controllers on the bus recognize this as stuff error and echo also an error telegram. On the bus a row of up to 13 dominate bits occur. In this case the next post scoring can earliest be done after 13 bit times. In this time also reset phase errors summarize. The compensation by the set synchronization jump width must be possible. This results in the second condition:

#### Condition 2

- $2 * \Delta F * (13 * t_{bit} - t_{PHASE2}) \leq \min(t_{PHASE1}, t_{PHASE2})$ (2)

**Time Quanta**

Observe the following when specifying:

- Number of time quanta inside of segment *PROP* ($Q_{PROP}$): choose according to the cable and component caused delays.
- The minimum number of time quanta in *PHASE1* ($Q_{PHASE1}$) is determined by the number of time quanta ($Q_{SJW}$) that are needed to compensate phase errors: must be higher than or equal the synchronization jump width.
- The minimum number of time quanta in *PHASE2* ($Q_{PHASE2}$) is determined by the synchronization jump width: consider processing time of the controller.
- Information processing time (IPT) begins with the sampling time and requires a certain amount of time quanta ($Q_{IPT}$): $Q_{PHASE2}$: must be higher than or equal $Q_{IPT} + Q_{SJW}$.

The number of time quanta in the first segment until the sampling point ($Q_{SP}$) is equal to the sum of all time quanta in segments *SYNC*, *PROP* and *PHASE1* and is determined with the value *wTS1*. The number of time quanta in the second segment after the sampling point ($Q_{SEG2}$) is equal to the sum of all time quanta in segments *PHASE2* and is determined with the value *wTS2*.

The length of a time quantum $t_Q$ also determines the value of *wSJW* and therefore is important for the post scoring resp. the compensation of phase errors.

In example *Segmentation of a bit in time quanta and segments, p. 35* with *wTS1*=8, *wTS2*=2 and $Q_{bit}$=10 the sampling point is 80 %. The resolution of a time quantum is 1/10 resp. 10 % of a bit time. If the value 1 is specified for *wSJW* the sampling point of a phase correction is shifted about ± 10 % of a bit time. Higher values than 1 are not allowed for *wSJW* in this example, because sampling errors could occur.

With a high number of time quanta phase errors can be corrected more precisely because the length of a time quanta is shortened by this.

A sampling point of 80 % can e. g. be reached if for *wTS1* the value 80 and for *wTS2* the value 20 ($Q_{bit}$=100) is specified. The resolution of a time quantum then is 1 % of a bit time. In this case with *wSJW*=1 phase errors up to ±1 % of a bit time can be corrected.

The resolution of a time quantum theoretically can be shortened down to 1/131070 ≈7,63*$10^{-6}$ resp. 7,63 ppm. As the values for the individual segments have to be converted to the hardware specific register, the limits are higher. Regarding the SJA1000 with 16 MHz clock frequency the maximum possible value for $Q_{bit}$ is 25 (1+16+8) and therefore the minimum possible resolution is 1/25 resp. 4 % of a bit time. With higher bit times the number of time quanta is reduced and is for 1 Mbit only 8, that results in a resolution of 1/8 resp. 12,5 % of a bit time.

▶ To get information about the value ranges of the individual segments supported by the hardware call function `ICanSocket2::GetCapabilities`.

  ▷ Fields *sSdrRangeMin*, *sSdrRangeMax* resp. *sFdrRangeMin* and *sFdrRangeMax* of structure *`CANCAPABILITIES2`* indicated with calling of the function contain hardware specific minimum and maximum values.

### Mode `CAN_BTMODE_RAW`

- Field *dwBPS* contains the value for the frequency divider ($N_P$) in the CAN controller (instead of bit rate).
- Field *wTS1* contains segments *PROP* and *PHASE1* (instead of time segments *SYNC*, *PROP* and *PHASE1*)
- Number of time quanta in segment SYNC is fixed and always one
- Assignments of fields *wTS2* and *wSJW* remain the same.

The following figure shows the assignment of the fields to the individual segments and the generation of the frequency for the bit processor and the resulting times.

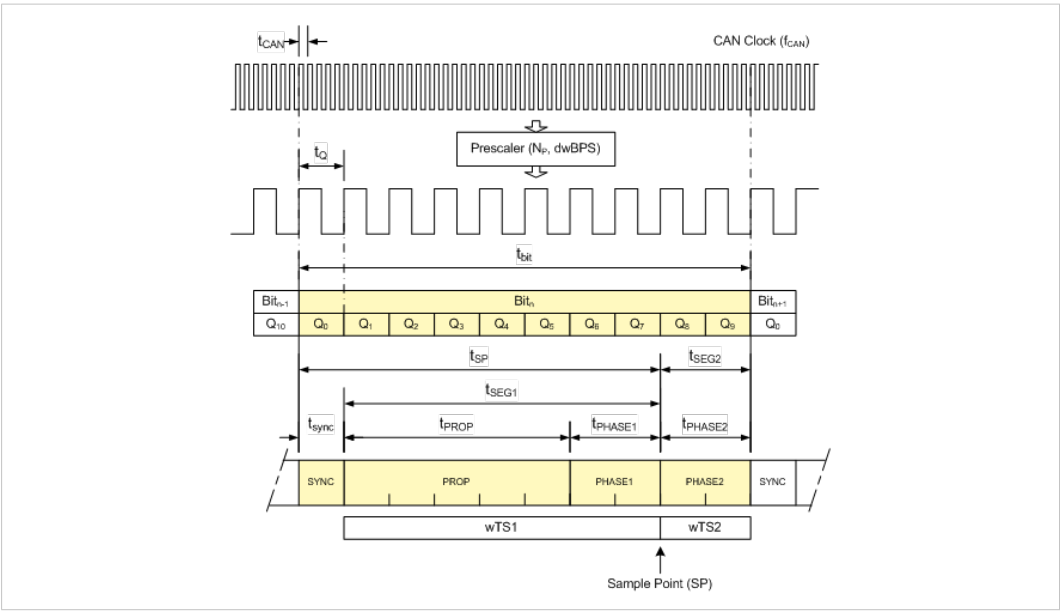


**Fig. 23 Clock generator for the bit processor in the CAN controller**

The field *dwCanClkFreq* of structure `CANCAPABILITIES2` returns the frequency of the clock generator $f_{CAN}$ for the bit processor. This system frequency is divided by an adjustable frequency divider (prescaler). The output of the frequency divider determines the length of a time quantum $t_Q$:

- $t_Q = t_{CAN} * N_P = N_P / f_{CAN}$

The bit time $t_{bit}$ is an integral multiple of a time quantum $t_Q$ and is calculated by:

- $t_{bit} = t_Q * Q_{bit} = Q_{bit} * N_P / f_{CAN}$

The bit rate $fb_{it}$ is calculated by:

- $f_{bit} = 1/t_{bit} = f_{CAN} / (Q_{bit} * N_P)$

To specify the bit rate $f_{bit}$ with predefined frequency $f_{CAN}$ the prescaler $N_P$ and the number of time quanta $Q_{bit}$ must be specified.

A possibility to specify the parameters is e. g. to begin with the maximally possible time quanta max($Q_{bit}$) and to determine with that the value for the prescaler $N_P$.

- $N_P = f_{CAN} / (f_{bit} * Q_{bit})$

If no appropriate value results for $N_P$, the number of time quanta is reduced by 1 and a new value for $N_P$ is calculated. This is proceeded until either a appropriate value for $N_P$ is found or the value has fallen below the minimal amount of time quanta min($Q_{bit}$).

If the value has fallen below the minimal amount of time quanta there is no solution for the demanded bit rate. In the other case with the found values for $N_P$ and $Q_{bit}$ the values for *wTS1*, *wTS2* and *wSJW* can be determined in the following way:

- Calculate time of a time quantum:

  $t_Q = N_P / f_{CAN}$

- Determine amount of time quanta $Q_{SJW}$ required for the post scoring with *Condition 1* and *Condition 2*.

---

*The value is dependent on the oscillation tolerance ΔF. The oscillation tolerance of IXXAT CAN interfaces is normally smaller than 0,1 % but in this case the greatest oscillation tolerance of all nodes existing in the network must be considered.*

---

- To calculate the number of required time quanta for the segment *PROP* ($Q_{PROP}$) divide the cable and component caused delays $t_{PROP}$ by the length of a time quantum $t_Q$ and round up to the next integral number:

  $Q_{PROP} = round\_up(t_{PROP} / t_Q)$

- Calculate the total number of time quanta for the phase compensation $Q_{PHASE}$:

  $Q_{PHASE} = Q_{bit} – (Q_{SYNC} + Q_{PROP}) = Q_{bit} - 1 - Q_{PROP}$

  $Q_{PHASE1}$ and $Q_{PHASE2}$ are calculated by a integral division of $Q_{PHASE}$ by 2 and the remaining. In case of an uneven value for $Q_{PHASE}$ the smaller part is assigned to $Q_{PHASE1}$ and the greater to $Q_{PHASE2}$.

  $Q_{PHASE1} = INT(Q_{PHASE}/2)$

  $Q_{PHASE2} = INT(Q_{PHASE}/2) + MOD(Q_{PHASE}/2)$

  If $Q_{PHASE1}$ is less than $Q_{SJW}$ or $Q_{PHASE2}$ is less than $Q_{SJW} + Q_{IPT}$ there is no solution for the requested bit rate. The minimum value of *sSdrRangeMin.wTS2* resp. *sFdrRangeMin.wTS2* corresponds to $Q_{IPT}$.

Further information about the setting of the bit rate see CAN resp. CAN FD specification and in the CAN FD white paper of Bosch both in chapter “Bit Timing Requirements”.

For information about the calculation of the parameter for the fast bit rate see CAN FD specification.

### Determine the Bit Rate Used in the Network

If the CAN connector is connected to a running network with unknown bit rate the current bit rate can be determined.

- Use CAN controller in listen only mode.
- Make sure that two further bus participants are transmitting messages.
- Call function `DetectBaud`.
  - ▷ Field *bIndex* of structure *CANBTRTABLE* contains table index of the found bus timing values.
- The determined bus timing values can be used when calling the function `InitLine`.

Function `DetectBaud` requires a pointer to the initialized structure of type *CANBTRTABLE*, that contains a predefined set of bit timing values. The extended version requires a pointer to the initialized structure of type *CANBTPTABLE*, that contains a predefined set of bit timing values for the needed standard resp. nominal bit rates and eventually also for the according fast data bit rate.

**Example of Usage of the Function to Adjust CAN Controller Automatically to the Bit Rate of the Running System:**

```
BOOL AutoInitLine( ICanControl* pControl )
{
  static UINT8 abBtr0[] =
  {
    CAN_BT0_10KB, CAN_BT0_20KB, CAN_BT0_50KB,
    CAN_BT0_100KB, CAN_BT0_125KB, CAN_BT0_250KB,
    CAN_BT0_500KB, CAN_BT0_800KB, CAN_BT0_1000KB
  };

  static UINT8 abBtr1[] =
  {
    CAN_BT1_10KB, CAN_BT1_20KB, CAN_BT1_50KB,
    CAN_BT1_100KB, CAN_BT1_125KB, CAN_BT1_250KB,
    CAN_BT1_500KB, CAN_BT1_800KB, CAN_BT1_1000KB
  };
  HRESULT hResult;
  CANBTRTABLE sBtrTab;

 // determine bit rate
 sBtrTab.bCount = sizeof(abBtr0) / sizeof(abBtr0[0]);
 sBtrTab.bIndex = 0xFF;
 memcpy(sBtrTab.abBtr0, abBtr0, sizeof(abBtr0));
 memcpy(sBtrTab.abBtr1, abBtr1, sizeof(abBtr1));

 hResult = pControl->DetectBaud(10000, &sBtrTab);
 if (hResult == VCI_OK)
 {
    CANINITLINE sInitParam;

    sInitParam. bOpMode = CAN_OPMODE_STANDARD|CAN_OPMODE_ERRFRAME;
    sInitParam. bReserved = 0;
    sInitParam. bBtReg0 = sBtrTab.abBtr0[sBtrTab.bIndex];
    sInitParam. bBtReg1 = sBtrTab.abBtr1[sBtrTab.bIndex];

    hResult = pControl->InitLine(&sInitParam);
  }

  return( hResult == VCI_OK );
}
```

### 5.2.4 Message Filter

All control units and message channels with expanded functionality have a two-level message filter to filter the data messages received from the bus. Information, error and status messages that are transmitted by the controller resp. the control unit always can pass unhindered.

The data messages are exclusively filtered by the ID in field *dwMsgId* of structure *CANMSG* resp. *CANMSG2*. The other fields of a message, including the data bytes in field *abData* are not considered.

## Operating Modes

Message filters can be ran in different operation modes:

- Blocking mode (`CAN_FILTER_LOCK`):

  Filter blocks all messages of type CAN_MSGTYPE_DATA, independent of the ID. Used e. g. if an application is only interested in information, error or status messages.

- Passing mode (`CAN_FILTER_PASS`):

  Filter is completely opened and all data messages can pass. Default operation mode in case of usage of interface *`ICanChannel`*.

- Inclusive filtering (`CAN_FILTER_INCL`):

  All data messages with an ID either released in the acceptance filter or registered in the filter list can pass the filter (e. i. all registered IDs). Default operating mode in case of usage of interface *`ICanControl`*.

- Exclusive filtering (`CAN_FILTER_EXCL`):

  All data messages with an ID either released in the acceptance filter or registered in the filter list are blocked by the filter (e. i. all registered IDs).

In case that the interface *`ICanControl`* is used the operating mode of the filter can not be changed and is preset to `CAN_FILTER_INCL`. In case that the interface *`ICanControl2`* resp. *`ICanChannel2`* is used the operation mode can be set to one of the above stated modes with the function `SetFilterMode`.

*To ask for the operating mode of the filter call function `GetFilterMode`.*

### Inclusive and Exclusive Operating Mode

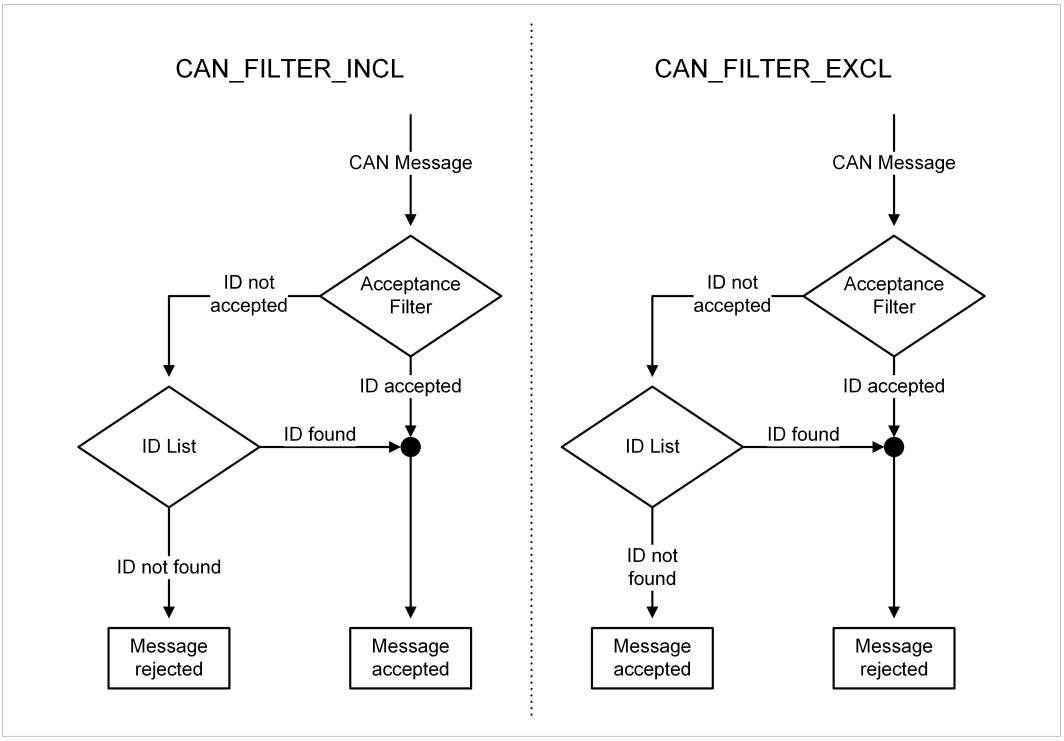


**Fig. 24 Filtering mechanism inclusive and exclusive operating mode**

The first filter level consists of an acceptance filter that compares the ID of a received message with a binary bit sample. If the ID correlates with the set bit sample the ID is accepted. In case of inclusive operating mode the message is accepted. In case of exclusive operating mode the message is immediately rejected.

If the first filter level does not accept the ID it is forwarded to the second filter level. The second filter list consists of a list with registered message IDs. If the ID of the received message is equal to an ID in the list, the message is accepted in case of inclusive filtering and rejected in case of exclusive filtering.

### Filter Chain

Each message channel is connected to a controller either directly or indirectly via a distributor (see *Message Channels, p. 23*). If a filter is used both with the controller and with the message channel a multi-level filter chain is formed. Messages that are filtered out by the controller are invisible for the down-streamed channels.

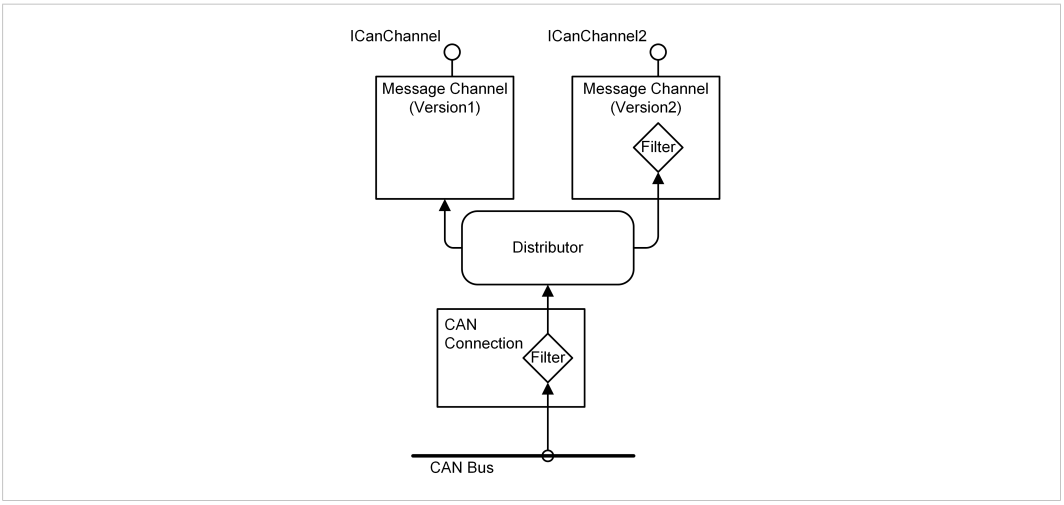


**Fig. 25 Filter Chain**

## Setting the Filter

Control units and message channels have separated and independent filters for 11 bit and 29 bit IDs. Messages with 11 bit ID are filtered by the 11 bit filter and messages with 29 bit ID by the 29 bit filter.

To distinguish between 11 and 29 bit filter all stated functions have the parameter *bSelect*.

*Changes of the filters during operation are not possible.*

- Make sure that control unit is *offline* resp. that the message channel is inactive.

If the interfaces `ICanControl2` resp. `ICanChannel2` are used the operating mode of the filter is preset during the initialization of the component. The specified value serves simultaneously as default value for the function `ICanControl2::ResetLine`.

- To set the filter after initialization, call function `SetFilterMode`.
- To set the acceptance filter call function `SetAccFilter`.
- Specify filter list with functions `AddFilterIds` and `RemFilterIds`.
- In parameter *bSelect* select 11 or 29 bit filter.

- In parameters *dwCode* and *dwMask* specify two bit samples that determine one or more IDs that must be registered.
  - Value of *dwCode* determines the bit sample of the ID.
  - *dwMask* determines which bits in *dwCode* are valid and used for the comparison.

If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not used for the comparison. But if it has the value 1 it is relevant for the comparison.

In case of the 11 bit filter exclusively the lower 12 bits are used. In case of the 29 bit filter the bits 0 to 29 are used. Bit 0 of every value defines the value of the remote transmission request bit (RTR). All other bits of the 32 bit value must be set to 0 before one of the functions is called.

Correlation between the bits in the parameter *dwCode* and *dwMask* and the bits in the message ID:

11 bit filter

| Bit | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ID10 | ID9 | ID8 | ID7 | ID6 | ID5 | ID4 | ID3 | ID2 | ID1 | ID0 | RTR |

29 bit filter

| Bit | 29 | 28 | 27 | 26 | 25 | ... | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ID28 | ID27 | ID26 | ID25 | ID24 | ... | ID4 | ID3 | ID2 | ID1 | ID0 | RTR |

The bits 1 to 11 resp. 1 to 29 of the values in *dwCode* resp. *dwMask* corresponds to the bits 0 to 10 resp. 0 to 28 of the ID of a CAN message.

The following example shows the values that must be used for *dwCode* and *dwMask* to register message IDs in the range of 100 h to 103 h (of which also the RTR bit must be 0) in the filter:

| | |
|---|---|
| *dwCode* | 001 0000 0000 0 |
| *dwMask* | 111 1111 1100 1 |
| VAlid IDs: | 001 0000 00xx 0 |
| ID 100h, RTR = 0: | 001 0000 0000 0 |
| ID 101h, RTR = 0: | 001 0000 0001 0 |
| ID 102h, RTR = 0: | 001 0000 0010 0 |
| ID 103h, RTR = 0: | 001 0000 0011 0 |

The example shows that with a simple acceptance filter only individual IDs or groups of IDs can be released. If the desired identifiers don't correspond with a certain bit sample, a second filter level, a list with IDs, must be used. The amount of IDs a list can receive can be configured. Normally the 11 bit ID list is configured in order that all 2048 possible IDs have enough space.

- Register individual or groups of IDs with function `AddFilterIds`.
- If necessary remove from list with function `RemFilterIds`.

The parameters *dwCode* and *dwMask* have the same format as showed above.

If `AddFilterIds` is called with same values as in the above example the function enters the identifier 100 h to 103 h to the list.

- To register exclusively an individual ID in the list, specify the desired ID (including RTR bit) in *dwCode* and in *dwMask* the value `0xFFF` (11 bit ID) resp. `0x3FFFFFFF` (29 bit ID).

- To disable acceptance filter completely, when calling function `SetAccFilter` enter in *dwCode* the value `CAN_ACC_CODE_NONE` and in *dwMask* the value `CAN_ACC_MASK_NONE`.
  - ▷ Filtering is exclusively done with ID list.

  or
- Configure acceptance filter with the values `CAN_ACC_CODE_ALL` and `CAN_ACC_MASK_ALL`.
  - ▷ Acceptance filter accepts all IDs and ID list is ineffective.

### 5.2.5 Cyclic Transmitting List

With the optionally provided transmitting list of the controller up to 16 messages can be transmitted cyclically. The access to this list is limited to one application and therefore can not be used by several programs simultaneously.

Open interface with function `IBalObject::OpenSocket`.

- In parameter *riid* enter value `IID_ICanScheduler`.
- In case of controller with extended functionality in parameter *riid* enter value `IID_ICanScheduler2`.
  - ▷ If function returns an error code respective *access denied* the transmitting list is already under control of another program and can not be opened again.
- To allow access for other applications, close open transmitting list with function `Release`.
- Add message objects with `ICanScheduler::AddMessage` resp. in case of controller with extended functionality with `ICanScheduler2::AddMessage` to the list. Functions expect pointer to an initialized object of type `CANCYCLICTXMSG` resp. `CANCYCLICTXMSG2`.
  - ▷ In case of successful execution both functions return the list index of the newly added transmitting object.

One controller exclusively supports one transmitting list. Irrespective if the functions of the interface `ICanScheduler` or `ICanScheduler2` are used, the list index always refers to the same list. As the interfaces are exclusively different regarding the data type of the transmitted messages, whereas the functionality is identical, only the functionalities of the interface `ICanScheduler` is described hereafter.

- Specify cycle time of a message in number of ticks in field *wCycleTime* of structure `CANCYCLICTXMSG` or `CANCYCLICTXMSG2`.
- Make sure that the specified value is higher than 0 but less than or equal the value in `CANCAPABILITIES` field *dwCmsMaxTicks* of one of the structures `CANCAPABILITIES` resp. `CANCAPABILITIES2`.
- Calculate length of a tick of the cycle timer of the transmitting list ($t_{cycle}$) with values in fields *dwClockFreq* and *dwCmsDivisor* (see `CANCAPABILITIES`), resp. in case of extended functionality with values in fields *dwCmsClockFreq* and *dwCmsDivisor* (see `CANCAPABILITIES2`) with the following formula:

  $t_{cycle}$ [s] = (*dwCmsDivisor* / *dwClockFreq*)

  or

  $t_{cycle}$ [s] = (*dwCmsDivisor* / *dwCmsClockFreq*)

The transmitting task of the cyclic transmitting list divides the available time in individual segments resp. time frames. The length of a time frame is exactly the same as the length of a tick of the cyclic timer ($t_{cycle}$).

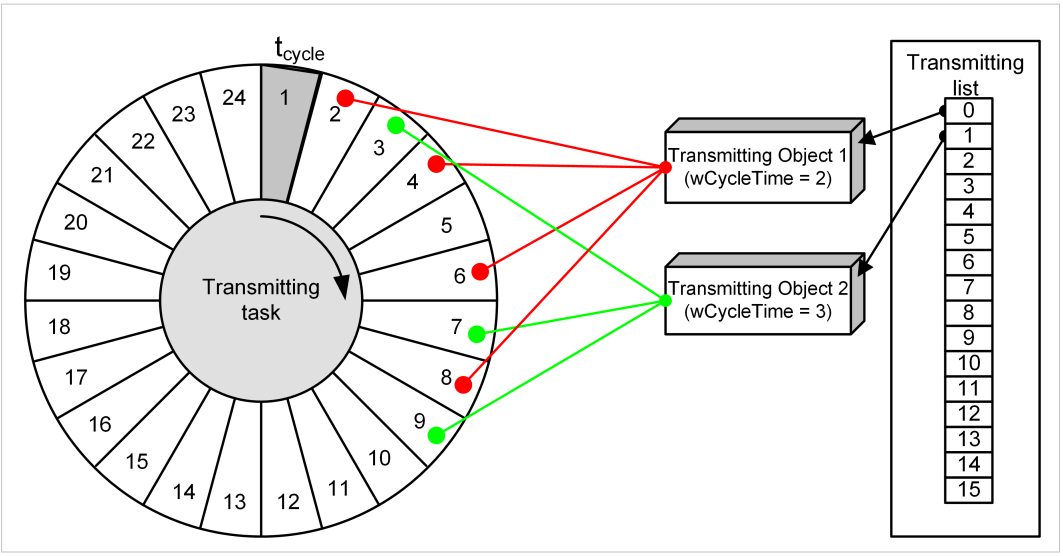


**Fig. 26 Transmitting task of the cyclic transmitting list with 24 time frames**

The number of the time frames supported by the transmitting task is equal to the value in field *dwCmsMaxTicks* of structure `CANCAPABILITIES` resp. `CANCAPABILITIES2`. *dwCmsMaxTicks* contains the value 24.

The transmitting task can transmit exclusively one message per tick, e. i. exclusively one transmitting object can be matched to a time frame. If the transmitting object is created with a cyclic time of 1 all time frames are occupied and no other objects can be created. The more transmitting objects are created, the larger their cyclic time must be selected. The rule is: The total of all 1/*wCycleTime* has to be less than 1.

In the example a message shall be transmitted every 2 ticks and a further message every 3 ticks, this amounts 1/2 + 1/3 = 5/6 = 0,833 and therefore a valid value.

If the transmitting object 1 is created with a *wCycleTime* of 2 the time frames 2, 4, 6, 8, etc. are occupied. If the second transmitting object is created with a *wCycleTime* of 3, it leads to a collision in the time frames 6, 12, 18, etc. because these time frames are already occupied by the transmitting object 1.

Collisions are resolved in shifting the new transmitting object in the respectively next free time frame. The transmitting object of the example above then occupies the time frames 3, 7, 9, 13, 19, etc. The cyclic time of the second object therefore is not met exactly and in this case leads to a inaccuracy of +1 tick.

The temporal accuracy of the transmitting of the objects is heavily depending on the message load on the bus. With increasing load the transmitting time gets more and more imprecise. The general rule is that the accuracy decreases with increasing bus load, smaller cycle times and increasing number of transmitting objects.

The field *bIncrMode* of structure `CANCYCLICTXMSG` or `CANCYCLICTXMSG2` determines if certain parts of a message are automatically incremented after transmitting or if they remain unmodified.

If in *bIncrMode* CAN_CTXMSG_INC_NO is specified, the content of the message remains unmodified. With the value CAN_CTXMSG_INC_ID the field *dwMsgId* of the message automatically increases by 1 after every transmission. If field *dwMsgId* reaches the value 2048 (11 bit ID) resp. 536.870.912 (29 bit ID) an overflow automatically takes place.

With the values CAN_CTXMSG_INC_8 resp. CAN_CTXMSG_INC_16 an individual 8 bit resp.16 bit value is increment in the data field *abData[]* after each transmission. The field *bByteIndex* of structure *CANCYCLICTXMSG* or *CANCYCLICTXMSG2* determines the starting position of the data value.

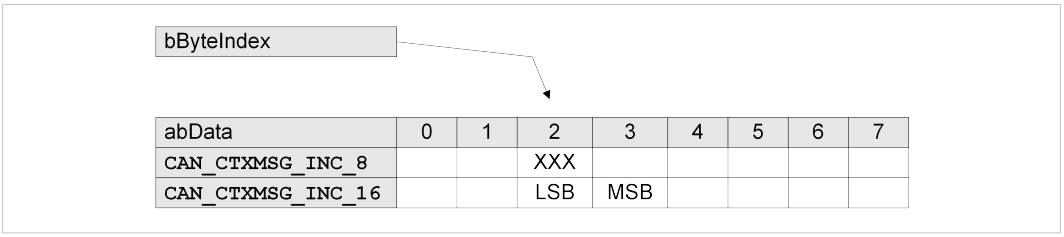


**Fig. 27 Auto increment of data fields**

Regarding 16 bit values, the low byte (LSB) is located in field *abData[bByteIndex]* and the high byte (MSB) in field *abData[bByteIndex+1]*. If the value 255 (8 bit) resp. 65535 (16 bit) is reached, an overflow to 0 takes place.

- If necessary, remove transmitting object from list with function *RemMessage*. The function expects the list index of the object to remove returned by *AddMessage*.
- To transmit newly created transmitting object, call function *StartMessage*.
- If necessary, stop transmitting with function *StopMessage*.

The current status of the transmitting task and of all created transmitting objects is returned by the function *GetStatus*. The required memory is provided as structure of type *CANSCHEDULERSTATUS* by the application. After successful execution of the function the fields *bTaskStat* and *abMsgStat* contain the state of the transmitting list and the transmitting objects.

To determine the state of an individual transmitting object the list index returned by function *AddMessage* is used as index in the table *abMsgStat* i. e. *abMsgStat[Index]* contains the state of the transmitting object of the specified index.

The transmitting task is deactivated after opening the transmitting list. The transmitting task does not transmit any message in deactivated state, even if the list is created and contains started transmitting objects.

- To start all transmitting objects simultaneously, configure all transmitting objects with function *StartMessage*.
- Start transmitting task with function *Resume*.
- To deactivate a transmitting task call function *Suspend*.
- To reset a transmitting task call function *Reset*.
  - ▷ Transmitting task is stopped.
  - ▷ All registered transmitting objects are removed from the specified cyclic transmitting list.

## 5.3 LIN-Controller

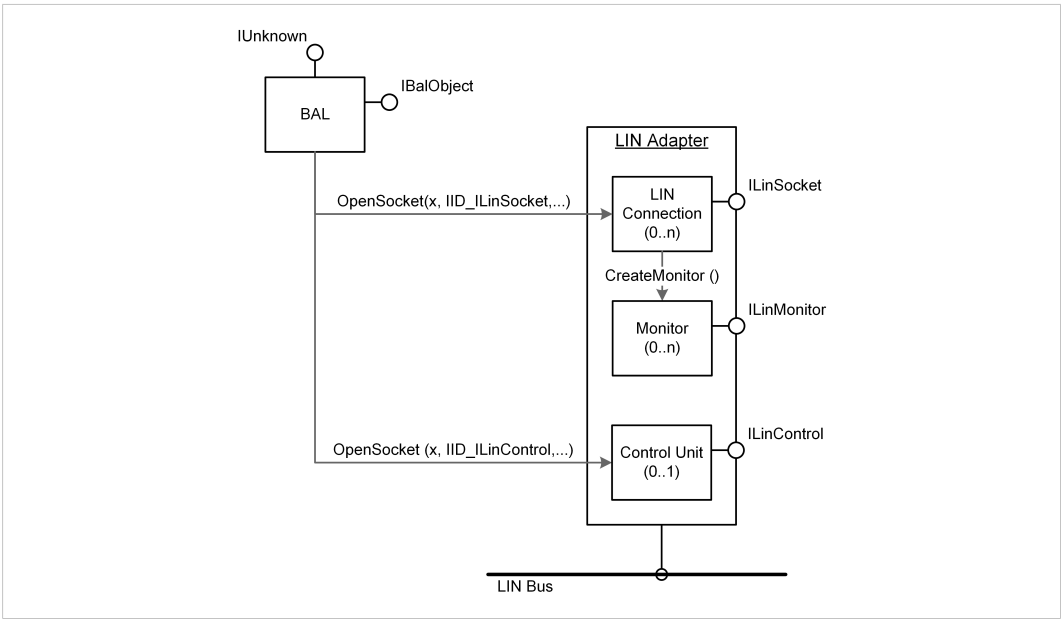


**Fig. 28 Components LIN controller**

- access to individual components via function `IBalObject::OpenSocket` (required IDs see figure above)
- access to individual sub components via interfaces `ILinControl` or `ILinMonitor` (see *Message Monitors, p. 48*)

`ILinSocket` (see *Socket Interface, p. 47*) provides the following functions:

- requesting of LIN controller functionalities and state
- creating of message monitors, that are required for receiving messages

`ILinControl` (see *Control Unit, p. 51*) provides the following functions:

- configuration of LIN controller
- configuration of transmitting features
- requesting of current controller status

### 5.3.1 Socket Interface

The interface `ILinSocket` is not subjected to any access restrictions and can be opened by multiple applications simultaneously. Controlling via this interface is not possible.

Open with function `IBalObject::OpenSocket`.

▶ In parameter *riid* enter value `IID_ILinSocket`.

▶ To request information about functionalities of LIN controller, type of LIN controller and the supported functionalities call function `GetCapabilities` (for further information see `LINCAPABILITIES`).

▶ To determine current operating mode and status of Controller call function `GetLineStatus` (for further information see `LINLINESTATUS`).

▶ To create message monitors call function `CreateMonitor` (for further information see *Message Monitors, p. 48*).

## 5.3.2 Message Monitors

A LIN message channel consists of a receiving FIFO. The size of an element in the FIFO conforms to the size of the structure *LINMSG*.

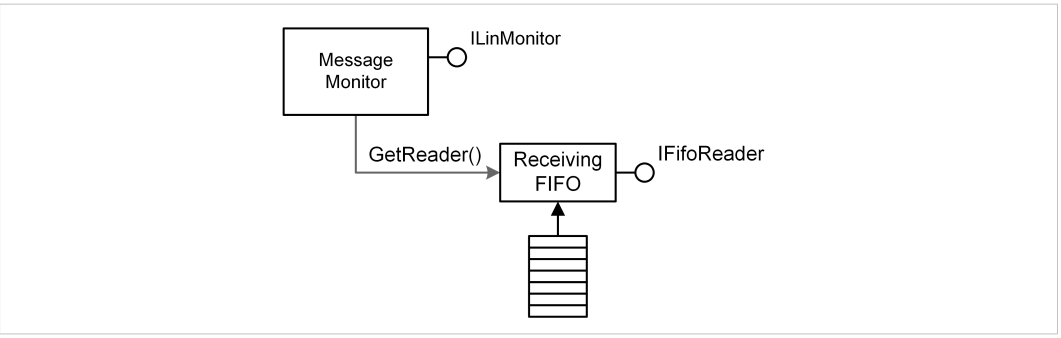


**Fig. 29 Components and interfaces LIN message monitor**

The functionality of a message monitor is the same, irrespective whether the connection is used exclusively or not.

In case of exclusive usage the message monitor is directly connected to the LIN controller.

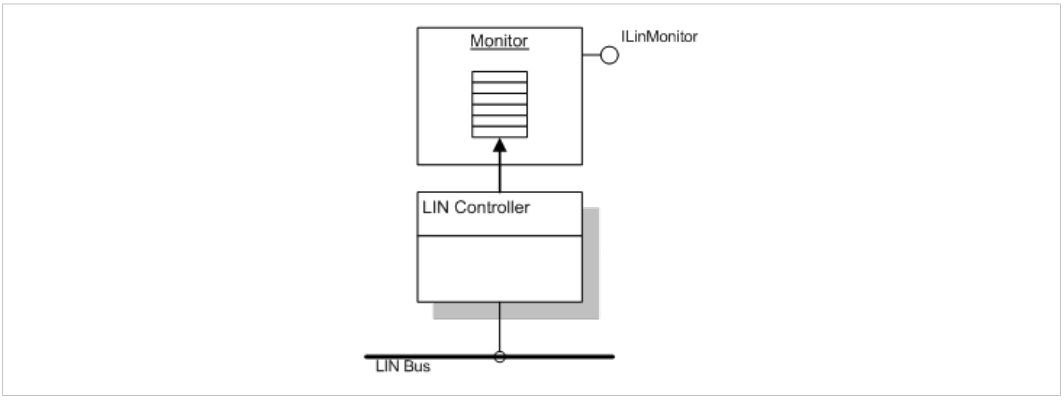


**Fig. 30 Exclusive usage**

In case of non-exclusive usage the individual message monitors are connected to the LIN controller via a distributor. The distributor transfers all on the LIN controller received messages to all active channels. No monitor is prioritized i. e. the algorithm used by the distributor is designed to treat all channels as equal as possible.

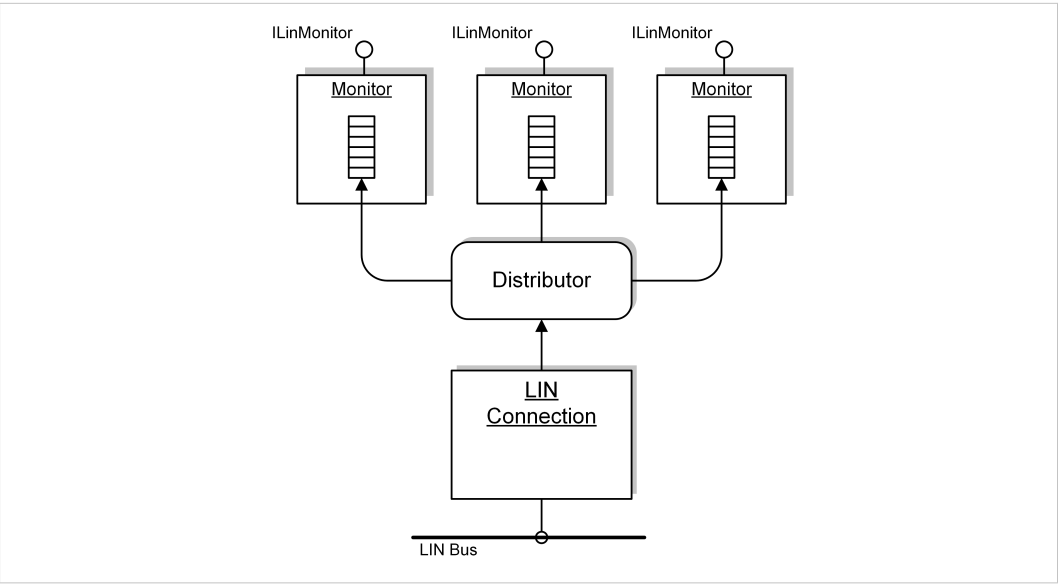


**Fig. 31 Non-exclusive usage (with distributor)**

## Creating a Message Monitor

Create a message monitor with function `ILinSocket::CreateMonitor`.

- To use controller exclusively (only possible when creating the first message monitor) enter in parameter *fExclusive* value `TRUE`. After successful execution no further message monitors can be created.

  or

  To use controller non-exclusively (creation of any number of monitors is possible) enter in parameter *fExclusive* value `FALSE`.

## Initializing the Message Monitor

A newly generated message monitor contains no receiving FIFO.

- To create receiving FIFO call function `Initialize`.
- Specify size of receiving FIFO in parameter *wRxSize*.
- Make sure that value in parameter *wRxSize* is higher than 0.

The size of an element in the FIFO conforms to the size of the structure `LINMSG`.
All functions to access the data elements of the FIFO attend resp. return a pointer to structures of type `LINMSG`.

## Activating the Message Channel

A newly generated monitor is deactivated. Message are exclusively received by the bus if the message monitor is active and if the LIN controller is started. Further information about LIN controllers see chapter *Control Unit, p. 51*.

- To activate the message monitor call function `Activate`.
- Disconnect active channel with function `Deactivate`.

### Reading Messages From the Receiving FIFO:

- To access the receiving FIFO call function `ILinMonitor::GetReader`.
  - ▷ Pointer to interface `IFifoReader` is returned.

Reading messages from the FIFO:

- Call function `IFifoReader::GetDataEntry`.

  Make sure that parameter *pvData* points to buffer of type `LINMSG`.

  or

- Call function `IFifoReader::AcquireRead`.
  - ▷ Returns pointer to next free message in the FIFO and the number of messages that can be read sequentially from this position onward.
  - ▷ Function returns pointer to array of type `LINMSG`.
- After processing remove data with function `IFifoReader::ReleaseRead` from FIFO.

*The address returned by `AcquireRead` points directly to the memory of the FIFO. Make sure that exclusively elements of the valid range are addressed.*

#### Possible Usage of Function `GetDataEntry`:

```
void DoMessages( IFifoReader* pReader )
{
  LINMSG sLinMsg;
  while( pReader->GetDataEntry (&sLinMsg) == VCI_OK )
  {
     // Processing of message
  }
}
```

#### Possible Usage of Functions `AcquireRead` and `ReleaseRead`:

```
void DoMessages( IFifoReader* pReader )
{
  PLINMSG pLinMsg;
  UINT16 wCount;

  while( pReader->AcquireRead((PVOID*) &pLinMsg, &wCount) == VCI_OK )
  {
    for( UINT16 i = 0; i < wCount; i++ )
    {
      // processing of message
      .
      .
      .
      // set pointer ahead to next message
      pLinMsg++;
    }

    // release read message
    pReader->ReleaseRead(wCount);
  }
}
```

## 5.3.3 Control Unit

The control unit provides the following functions via the interface *ILinControl*:

- configuration of operating mode
- configuration of transmitting features
- requesting of current controller state

The control unit can exclusively be opened by one application. Simultaneous multiple opening by several programs is not possible.

### Opening the Interface

Open with function `IBalObject::`*OpenSocket*.

▶ In parameter *riid* enter value `IID_ILinControl`.

▷ If the function returns an error code like *access denied* the component is already used by another program.

▶ With `Release` close control unit and release for access by other applications.

*If other interfaces of the controller are opened when the controller is closed, the current settings remain, i. e. a started LIN controller is not stopped automatically with calling `Release` as long as an additional message monitor is opened.*

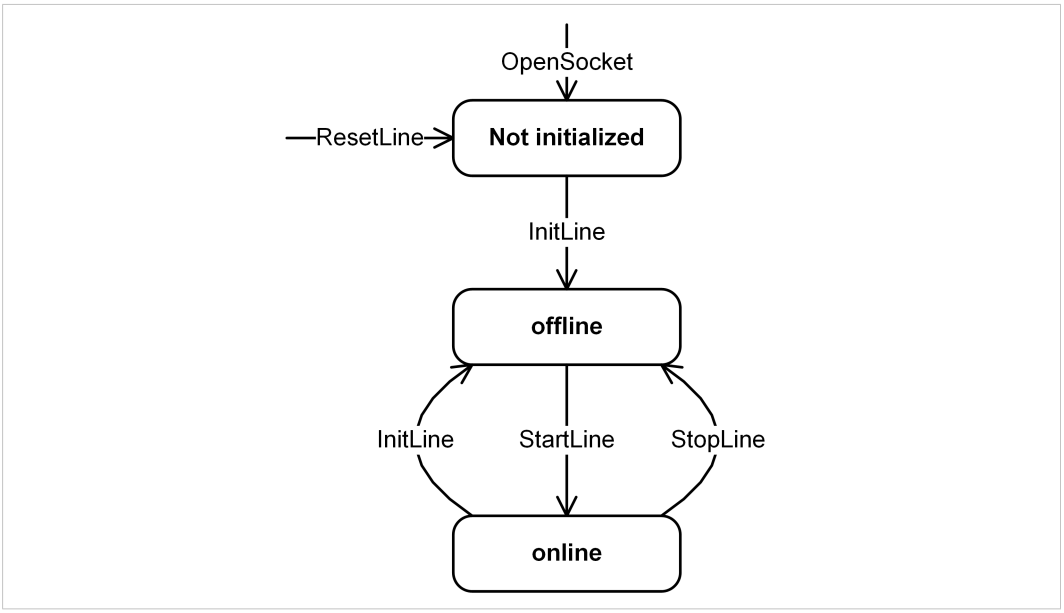


**Fig. 32 LIN controller states**

### Initializing the Controller

After the first opening of the interface *ILinControl* the controller is in a non-initialized state.

▶ To leave non-initzialized state call function *InitLine*.

▷ Controller is in state *offline*.

▶ Specify system mode and transmission rate with function *InitLine*.

▷ Function requires in parameter *pInitParam* pointer to an initialized structure of type *LININITLINE*.

- Specify transmission rate in bits per second in field *wBitrate* of structure *LININITLINE*.

  Valid values are between 1000 and 20000 Bit/s, resp. between the values specified by LIN_BITRATE_MIN and LIN_BITRATE_MAX.

If the controller supports automatic bit identification, in field *wBitrate* can be entered the value LIN_BITRATE_AUTO if the LIN controller is already connected to an active network.

### Starting and Stopping the Controller

- To start LIN controller call function *StartLine*.
  - LIN controller is in state *online*.
  - LIN controller is actively connected to bus.
  - Incoming messages are forwarded to all opened and active message monitors.
- To stop LIN controller call function *StopLine*.
  - LIN controller is in state *offline*.
  - Message transfer to the monitor is interrupted and controller is deactivated.
  - In case of an ongoing data transfer the function waits until the message is transmitted completely over the bus, before the message transmission is stopped.
- Call function *ResetLine* to shift controller in state *not initialized* and to reset controller hardware.

*With calling the function ResetLine a faulty message telegram on the bus is possible if an ongoing transmission is interrupted.*

Neither *ResetLine* nor *StopLine* delete the content of the receiving FIFOs of a message monitor.

## Transmitting CAN messages

Messages can be transmitted directly with the function *WriteMessage* or can be registered in a response table in the controller.

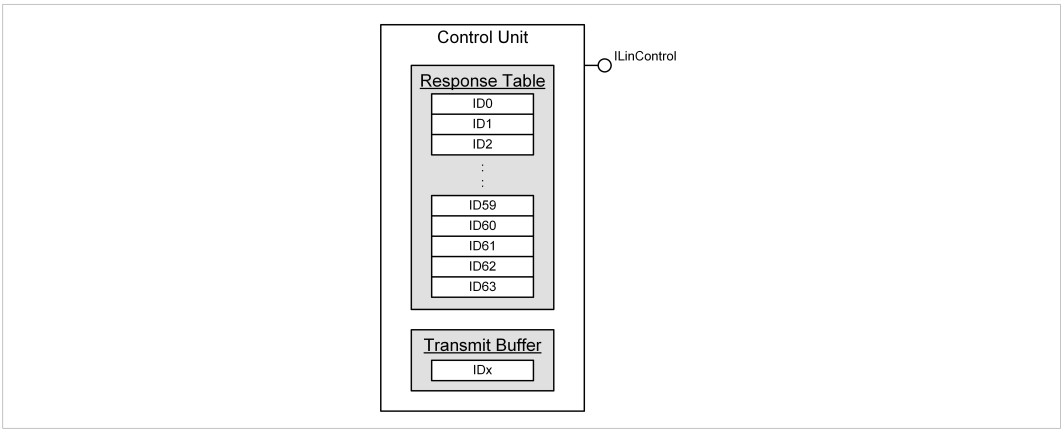


**Fig. 33 Internal structure of a control unit**

The control unit contains an internal response table with the response data for the IDs transmitted by the master. If the controller detects an ID that is assigned to it and transmitted by the master it transmits the response data entered in the table at the corresponding position automatically to the bus.

- To chance or update the content of the response table call function *WriteMessage*.
- Enter in parameter *fSend* the value FALSE and in parameter *pLinMsg* a valid LIN message.
- To clear response table call function *ResetLine*.

Field *abData* of structure *LINMSG* contains the response data. The LIN message must be of type LIN_MSGTYPE_DATA and must contain an ID in the range 0 to 63.

Irrespective of the operating mode (master or slave) the table must be initialized before the controller is started. It can be updated at any time without stopping the controller.

- Transmitting messages directly to the bus with function *WriteMessage*.
- Set parameter *fSend* to value TRUE true.
  - Message is registered in the transmitting buffer of the controller, instead of the response table.
  - Controller transmits message to bus as soon as it is free.

If the controller is configured as master, control messages LIN_MSGTYPE_SLEEP and LIN_MSGTYPE_WAKEUP and data messages of type LIN_MSGTYPE_DATA can be transmitted directly. If the controller is configured as slave exclusively LIN_MSGTYPE_WAKEUP messages can be directly transmitted. With all other message types the function returns an error code.

A message of type LIN_MSGTYPE_SLEEP generates a goto-Sleep frame, a message of type LIN_MSGTYPE_WAKEUP a wake-up frame on the bus. For further information see LIN specifications in chapter Network Management

In the master mode the function *WriteMessage* also serves for transmitting IDs. For this a message of type LIN_MSGTYPE_DATA with valid ID and data length, where the bit *uMsgInfo.Bits.ido* is set to 1, is required (for further information see *LINMSGINFO*).

Irrespective of the value of the parameter *fSend* *WriteMessage* always returns immediately to the calling program without waiting for the transmission to be completed. If the function is called before the last transmission is completed or before the transmission buffer is free again, the function returns with a respective error code.

# 6 Interface Description

## 6.1 Exported Functions

### 6.1.1 VciInitialize

The function initializes the VCI for the calling process.

```
HRESULT VCIAPI VciInitialize ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function must be called at the beginning of a program to initialize the DLL for the calling process. | |

### 6.1.2 VciFormatError

The function converts a VCI error code into a text resp. a character string that is readable for the user.

```
HRESULT VCIAPI VciFormatError (
     HRESULT hrError,
     PTCHAR  pszError,
     UINT32  dwLength);
```

| | | |
|---|---|---|
| **Parameter** | *hrError* | [in] Error code that is to be converted into text. |
| | *pszError* | [out] Pointer to buffer for the text string. Function saves character string including the final 0-character in this memory area. |
| | *dwLength* | [in] Size of buffer in number of strings. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | *VCI_E_INVALIDARG* | Parameter *pszError* points to invalid buffer. |

### 6.1.3 VciGetVersion

The function determines the current version numbers of the VCI and of the operating system that is currently running.

```
HRESULT VCIAPI VciGetVersion ( PVCIVERSIONINFO pVersionsInfo );
```

| | | |
|---|---|---|
| **Parameter** | *pVersionsInfo* | [out] Pointer to data block of type VCIVERSIONINFO. If run successfully the function stores the version information in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function can be called at the beginning of a program to check whether the current VCI of the application is sufficient.<br>Further information about the information returned by this function see description of data structure *VCIVERSIONINFO*. | |

### 6.1.4 VciCreateLuid

The function generates a VCI-specific, unique ID.

```
HRESULT VCIAPI VciCreateLuid ( PVCIID pVciid );
```

| | | |
|---|---|---|
| **Parameter** | *pVciid* | [out] Pointer to variable to type VCIID. If run successfully the function saves the VCI-specific ID in this variable. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Returned ID can be used during the running time of the system to mark application specific objects as unique. ID loses validity with next start of the system. | |

### 6.1.5 VciLuidToChar

The function converts a unique ID (VCIID) into a character string.

```
HRESULT VCIAPI VciLuidToChar (
     REFVCIID rVciid,
     PCHAR    pszLuid,
     LONG     cbSize );
```

| | | |
|---|---|---|
| **Parameter** | *rVciid* | Reference to the VCI-specific unique ID to be converted (VCIID). |
| | *pszLuid* | [out] Pointer to buffer for the character string. If run successfully the function saves the converted VCI-specific ID in this memory area. Buffer must provide space for at least 17 characters including the final 0–character. |
| | *cbSize* | [in] Size in bytes of buffer specified in *pszLuid*. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_INVALIDARG | Parameter *pszLuid* points to invalid buffer. |
| | VCI_E_BUFFER_OVERFLOW | The buffer specified in *pszLuid* is not large enough for the character string. |

### 6.1.6 VciCharToLuid

The function converts a 0-terminated character string into a VCI-specific unique ID (VCIID).

```
HRESULT VCIAPI VciCharToLuid (
     PCHAR  pszLuid,
     PVCIID pVciid );
```

| | | |
|---|---|---|
| **Parameter** | *pszLuid* | [in] Pointer to the 0-terminated string to be converted |
| | *pVciid* | [out] Pointer to variable to type VCIID. If run successfully, the function returns the converted ID in this variable. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_INVALIDARG | Parameter *pszLuid* or *pVciid* points to invalid buffer. |
| | VCI_E_FAIL | In *pszLuid* specified character string can not be converted to valid ID. |

### 6.1.7 VciGuidToChar

The function converts a globally unique identifier (GUID) into a character string.

```
HRESULT VCIAPI VciGuidToChar (
     REFGUID rGuid,
     PCHAR   pszLuid,
     LONG    cbSize );
```

| | | |
|---|---|---|
| **Parameter** | *rGuid* | [in] Reference to the globally unique ID to be converted. |
| | *pszGuid* | [out] Pointer to buffer for the character string. If run successfully the function saves the converted globally unique ID in this memory area. Buffer must provide space for at least 39 characters including the final 0-character. |
| | *cbSize* | [in] Size in bytes of buffer specified in *pszGuid*. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_INVALIDARG | Parameter *pszLuid* points to invalid buffer. |
| | VCI_E_BUFFER_OVERFLOW | The buffer specified in *pszLuid* is not large enough for the character string. |

### 6.1.8 VciCharToGuid

The function converts a 0-terminated character string into a globally unique ID (GUID).

```
HRESULT VCIAPI VciCharToGuid (
     PCHAR pszGuid,
     PGUID pGuid );
```

| | | |
|---|---|---|
| **Parameter** | *pszGuid* | [in] Pointer to the 0-terminated string to be converted. |
| | *pGuid* | [out] Pointer to variable to type GUID. If run successfully, the function returns the converted ID in this variable. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_INVALIDARG | Parameter *pszGuid* or *pGuid* points to invalid buffer. |
| | VCI_E_FAIL | In *pszGuid* specified character string can not be converted into a valid ID. |

### 6.1.9 VciGetDeviceManager

The function determines a pointer to the interface *IVciDeviceManager* of the VCI device manager.

```
HRESULT VCIAPI VciGetDeviceManager (
     IVciDeviceManager** ppDevMan );
```

| | | |
|---|---|---|
| **Parameter** | *ppDevMan* | [out] Address of a pointer variable. If run successfully the function saves the pointer to interface *IVciDeviceManager* of the VCI device manager. In case of an error the variable is set to NULL. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Further information about the device manager and the exported interfaces and functions see *Interfaces of the Device Management, p. 60*. | |

### 6.1.10 VciQueryDeviceByHwid

The function opens a device or controller with a particular hardware ID.

```
HRESULT VCIAPI VciQueryDeviceByHwid (
    REFGUID rHwid,
    IVciDevice** ppDevice );
```

| | | |
|---|---|---|
| **Parameter** | *rHwid* | [in] Reference to hardware ID of the controller to be opened. |
| | *ppDevice* | [out] Address of a pointer variable. If run successfully the function saves the pointer to interface *IVciDevice* of the VCI device manager. In case of an error the variable is set to NULL. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Every device or bus controller has a distinct hardware ID, that stays valid after the system is restarted. | |

### 6.1.11 VciQueryDeviceByClass

The function opens a device or controller with a particular device class.

```
HRESULT VCIAPI VciQueryDeviceByClass (
     REFGUID      rClass
     UINT32      dwInst,
     IVciDevice** ppDevice );
```

| | | |
|---|---|---|
| **Parameter** | *rClass* | [in] Reference to class ID of the controller to be opened. |
| | *dwInst* | [in] Number of controller to be opened. If several controllers of the same class are present, this value determines the number of the controller to be opened in the device list. Value 0 selects the first controller of the specified class. |
| | *ppDevice* | [out] Address of a pointer variable. If run successfully the function saves the pointer to interface *IVciDevice* of the opened device or adapter. In case of an error the variable is set to NULL. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Every bus controller is assigned to a distinct device class. The instance number of this controller is not fixed, but changes dependent on when or how the controller was activated or generated by the system. This is to be considered if USB or other external controllers are used, because these can be plugged in and off during the system is running. | |

### 6.1.12 VciCreateFifo

The function creates a new FIFO and determines a pointer to one of the interfaces *IVciFifo* bzw. *IVciFifo2*, *IFifoReader* or *IFifoWriter*.

```
HRESULT VCIAPI VciCreateFifo (
     PVCIID pResid,
     UINT16 wCapacity,
     UINT16 wElementSize,
     REFIID riid,
     PVOID* ppv );
```

| | | |
|---|---|---|
| **Parameter** | *pResid* | [out] Pointer to variable of type VCIID. If run successfully the VCI-specific individually unique ID is stored by newly generated FIFO. This ID can be used for further calls of the function *VciAccessFifo* to reach additional interfaces of the FIFO. |
| | *wCapacity* | [in] Number of elements in the newly generated FIFO. |
| | *wElementSize* | [in] Size of an element in number of bytes |
| | *riid* | [in] ID of the interface to access the component. FIFOs support the interface IDs `IID_IFifoReader`, `IID_IFifoWriter` and `IID_IVciFifo` resp. `IID_IVciFifo2`. |
| | *ppv* | [out] Address of a pointer variable. If run successfully the pointer is stored in the in *riid* requested interface. If the FIFO can't be generated or if the FIFO does not support the interface specified in *riid* variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | FIFOs occupy more than *(wCapacity*wElementSize)* bytes. The calculated size is always rounded up to the whole memory sites, with the result that the FIFO eventually contains more element than requested (further information about the memory consumption see *Communication Components, p. 12*). | |

### 6.1.13 VciAccessFifo

The function opens an existing FIFO and requests one of the interfaces *IVciFifo* resp. *IVciFifo2*, *IFifoReader* or *IFifoWriter*.

```
HRESULT VCIAPI VciAccessFifo (
     REFVCIID rResid,
     REFIID   riid,
     PVOID*   ppv );
```

| | | |
|---|---|---|
| **Parameter** | *rResid* | [in] Reference to the VCI-specific ID of the FIFO to be opened. |
| | *riid* | [in] ID of the interface to access the component. FIFOs support the interface IDs `IID_IFifoReader`, `IID_IFifoWriter` and `IID_IVciFifo` resp. `IID_IVciFifo2`. |
| | *ppv* | [out] Address of a pointer variable. If run successfully the pointer is stored in the in *riid* requested interface. If the interface specified in *riid* is not supported, the FIFO can't be opened or access is not possible, the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The interface *IFifoReader* resp. *IFifoWriter* can exclusively be opened once at a certain time. If the requested interface is already used the call fails. The anew opening of the interface is not possible until it is released. | |

## 6.2 Interface IUnknown

All components provided by the VCI implement the interface `IUnknown` specified in the Component Object Model of Microsoft (MS-COM). The interface provides besides the function `QueryInterface` to request further interfaces of the component, additionally the functions `AddRef` resp. `Release` to control the lifespan of the component.

### 6.2.1 QueryInterface

Via this function a particular interface of a component can be called.

```
ULONG QueryInterface ( REFIID riid, PVOID *ppv );
```

| | | |
|---|---|---|
| **Parameter** | *riid* | [in] Reference to the ID of the interface to access the component. |
| | *ppv* | [out] Address of a pointer variable. If run successfully the pointer is stored in the in *riid* requested interface. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | If run successfully the function increments the reference counter of the component automatically by 1. When the application do not need the interfaces resp. the components any more, the pointer returned in *ppv* must be releases with *`Release`*. | |

### 6.2.2 AddRef

The function increments the reference count of the component by 1.

```
ULONG AddRef ( void );
```

| | |
|---|---|
| **Parameter** | - |
| **Possible Responses** | Function returns the current value of the reference count. |
| **Remark** | The function always must be called, if the application stores a copy of the interface pointer. With this it is ensured that the component exists as long as the last reference to it is released. An interface resp. the connected component is released by the call of the function *`Release`*. |

### 6.2.3 Release

The function decrements the reference count of the component by 1. If the reference count falls to 0 the component is released.

```
ULONG Release ( void );
```

| | |
|---|---|
| **Parameter** | - |
| **Possible Responses** | Function returns the current value of the reference count. |
| **Remark** | After calling the function the pointer to the interface used by the application is not valid any longer and must not be used anymore. This also applies if the function returns a value lager than 0, i. e. the component itself is not released by this call. |

# 6.3 Interfaces of the Device Management

## 6.3.1 IVciDeviceManager

The interface is used to access the VCI device manager. A pointer to this interface is provided by the API function `VciGetDeviceManager`. The ID of the interface is `IID_IVciDeviceManager`.

### EnumDevices

The function creates an object to list all devices registered on VCI.

```
HRESULT EnumDevices( IVciEnumDevice** ppEnumDevice )
```

| | | |
|---|---|---|
| **Parameter** | *ppEnumDevice* | [out] Address of a pointer variable. If run successfully the function saves the pointer to interface `IVciEnumDevice` of the device list. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |

### OpenDevice

The function opens a device.

```
HRESULT OpenDevice (
     REFVCIID     rVciidDev,
     IVciDevice** ppDevice )
```

| | | |
|---|---|---|
| **Parameter** | *rVciidDev* | [in] Reference to the unique ID of the controller to be opened. |
| | *ppDevice* | [out] Address of a pointer variable. If run successfully the function saves the pointer to interface *`IVciDevice`* of the VCI device manager. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | ID of device to be opened can be determined with function `IVciEnumDevice::`*`Next`* (see *Listing Available Devices, p. 10*). | |

## 6.3.2 IVciEnumDevice

The interfaces serves for listing all devices that are currently registered in the VCI (functionality see *Listing Available Devices, p. 10*) The ID of the interface is `IID_IVciEnumDevice`.

### Next

The function determines the description of one or more devices in the device list and increments an internal index, with the result that a subsequent call of the function returns the description to the respectively next devices.

```
HRESULT Next (
     UINT32         dwNumElem,
     PVCIDEVICEINFO paDevInfo,
     PUINT32        pdwFetched );
```

| | | |
|---|---|---|
| **Parameter** | *dwNumElem* | [in] Number of list elements that are to be determined with this call. |
| | *paDevInfo* | Pointer of array of minimum *dwNumElem* elements of type *VCIDEVICEINFO*. If run successfully the function stores the individual information about the devices in this memory area |
| | *pdwFetched* | [out] Pointer to variable in which the function saves the actually determined elements if ran successfully. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| | `VCI_E_NO_MORE_ITEMS` | No further elements available or end of list is reached. |
| **Remark** | In Parameter *pdwFetched* function can be applied into the value `NULL` if in parameter *dwNumElem* value 1 is specified. | |

### Skip

The function skips a certain number of entries in the device list.

```
HRESULT Skip ( UINT32 dwNumElem );
```

| | | |
|---|---|---|
| **Parameter** | *dwNumElem* | [in] Number of elements to be skipped. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Usage of function only makes sense in static lists, because in static lists the order of the devices is fixed during the runtime. | |

### Reset

The function resets the internal index to initial state, with the result that a subsequent call of `Next` returns again the first element of the list.

```
HRESULT Reset ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |

### AssignEvent

The function assigns an *Event* to the device list, that is always set in signalized state when a device is added to or deleted from the list.

```
HRESULT AssignEvent ( HANDLE hEvent );
```

| | | |
|---|---|---|
| **Parameter** | *hEvent* | [in] Handle of event object. Specified handle must originate of Windows function *CreateEvent*. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

## 6.3.3 IVciDevice

The interface provides functions to request general information and to open application specific components of an adapter. The ID of the interface is IID_IVciDevice.

### GetDeviceInfo

The function determines general information about a device.

```
HRESULT GetDeviceInfo ( PVCIDEVICEINFO pInfo );
```

| | | |
|---|---|---|
| **Parameter** | *pInfo* | [out] Pointer to data block of type VCIDEVICEINFO. If run successfully the function stores the information about the device in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | For further information about the data returned by this function see *VCIDEVICEINFO*. | |

### GetDeviceCaps

The function determines information about the technical capabilities of a device.

```
HRESULT GetDeviceCaps ( PVCIDEVICECAPS pCaps );
```

| | | |
|---|---|---|
| **Parameter** | *pCaps* | [out] pointer to data block of type VCIDEVICECAPS. If run successfully the function stores the information about the technical capabilities in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | For further information about the data returned by this function see *VCIDEVICECAPS*. | |

### OpenComponent

The function opens an application specific component of the adapter.

```
HRESULT OpenComponent (
     REFCLSID rcid,
     REFIID   riid,
     PVOID*   ppv )
```

| | | |
|---|---|---|
| **Parameter** | *rcid* | [in] Reference to class ID of the component to be opened. `CLSID_VCIBAL`: Opens access to Bus Access Layer (BAL). |
| | *riid* | [in] Reference to the ID of the interface to access the component. |
| | *ppv* | [out] Address of a pointer variable. If run successfully the pointer is stored in the in *riid* requested interface of the *rcid* specified component. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For parameters *rcid* and *riid* the following combinations are possible<br>*rcid*: `CLSID_VCIBAL`<br>*riid*: `IID_IUnknown`, `IID_IBalObject`<br>For further information about the function see *Communication Components, p. 12*. Further information about the BAL and its components see *Accessing the Bus Controller, p. 20*. | |

# 6.4 Interfaces of the Communication Components

## 6.4.1 Interfaces for FIFOs

### IVciFifo

Common interface for all FIFO components. Detailed description of FIFO and functionality see *First In/First Out Memory (FIFO), p. 13*. The ID of the interface is `IID_IVciFifo`.

### GetCapacity

The function determines the capacity of the FIFO.

```
HRESULT GetCapacity ( PUINT16 pwCapacity );
```

| | | |
|---|---|---|
| **Parameter** | *pwCapacity* | [out] Pointer to variable to which the capacity of the FIFO is returned if the function succeeded. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Function returns number of data elements that can be stored in FIFO, not the number of bytes. Size of an individual data element can be determined with function `GetEntrySize`. | |

### GetEntrySize

The function determines the size of an individual data element in the FIFO in bytes.

```
HRESULT GetEntrySize ( PUINT16 pwSize );
```

| | | |
|---|---|---|
| **Parameter** | *pwSize* | [out] Pointer to variable to which the size of an individual data element is returned if the function succeeded. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |

### GetFreeCount

The function determines the current number of free data elements in the FIFO.

```
HRESULT GetFreeCount ( PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *pwCount* | [out] Pointer to variable to which the number of free data elements is returned if the function succeeded. Value informs how many data elements additionally fit into the FIFO. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

### GetFillCount

The function determines the current number of occupied data elements in the FIFO.

```
HRESULT GetFillCount ( PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *pwCount* | [out] Pointer to variable to which the number of occupied data elements is returned if the function succeeded. Value informs how many data elements are not yet read. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

### GetFillLevel

The function determines the filling level of the FIFO in percentage.

```
HRESULT GetFillLevel ( PUINT16 pwLevel );
```

| | | |
|---|---|---|
| **Parameter** | *pwLevel* | [out] Pointer to variable to which the current filling level in percentage is returned if the function succeeded. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

## IVciFifo2

The interface IVciFifo2 expands the interface IVciFifo with additional features. The ID of the interface is IID_IVciFifo2.

### Reset

The function deletes the current FIFO content.

```
HRESULT Reset ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error |
| | VCI_E_ACCESSDENIED | An interface is opened. |
| **Remark** | Function is exclusively ran successfully if the FIFO is neither accessed reading nor writing during the function is called. When the function is called neither interface *IFifoReader* nor *IFifoWriter* must be opened. | |

## IFifoReader

The interface is used for the reading access to FIFOs (description see *Functionality Receiving FIFO, p. 16*). The ID of the interface is `IID_IFifoReader`.

### Lock

The function waits until the calling thread has exclusive access to the interface and then locks the access to the interface for all other threads of the application.

```
HRESULT Lock ( void );
```

| | |
|---|---|
| **Parameter** | - |
| **Possible Responses** | `VCI_OK` |
| **Remark** | Applications that access the interface simultaneously from several threads must synchronize the access. For that at the beginning of a reading sequence `Lock` is always called and at the end `Unlock`. The functions *`GetCapacity`* and *`GetEntrySize`* are excluded because the values returned by this functions are unchangeable. Multiple overlapping calls of `Lock` and `Unlock` are possible. Make sure that after every call of `Lock` a call of *`Unlock`* follows. |

### Unlock

The function releases the access to the interface that is locked with *`Lock`*.

```
HRESULT Unlock ( void );
```

| | |
|---|---|
| **Parameter** | - |
| **Possible Responses** | `VCI_OK` |
| **Remark** | For further information see *`Lock`*. |

### AssignEvent

The function assigns an *Event* to the FIFO which is always set in signalized state if the filling level of the FIFO exceeds a certain threshold.

```
HRESULT AssignEvent ( HANDLE hEvent );
```

| | | |
|---|---|---|
| **Parameter** | *hEvent* | [in] Handle of object. Specified handle must originate of Windows function `CreateEvent`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Exclusively one *Event* can be assigned to the interface. If the function is multiply called a previously assigned *Event* is overwritten. The currently assigned *Event* can be removed by calling the function with value `NULL` in parameter *hEvent*. The *Event* is triggered if an element is assigned to the FIFO and the filling level reaches or exceeds the set threshold. For further information about the function see *Functionality Receiving FIFO, p. 16*. | |

### SetThreshold

The function determines the threshold for the filling level at which the currently assigned *Event* is signalized.

```
HRESULT SetThreshold ( UINT16 wThreshold );
```

| | | |
|---|---|---|
| **Parameter** | *wThreshold* | [in] Threshold at which the currently with `AssignEvent` assigned *Event* is signalized. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | If the value specified in parameter *wThreshold* exceeds the valid area the function automatically limits the threshold to the capacity of the FIFO. The currently assigned *Event* is triggered if a data element is stored in the FIFO and reaches or exceeds the threshold specified in parameter *wThreshold*. For further information see *Functionality Receiving FIFO, p. 16*. | |

### GetThreshold

The function determines the specified threshold at which a currently assigned *Event* is signalized.

```
HRESULT GetThreshold ( PUINT16 pwThreshold );
```

| | | |
|---|---|---|
| **Parameter** | *pwThreshold* | [out] Pointer to variable to which the currently assigned threshold is returned if the function succeeded. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For further information see `SetThreshold`. | |

### GetCapacity

The function determines the capacity of the FIFO.

```
HRESULT GetCapacity ( PUINT16 pwCapacity );
```

| | | |
|---|---|---|
| **Parameter** | *pwCapacity* | [out] Pointer to variable to which the capacity of the FIFO is returned if the function succeeded. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Function returns number of data elements that can be stored in FIFO, not the number of bytes. Size of an individual data element can be determined with function `GetEntrySize`. | |

### GetEntrySize

The function determines the size of an individual data element in the FIFO in bytes.

```
HRESULT GetEntrySize ( PUINT16 pwSize );
```

| | | |
|---|---|---|
| **Parameter** | *pwSize* | [out] Pointer to variable to which the size of an individual data element is returned if the function succeeded. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |

### GetFillCount

The function determines the current number of not yet read resp. valid data elements in the FIFO.

```
HRESULT GetFillCount ( PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *pwCount* | [out] Pointer to variable to which the current number of occupied data elements is returned if the function succeeded. Value informs how many data elements are not yet read. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

### GetFreeCount

The function determines the current number of free data elements in the FIFO.

```
HRESULT GetFreeCount ( PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *pwCount* | [out] Pointer to variable to which the number of free data elements is returned if the function succeeded. Value informs how many data elements additionally fit into the FIFO. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Value returned in *pwCount* informs how many elements additionally fit into the FIFO until it is crowded. | |

### GetDataEntry

The function reads the next valid data element in the FIFO.

```
HRESULT GetDataEntry ( PVOID pvData );
```

| | | |
|---|---|---|
| **Parameter** | *pvData* | [out] Pointer to the buffer memory of the data element to be read If the value NULL is entered the function removes the next element in the FIFO. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_RXQUEUE_EMPTY | No data element in FIFO while calling the function |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function copies the content of the next valid data element to the memory area to which parameter *pvData* is pointing. Because of that the memory area must be at least as big as a data element in the FIFO. Size of an individual data element can be determined with function *GetEntrySize*. | |

### AcquireRead

The function determines a pointer to the next unread data element in the FIFO and the number of elements that can be read sequentially from this position onward.

```
HRESULT AcquireRead (PVOID* ppvData, PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *ppvData* | [out] Address of a pointer variable. If run successfully address of first valid element that can be read is stored. |
| | *pwCount* | [out] If run successfully pointer to variable in which number of valid elements is stored that can be read from the in *ppvData* returned address onward. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_INVALIDARG | Invalid parameter |
| | VCI_E_RXQUEUE_EMPTY | FIFO contains no more valid elements. |
| **Remark** | In *ppvData* returned address can be used as pointer to an array with *pwCount* elements. Every element in the array has the size that is specified in bytes when creating the FIFO. Since the pointer returned in *ppvData* points directly to the memory of the FIFO it must be made sure that no element outside the valid area is read. In parameter *pwCount* the value NULL can be specified if the program is only interested in the next free element. In this case when calling *ReleaseRead* it is maximally allowed to specify parameter *wCount* with 1. | |

### ReleaseRead

The function releases a certain number of data element from the current reading position in the FIFO onward.

```
HRESULT ReleaseRead ( UINT16 wCount );
```

| | | |
|---|---|---|
| **Parameter** | *wCount* | [in] Number of data elements in the FIFO to be released. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function updates the reading position in the FIFO corresponding to the number of elements specified in *wCount*. Value specified in *wCount*must not exceed the number returned by *AcquireRead* but can be 0 if no element is to be released. | |

## IFifoWriter

The interface is used for the transmitting access to FIFOs (further information see *Functionality Transmitting FIFO, p. 18*). The ID of the interface is IID_IFifoWriter.

### Lock

The function waits until the calling thread has exclusive access to the interface and then locks the access to the interface for all other threads of the application.

```
HRESULT Lock ( void );
```

| | |
|---|---|
| **Parameter** | - |
| **Possible Responses** | VCI_OK |
| **Remark** | Applications that access the interface simultaneously from several threads must synchronize the access. For that at the beginning of a writing sequence Lock is always called and at the end Unlock. The functions *GetCapacity* and *GetEntrySize* are excluded because the values returned by this functions are unchangeable. Multiple overlapping calls of Lock and Unlock are possible. Make sure that after every call of Lock a call of Unlock follows. |

### Unlock

The function releases the access to the interface that was locked with `Lock`.

```
HRESULT Unlock ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | |
| **Remark** | For further information see function *`Lock`*. | |

### AssignEvent

The function assigns an *Event* to the FIFO which is always set in signalized state if the number of free elements exceed a certain value or if the filling level is below a certain value.

```
HRESULT AssignEvent ( HANDLE hEvent );
```

| | | |
|---|---|---|
| **Parameter** | *hEvent* | [in] Handle of event. Specified handle must originate of Windows API function `CreateEvent`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Exclusively one *Event* can be assigned to the interface. If the function is multiply called a previously assigned *Event* is overwritten. The currently assigned *Event* can be removed by calling the function with value `NULL` in parameter *hEvent*. The *Event* is triggered if an element is removed from the FIFO and the specified threshold is reached or exceeded or if the filling level is below a specified value. For further information see *Functionality Transmitting FIFO, p. 18*. | |

### SetThreshold

The function determines the threshold for the filling level at which the currently assigned *Event* is signalized.

```
HRESULT SetThreshold ( UINT16 wThreshold );
```

| | | |
|---|---|---|
| **Parameter** | *wThreshold* | [in] Threshold at which the currently with *`AssignEvent`* assigned *Event* is signalized. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | If the value specified in parameter *wThreshold* exceeds the valid area the function automatically limits the threshold to the capacity of the FIFO. The currently assigned *Event* is triggered if an element is removed from the FIFO and the number of free entries reaches or exceeds the threshold specified in parameter *wThreshold* or if the filling level is below a specified value. For further information see *Functionality Receiving FIFO, p. 16*. | |

### GetThreshold

The function determines the specified threshold at which a currently assigned *Event* is signalized.

```
HRESULT GetThreshold ( PUINT16 pwThreshold );
```

| | | |
|---|---|---|
| **Parameter** | *pwThreshold* | [out] Pointer to variable to which the currently assigned threshold is returned if the function succeeded. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For further information see *`SetThreshold`*. | |

### GetCapacity

The function determines the capacity of the FIFO.

```
HRESULT GetCapacity ( PUINT16 pwCapacity );
```

| | | |
|---|---|---|
| **Parameter** | *pwCapacity* | [out] Pointer to variable to which the capacity of the FIFO is returned if the function succeeded. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Function returns number of data elements that can be stored in FIFO, not the number of bytes. Size of an individual data element can be determined with function *GetEntrySize*. | |

### GetEntrySize

The function determines the size of an individual data element in the FIFO in bytes.

```
HRESULT GetEntrySize ( PUINT16 pwSize );
```

| | | |
|---|---|---|
| **Parameter** | *pwSize* | [out] Pointer to variable to which the size of an individual data element is returned if the function succeeded. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

### GetFillCount

The function determines the current number of not yet read resp. valid data elements in the FIFO.

```
HRESULT GetFillCount ( PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *pwCount* | [out] Pointer to variable to which the current number of occupied data elements is returned if the function succeeded. Value informs how many data elements are not yet read. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

### GetFreeCount

The function determines the current number of free data elements in the FIFO.

```
HRESULT GetFreeCount ( PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *pwCount* | [out] Pointer to variable to which the number of free data elements is returned if the function succeeded. Value informs how many data elements additionally fit into the FIFO. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Value returned in *pwCount* informs how many elements additionally fit into the FIFO until it is crowded. | |

### PutDataEntry

The function writes a data element to the FIFO.

```
HRESULT PutDataEntry ( PVOID pvData );
```

| | | |
|---|---|---|
| **Parameter** | *pvData* | [in] Pointer to the data element to be written. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_TXQUEUE_FULL | No space available in FIFO. |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function copies the content of the memory area to which parameter *pvData* is pointing to next valid data element. Because of that the memory area specified in *pvData* must be at least as big as a data element in the FIFO<br>Size of a data element can be determined with function *GetEntrySize*. | |

### AcquireWrite

The function determines a pointer to the next unread data element in the FIFO and the number of elements that can be addressed linearly from this position onward.

```
HRESULT AcquireWrite ( PVOID* ppvData, PUINT16 pwCount );
```

| | | |
|---|---|---|
| **Parameter** | *ppvData* | [out] Address of a pointer variable. If run successfully address of first valid element that can be addressed is stored. |
| | *pwCount* | [out] If run successfully pointer to variable in which number of valid elements is stored that can be addressed from the in *ppvData* returned address onward. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | VCI_E_INVALIDARG | Invalid parameter |
| | VCI_E_TXQUEUE_FULL | FIFO contains no more valid elements. |
| **Remark** | In *ppvData* returned address can be used as pointer to an array with *pwCount* elements. Every element in the array has the size that is specified in bytes when creating the FIFO.<br>Since the pointer returned in *ppvData* points directly to the memory of the FIFO it must be made sure that no element outside the valid area is addressed.<br>In parameter *pwCount* the value NULL can be specified if the program is only interested in the next free element. In this case when calling *ReleaseRead* it is maximally allowed to specify parameter *wCount* with 1. | |

### ReleaseWrite

The function releases a certain number of data element from the current reading position in the FIFO onward.

```
HRESULT ReleaseWrite ( UINT16 wCount );
```

| | | |
|---|---|---|
| **Parameter** | *wCount* | [in] Number of data element to be declared valid in the FIFO. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function updates the writing position in the FIFO corresponding to the number of elements specified in *wCount*.<br>Value specified in *wCount*must not exceed the number returned by *AcquireWrite* but can be 0 if no element is to be declared valid. | |

## 6.5 BAL Specific Interfaces

The following chapters describe the interfaces and functions for the access of the controllers of a bus adapter. Introducing information see *Accessing the Bus Controller, p. 20*.

### 6.5.1 IBalObject

The interface provides functions to determine the features of the BAL and to open bus controllers. The ID of the interface is `IID_IBalObject`.

#### GetFeatures

The function determines the functionalities of the Bus Access Layer (BAL) of the bus adapter.

```
HRESULT GetFeatures ( PBALFEATURES pBalFeatures );
```

| | | |
|---|---|---|
| **Parameter** | *pBalFeatures* | [out] Pointer to data block of type `BALFEATURES`. If run successfully the function stores the features of the BAL in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *`BALFEATURES`*. | |

#### OpenSocket

The function opens a bus controller and request an interface from it.

```
HRESULT OpenSocket ( UINT32 dwBusNo, REFIID riid, PVOID* ppv );
```

| | | |
|---|---|---|
| **Parameter** | *dwBusNo* | [in] Number of the bus controller to be opened. Value 0 selects the first bus controller, value 1 the second, etc. |
| | *riid* | [in] Reference to the ID of the interface to access the bus component. |
| | *ppv* | [out] Address of a pointer variable. If run successfully the pointer is stored in the in *riid* requested interface. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | If run successfully the function increments the reference counter of the opened bus controller automatically by 1. When the application do not need the interfaces resp. the bus controller any more, the pointer returned in *ppv* must be releases with *`Release`*. Information about number and type of available bus controllers and possible values for *dwBusNo* see description of data structure *`BALFEATURES`*. | |

# 6.6 CAN Specific Interfaces

## 6.6.1 ICanSocket

The interface contains functions to request for features and to create message channels for a CAN controller. The ID of the interface is `IID_ICanSocket`.

### GetSocketInfo

The function determines general information about the bus controller.

```
HRESULT GetSocketInfo ( PBALSOCKETINFO pSocketInfo );
```

| | | |
|---|---|---|
| **Parameter** | *pSocketInfo* | [out] Pointer to memory area of type `BALSOCKETINFO`. If run successfully the function stores the information about the bus controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *`BALSOCKETINFO`*. | |

### GetCapabilities

The function determines the features of the CAN controller.

```
HRESULT GetCapabilities ( PCANCAPABILITIES pCanCaps );
```

| | | |
|---|---|---|
| **Parameter** | *pCanCaps* | [out] Pointer to memory area of type `CANCAPABILITIES`. If run successfully the function stores the features of the CAN controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *`CANCAPABILITIES`*. | |

### GetLineStatus

The function determines the current settings and the current state of the CAN Controller.

```
HRESULT GetLineStatus ( PCANLINESTATUS pLineStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pLineStatus* | [out] Pointer to memory area of type `CANLINESTATUS`. If run successfully the function saves the current settings and the current state of the controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *`CANLINESTATUS`*. | |

### CreateChannel

The function opens resp. creates a message channel for the CAN controller.

```
HRESULT CreateChannel (
  BOOL         fExclusive,
  PCANCHANNEL* ppChannel );
```

| | | |
|---|---|---|
| **Parameter** | *fExclusive* | [in] Determines if the controller is exclusively used for the channel to be opened. If the value `TRUE` is specified no further message channels can be opened after the function ran successfully until the newly generated channel is released again. If value `FALSE` is specified multiple message channels for the CAN controller can be opened. |
| | *ppChannel* | [out] Address of a variable to which a pointer to the interface *`ICanChannel`* is assigned by the newly generated message channel if ran successfully. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The program that calls the function first with the value `TRUE` in parameter *fExclusive* exclusively controls the message transfer on the CAN bus. If the message channel is not required any more the pointer returned in *ppChannel* must be released by calling the function *`Release`*. For general information about message channels see *Message Channels, p. 23*. | |

## 6.6.2 ICanSocket2

The interface contains functions to request for the features and to create message channels for a expanded CAN controller. The ID of the interface is `IID_ICanSocket2`.

### GetSocketInfo

The function determines general information about the bus controller.

```
HRESULT GetSocketInfo ( PBALSOCKETINFO pSocketInfo );
```

| | | |
|---|---|---|
| **Parameter** | *pSocketInfo* | [out] Pointer to memory area of type `BALSOCKETINFO`. If run successfully the function stores the information about the bus controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *`BALSOCKETINFO`*. | |

### GetCapabilities

The function determines the features of the CAN controller.

```
HRESULT GetCapabilities ( PCANCAPABILITIES pCanCaps );
```

| | | |
|---|---|---|
| **Parameter** | *pCanCaps* | [out] Pointer to memory area of type `CANCAPABILITIES2`. If run successfully the function stores the features of the CAN controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *`CANCAPABILITIES2`*. | |

### GetLineStatus

The function determines the current settings and the current state of the CAN Controller.

```
HRESULT GetLineStatus ( PCANLINESTATUS2 pLineStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pLineStatus* | [out] Pointer to memory area of type `CANLINESTATUS2`. If run successfully the function saves the current settings and the current state of the controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *CANLINESTATUS2*. | |

### CreateChannel

The function opens resp. creates a message channel for the CAN controller.

```
HRESULT CreateChannel (
  BOOL           fExclusive,
  PCANCHANNEL2* ppChannel );
```

| | | |
|---|---|---|
| **Parameter** | *fExclusive* | [in] Determines if the controller is exclusively used for the channel to be opened. If the value `TRUE` is specified no further message channels can be opened after the function ran successfully until the newly generated channel is released again. If value `FALSE` is specified multiple message channels for the CAN controller can be opened. |
| | *ppChannel* | [out] Address of a variable to which a pointer to the interface *ICanChannel2* is assigned by the newly generated message channel if ran successfully. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The program that calls the function first with the value `TRUE` in parameter *fExclusive* exclusively controls the message transfer on the CAN bus. If the message channel is not required any more the pointer returned in *ppChannel* must be released by calling the function *Release*. For general information about message channels see *Message Channels, p. 23*. | |

## 6.6.3 ICanControl

Basic information to the functionality of the component see *Control Unit, p. 31*. The ID of the interface is `IID_ICanControl`.

### DetectBaud

The function determines the current bit rate of the CAN bus connected to the adapter.

```
HRESULT DetectBaud (
  UINT16       wTimeoutMs,
  PCANBTRTABLE pBtrTable );
```

| | | |
|---|---|---|
| **Parameter** | *wTimeoutMs* | [in] Maximum waiting time in milliseconds between two receiving messages on the bus. |
| | *pBtrTable* | [in/out] Pointer to a initialized structure of type *CANBTRTABLE* with predefined set of bus timing values to be tested. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| | `VCI_E_NOT_IMPLEMENTED` | Function not supported by device |
| | `VCI_E_TIMEOUT` | No communication on the bus during the time specified in *wTimeoutMs*. |
| **Remark** | If run successfully, field *bIndex* of structure *CANBTRTABLE* contains table index of the found bus timing values. The values at the respective positions in the tables *abBtr0* and *abBtr1* can then be used to initialize the CAN controller with *InitLine*.<br>Before calling it is possible to specify in *bIndex* additional parameters about the operating mode that is used by the bit rate to detect. Valid is either `CAN_OPMODE_LOWSPEED` or 0, if no low speed coupling is desired.<br>The function can be called in undefined state or after a reset of the controller. Further information about the automatic detection of the bit rate see *Determine the Bit Rate Used in the Network, p. 39*. | |

### InitLine

The function specifies the operating mode and the bit rate of the CAN controller.

```
HRESULT InitLine ( PCANINITLINE pInitParam );
```

| | | |
|---|---|---|
| **Parameter** | *pInitParam* | [in] Pointer to initialized structure of type `CANINITLINE`. Field *bOpMode* determines the operating mode, Fields *bBtReg0* and *bBtReg1* the bit rate of the CAN controller. Further information about the fields see description of data structure *CANINITLINE*.. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function resets the controller hardware like the function *ResetLine*. Controller is newly initialized with specified values.<br>Values for the bus timing register BTR0 and BTR1 resp. the therefore defined constants for the CiA resp. CANopen specified bit rates and further information about setting the bit rate see *Specifying the Bit Rate, p. 34*. | |

## ResetLine

The function resets the CAN controller and the message filters of the control unit to the initial state.

```
HRESULT ResetLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | For further information see *Stopping (resp. Resetting) the Controller, p. 33*. | |

## StartLine

The function starts the CAN controller.

```
HRESULT StartLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | For further information see *Control Unit, p. 31*. | |

## StopLine

The function stops the CAN controller.

```
HRESULT StopLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Unlike *ResetLine* the specified message filters are not modified when the controller is stopped. For further information see *Control Unit, p. 31*. | |

## GetLineStatus

The function determines the current settings and the current state of the CAN Controller.

```
HRESULT GetLineStatus ( PCANLINESTATUS pLineStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pLineStatus* | [out] Pointer to memory area of type CANLINESTATUS. If run successfully the function saves the current settings and the current state of the controller in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function can always be called, even before the first call of one of the functions *InitLine* or *DetectBaud*. Further information about the data returned by this function see description of data structure *CANLINESTATUS*. | |

### SetAccFilter

The function specifies an acceptance filter of the CAN controller.

```
HRESULT SetAccFilter (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *dwCode* | [in] Bit sample of the CAN identifiers to be accepted including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not used for the comparison. But if it has the value 1 it is relevant for the comparison. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

### AddFilterIds

The function assigns one or more CAN message IDs (CAN-ID) in the 11 or 29 bit filter list of the CAN controller.

```
HRESULT AddFilterIds (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *dwCode* | [in] Bit sample of the CAN identifiers to be registered including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not considered. But if it has the value 1 it is relevant. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

### RemFilterIds

The function removes one or more CAN message IDs (CAN-ID) from the 11 or 29 bit filter list of the CAN controller.

```
HRESULT RemFilterIds (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *dwCode* | [in] Bit samples of the CAN identifiers to be removed including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not considered. But if it has the value 1 it is relevant. |

| | | |
|---|---|---|
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

## 6.6.4 ICanControl2

Basic information to the functionality of the component see *Control Unit, p. 31*. The ID of the interface is IID_ICanControl2.

### DetectBaud

The function determines the current bit rate of the CAN buc connected to the adapter.

```
HRESULT DetectBaud (
  UINT8         bOpMode
  UINT8         bExMode
  UINT16        wTimeoutMs,
  PCANBTPTABLE pBtpTable );
```

| | | |
|---|---|---|
| **Parameter** | *bOpMode* | [in] Operating mode of the controller used for detection.<br>CAN_OPMODE_LOWSPEED: CAN controller uses low speed bus coupling. |
| | *bExMode* | [in] Extended operating mode of the controller used for detection. If supported by the controller, a logical combination of one or more of the following constants can be specified:<br>CAN_EXMODE_FASTDATA: Allows higher bit rates for the data field<br>CAN_EXMODE_NONISO: Usage of non-ISO-conform message frames. This option is exclusively available with older CAN FD controller with the feature CAN_FEATURE_NONISOFRM.<br>If the value CAN_EXMODE_DISABLED is specified there is no detection of the fast bit rates. The value also must be specified with all other controllers that do not support extended CAN FD operating mode.<br>See description of field *dwFeatures* of structure *CANCAPABILITIES2*. |
| | *wTimeoutMs* | [in] Maximum waiting time in milliseconds between two receiving messages on the bus. |
| | *pBtpTable* | [in/out] Pointer to a initialized structure of type *CANBTPTABLE* with preset bus timing values to be tested. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| | VCI_E_NOT_IMPLEMENTED | Function not supported by device |
| | *wTimeoutMs* | No communication on the bus during the specified time. |
| **Remark** | If run successfully, field *bIndex* of structure *CANBTPTABLE* contains table index of the found bus timing values. The values at the respective positions in the table can then be used to initialize the CAN controller with *InitLine*.<br>The function can be called in undefined state or after a reset of the controller. Further information about the automatic detection of the bit rate see *Determine the Bit Rate Used in the Network, p. 39*. | |

### InitLine

The function specifies the operating mode and bit rate of the CAN controller and the default and reset values of the operating mode of the message filters.

```
HRESULT InitLine ( PCANINITLINE2 pInitParam );
```

| | | |
|---|---|---|
| **Parameter** | *pInitParam* | [in] Pointer to structure of type `CANINITLINE2` with the parameters required for configuration of operating mode, bit rate and message filters. Further information about the fields see description of data structure *CANINITLINE2*. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function resets the controller hardware like the function `ResetLine`. Controller is newly initialized with specified values.<br>Further information about setting the bit rate see *Specifying the Bit Rate, p. 34*. | |

### ResetLine

The function resets the CAN controller and the message filters of the control unit to initial state.

```
HRESULT ResetLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For further information see *Stopping (resp. Resetting) the Controller , p. 33*. | |

### StartLine

The function starts the CAN controller.

```
HRESULT StartLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For further information see *Starting the Controller, p. 33*. | |

### StopLine

The function stops the CAN controller.

```
HRESULT StopLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Unlike `ResetLine` the specified message filters are not modified when the controller is stopped. For further information see *Control Unit, p. 31*. | |

## GetLineStatus

The function determines the current settings and the current state of the CAN Controller.

```
HRESULT GetLineStatus ( PCANLINESTATUS2 pLineStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pLineStatus* | [out] Pointer to memory area of type *CANLINESTATUS2*. If run successfully the function saves the current settings and the current state of the controller in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function can always be called, even before the first call of one of the functions InitLine or DetectBaud. Further information about the data returned by this function see description of data structure *CANLINESTATUS2*. | |

## GetFilterMode

The function determines the current operating mode of the message filter of the control unit.

```
HRESULT GetFilterMode (
  UINT8  bSelect,
  PUINT8 pbMode );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selecting the filter. With CAN_FILTER_STD the 11 bit, with CAN_FILTER_EXT the 29 bit filter is selected. |
| | *pbMode* | [out] Pointer to variable of type UINT8. If run successfully the value of the currently specified operating mode is assigned. Further information about the returned value see description of function SetFilterMode. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Detailed description of functionality of message filter see *Message Filter, p. 40*. | |

## SetFilterMode

The function specifies the operating mode of the message filter of the control unit.

```
HRESULT SetFilterMode (
  UINT8  bSelect,
  UINT8  bNewMode,
  PUINT8 pbPrevMode );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selecting the filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *bNewMode* | [in] Parameter determines new operating mode for selected filter. One of the following constants can be specified:<br>`CAN_FILTER_LOCK`: Filter blocks all messages of type `CAN_MSGTYPE_DATA`, independent of the ID. The other message types like e. g. `CAN_MSGTYPE_INFO` are not concerned and can always pass.<br>`CAN_FILTER_PASS`: Filter is completely opened and all data messages can pass.<br>`CAN_FILTER_INCL`: All data messages of type `CAN_MSGTYPE_DATA` with an ID either released in the acceptance filter or registered in the filter list can pass the filter (e. i. all registered IDs). Other message types are not concerned and can always pass.<br>`CAN_FILTER_EXCL`: All data messages of type `CAN_MSGTYPE_DATA` with an ID either released in the acceptance filter or registered in the filter list are blocked by the filter (e. i. all registered IDs). Other message types are not concerned and can always pass. |
| | *pbPrevMode* | [out] Pointer to variable of type `UINT8`. If run successfully the value of the last specified operating mode is assigned. The parameter is optional and can be set to `NULL` if the value is not required. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Detailed description of FIFO and functionality see *Message Filter, p. 40*. | |

## SetAccFilter

The function specifies an acceptance filter of the CAN controller.

```
HRESULT SetAccFilter (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *dwCode* | [in] Bit sample of the CAN identifiers to be accepted including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not used for the comparison. But if it has the value 1 it is relevant for the comparison. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

## AddFilterIds

The function assigns one or more CAN message IDs (CAN-ID) in the 11 or 29 bit filter list of the control unit.

```
HRESULT AddFilterIds (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filters selected. |
| | *dwCode* | [in] Bit sample of the identifiers to be registered including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not considered. But if it has the value 1 it is relevant. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

## RemFilterIds

The function removes one or more CAN message IDs (CAN-ID) from the 11 or 29 bit filter list of the control unit.

```
HRESULT RemFilterIds (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *dwCode* | [in] Bit sample of the identifiers to be removed including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not considered. But if it has the value 1 it is relevant. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

## 6.6.5 ICanChannel

The interface provides functions to create a message channel. Basic information to the functionality of the component see *Message Channels, p. 23*. The ID of the interface is `IID_ICanChannel`.

### Initialize

The function initializes the receiving and the transmitting FIFO of the message channel.

```
HRESULT Initialize ( UINT16 wRxFifoSize, UINT16 wTxFifoSize );
```

| | | |
|---|---|---|
| **Parameter** | *wRxFifoSize* | [in] Size of receiving FIFO in number of CAN messages. |
| | *wTxFifoSize* | [in] Size of transmitting FIFO in number of CAN messages. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `VCI_E_INVALIDARG` | Value in parameter *wRxFifoSize* must be higher than 0. |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The specified values determine exclusively the lower limit for the size of the respective FIFO. The actual size is eventually bigger as specified, because the memory for the FIFOs is reserved page by page and the pages are always used completely. For further information see *First In/First Out Memory (FIFO), p. 13*.<br>In parameter *wRxFifoSize* a value higher than 0 must be set. Otherwise the function returns an error code. If no transmitting FIFO is needed, for example if the controller is run in *listen-only* mode, value 0 can be set in *wTxFifoSize*. | |

### Activate

The function activates the message channel and connects the receiving and the transmitting FIFO to the CAN controller.

```
HRESULT Activate ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | By default the message channel is deactivated and disconnected from the bus after it is created resp. initialized. To connect the channel to the bus, the bus must be activated. For further information see *Message Channels, p. 23*. | |

### Deactivate

The function deactivates the message channel and disconnects the receiving and the transmitting FIFO from the CAN controller.

```
HRESULT Deactivate ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | After deactivating the channel no more messages can be transmitted or received. | |

### GetReader

The function determines a pointer to the interface `IFifoReader` of the receiving FIFO of the message channel.

```
HRESULT GetReader ( IFifoReader** ppReader );
```

| | | |
|---|---|---|
| **Parameter** | *ppReader* | [out] Address of a variable to which a pointer to the interface `IFifoReader` is assigned by the receiving FIFO if ran successfully. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function succeeds if the channel is initialized with the function *`Initialize`*. If the pointer returned by the function is not required any more the pointer must be released with `Release`. | |

### GetWriter

The function determines a pointer to the interface *`IFifoWriter`* of the transmitting FIFO of the message channel.

```
HRESULT GetWriter ( IFifoWriter** ppWriter );
```

| | | |
|---|---|---|
| **Parameter** | *ppWriter* | [out] Address of a variable to which a pointer to the interface *`IFifoWriter`* is assigned by the transmitting FIFO if ran successfully. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function succeeds if the channel is initialized with the function `Initialize`. If the pointer returned by the function is not required any more the pointer must be released with `Release`. | |

### GetStatus

The function determines the current state of the message channel and the CAN controller that is connected to the message channel.

```
HRESULT GetStatus ( PCANCHANSTATUS pStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pStatus* | [out] Pointer to memory area of type `CANCHANSTATUS`. If run successfully the function saves the current settings and the current state of the message channel in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function can always be called, even before the first call of one of the function `Initialize`. Further information about the data returned by this function see description of structure *`CANCHANSTATUS`*. | |

## 6.6.6 ICanChannel2

The interface provides functions to create a message channel. Basic information to the functionality of the component see *Message Channels, p. 23*. The ID of the interface is `IID_ICanChannel2`.

### Initialize

The function initializes the receiving and the transmitting FIFO of the message channel.

```
HRESULT Initialize (
  UINT32 dwRxFifoSize,
  UINT32 dwTxFifoSize
  UINT32 dwFilterSize
  UINT8  bFilterMode );
```

| | | |
|---|---|---|
| **Parameter** | *dwRxFifoSize* | [in] Size of receiving FIFO in number of CAN messages. |
| | *dwTxFifoSize* | [in] Size of transmitting FIFO in number of CAN messages. |
| | *dwFilterSize* | [in] Number of 29 bit message IDs supported by the filter list. 11 bit filter list supports all 2048 possible message IDs. If value 0 is specified, no filter list is created. In this case exact filtering is not possible. Filtering with acceptance filter is not affected by that. |
| | *bFilterMode* | [in] Default value for operating mode of message filter. One of the following constants can be specified:<br>`CAN_FILTER_LOCK`: Filter blocks all messages of type `CAN_MSGTYPE_DATA`, independent of the ID. The other message types like e. g. `CAN_MSGTYPE_INFO` are not concerned and can always pass.<br>`CAN_FILTER_PASS`: Filter is completely opened and all messages can pass.<br>`CAN_FILTER_INCL`: All data messages of type `CAN_MSGTYPE_DATA` with an ID either released in the acceptance filter or registered in the filter list can pass the filter (e. i. all registered IDs). Other message types are not concerned and can always pass.<br>`CAN_FILTER_EXCL`: All data messages with an ID either released in the acceptance filter or registered in the filter list are blocked by the filter (e. i. all registered IDs). Other message types are not concerned and can always pass.<br>The filter operating mode can be combined with the constant `CAN_FILTER_SRRA`. The message channel then receives all self reception messages that are transmitted to the controller by other channels. If `CAN_FILTER_SRRA` is not specified the channel exclusively receives its own self reception messages. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| | `VCI_E_INVALIDARG` | Value in parameter *dwRxFifoSize* must be higher than 0. |
| **Remark** | The values specified in *dwRxSize* resp. *dwTxSize* determine exclusively the lower limit for the size of the respective FIFO. The actual size is eventually bigger as specified, because the memory for the FIFOs is reserved page by page and the pages are always used completely. For further information see *First In/First Out Memory (FIFO), p. 13*.<br>In parameter *dwRxFifoSize* a value higher than 0 must be set. Otherwise the function returns an error code. If no transmitting FIFO is needed, for example if the controller is run in *listen-only* mode, value 0 can be set in *dwTxFifoSize*. | |

### Activate

The function activates the message channel and connects the receiving and the transmitting FIFO to the CAN controller.

```
HRESULT Activate ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | By default the message channel is deactivated and disconnected from the bus after it is created resp. initialized. To connect the channel to the bus, the bus must be activated. For further information see *Message Channels, p. 23*. | |

### Deactivate

The function deactivates the message channel and disconnects the receiving and the transmitting FIFO from the CAN controller.

```
HRESULT Deactivate ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | After deactivating the channel no more messages can be transmitted or received. | |

### GetReader

The function determines a pointer to the interface *IFifoReader* of the receiving FIFO of the message channel.

```
HRESULT GetReader ( IFifoReader** ppReader );
```

| | | |
|---|---|---|
| **Parameter** | *ppReader* | [out] Address of a variable to which a pointer to the interface *IFifoReader* is assigned by the receiving FIFO if ran successfully. In case of an error the variable is set to NULL. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function succeeds if the channel is initialized with the function Initialize. If the pointer returned by the function is not required any more the pointer must be released with Release. | |

### GetWriter

The function determines a pointer to the interface *IFifoWriter* of the transmitting FIFO of the message channel.

```
HRESULT GetWriter ( IFifoWriter** ppWriter );
```

| | | |
|---|---|---|
| **Parameter** | *ppWriter* | [out] Address of a variable to which a pointer to the interface *IFifoWriter* is assigned by the transmitting FIFO if ran successfully. In case of an error the variable is set to NULL. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function succeeds if the channel is initialized with the function Initialize. If the pointer returned by the function is not required any more the pointer must be released with Release. | |

### GetStatus

The function determines the current state of the message channel and the CAN controller that is connected to the message channel.

```
HRESULT GetStatus ( PCANCHANSTATUS2 pStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pStatus* | [out] Pointer to memory area of type `CANCHANSTATUS2`. If run successfully the function saves the current settings and the current state of the message channel in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function can always be called, even before the first call of one of the function `Initialize`. Further information about the data returned by this function see description of structure *`CANCHANSTATUS2`*. | |

### GetControl

The function opens the control unit of the controller that is connected to the message channel.

```
HRESULT GetControl ( PCANCONTROL2* ppCanCtrl );
```

| | | |
|---|---|---|
| **Parameter** | *ppCanCtrl* | [out] Address of a variable to which a pointer to the interface `ICanControl2` is assigned by the opened control unit if run successfully. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The control unit of a controller can exclusively be opened once at a certain time. If the control unit is not required any more the pointer returned in *ppCanCtrl* must be released by calling the function `Release`. | |

### GetFilterMode

The function determines the current operating mode of the message channel.

```
HRESULT GetFilterMode (
  UINT8  bSelect,
  PUINT8 pbMode );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selecting the filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *pbMode* | [out] Pointer to variable of type `UINT8`. If run successfully the value of the currently specified operating mode is assigned. Further information about the returned value see description of function `SetFilterMode`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Detailed description of functionality of message filter see *Message Filter, p. 40*. | |

## SetFilterMode

The function determines the operating mode of the message channel.

```
HRESULT SetFilterMode (
  UINT8  bSelect,
  UINT8  bNewMode,
  PUINT8 pbPrevMode );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selecting the filter. With value CAN_FILTER_STD the 11 bit and with value CAN_FILTER_EXT the 29 bit filter is selected. |
| | *bNewMode* | [in] Parameter determines new operating mode for selected filter. One of the following constants can be specified:<br>CAN_FILTER_LOCK: Filter blocks all messages of type CAN_MSGTYPE_DATA, independent of the ID. The other message types like e. g. CAN_MSGTYPE_INFO are not concerned and can always pass.<br>CAN_FILTER_PASS: Filter is completely opened and all messages can pass.<br>CAN_FILTER_INCL: All data messages of type CAN_MSGTYPE_DATA with an ID either released in the acceptance filter or registered in the filter list can pass the filter (e. i. all registered IDs). Other message types are not concerned and can always pass.<br>CAN_FILTER_EXCL: All data messages of type CAN_MSGTYPE_DATA with an ID either released in the acceptance filter or registered in the filter list are blocked by the filter (e. i. all registered IDs). Other message types are not concerned and can always pass. |
| | *pbPrevMode* | [out] Pointer to variable of type UINT8. If run successfully the value of the last specified operating mode is assigned. The parameter is optional and can be set to NULL if the value is not required. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Detailed description of functionality of message filter see *Message Filter, p. 40*. | |

## SetAccFilter

The function specifies the 11 or the 29 bit acceptance filter of the CAN message channel.

```
HRESULT SetAccFilter (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With CAN_FILTER_STD the 11 bit, with CAN_FILTER_EXT the 29 bit filter is selected. |
| | *dwCode* | [in] Bit sample of the CAN identifiers to be accepted including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not used for the comparison. But if it has the value 1 it is relevant for the comparison. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

## AddFilterIds

The function assigns one or more CAN message IDs (CAN-ID) in the 11 or 29 bit filter list of the message list.

```
HRESULT AddFilterIds (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *dwCode* | [in] Bit sample of the CAN identifiers to be registered including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not considered. If it has the value 1 it is relevant. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

## RemFilterIds

The function removes one or more CAN message IDs (CAN-ID) from the 11 or 29 bit filter list of the message channel.

```
HRESULT RemFilterIds (
  UINT8  bSelect,
  UINT32 dwCode,
  UINT32 dwMask );
```

| | | |
|---|---|---|
| **Parameter** | *bSelect* | [in] Selects the acceptance filter. With `CAN_FILTER_STD` the 11 bit, with `CAN_FILTER_EXT` the 29 bit filter is selected. |
| | *dwCode* | [in] Bit sample of the CAN identifiers to be removed including RTR bit. |
| | *dwMask* | [in] Bit samples of the relevant bits in *dwCode*. If a bit in *dwMask* has the value 0 the correlating bit in *dwCode* is not considered. If it has the value 1 it is relevant. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For detailed description of the functionality of the filter and the values for the parameter *dwCode* and *dwMask* see *Message Filter, p. 40*. | |

## 6.6.7 ICanScheduler

The interface provides functions to create, start and stop the cyclic transmitting list of a CAN controller. Basic information to the functionality of the component see *Cyclic Transmitting List, p. 44*. The ID of the interface is `IID_ICanScheduler`.

### Resume

The function starts the transmitting task of the cyclic transmitting list and therefore the transfer of all currently registered transmitting objects.

```
HRESULT Resume ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function can be used to start all registered transmitting objects simultaneously. Before calling the function all transmitting objects must be set in a started state with the function `StartMessage`. A subsequent call of this function then guarantees a simultaneous start of all registered transmitting objects. | |

### Suspend

The function stops the transmitting task of the cyclic transmitting list and therefore the transfer of all currently registered transmitting objects.

```
HRESULT Suspend ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function stops all transmitting objects simultaneously. | |

### Reset

The function stops the transmitting task and removes all transmitting objects from the cyclic transmitting list.

```
HRESULT Reset ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |

### GetStatus

The function determines the current state of the transmitting task and of all registered transmitting objects of a cyclic transmitting list.

```
HRESULT GetStatus ( PCANSCHEDULERSTATUS pStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pStatus* | [out] Pointer to structure of type `CANSCHEDULERSTATUS`. If run successfully the function saves the current settings and the current state of all cyclic transmitting objects in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function returns in array `CANSCHEDULERSTATUS.abMsgStat` the state of all transmitting objects. The list index returned by function `AddMessage` is used to request the state of a transmitting object, this means the array element `abMsgStat[Index]` contains the state of the transmitting object of the specified index. Further information about the data returned by this function see description of data structure *CANSCHEDULERSTATUS*. | |

### AddMessage

The function adds a new transmitting object to the cyclic transmitting list.

```
HRESULT AddMessage (
  PCANCYCLICTXMSG pMessage,
  PUINT32         pdwIndex );
```

| | | |
|---|---|---|
| **Parameter** | *pMessage* | [in] Pointer to initialized structure of type *CANCYCLICTXMSG* with the cyclic transmitting object. |
| | *pdwIndex* | [out] Pointer to variable of type UINT32. In case of successful execution the function returns the list index of the newly added transmitting object of this variable. In case of an error the variable is set to `0xFFFFFFFF`. This index is required for all further callings of functions. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The cyclic transmitting of the newly added transmitting object starts after the successful calling of function `StartMessage`. The transmitting list must be simultaneously active (see *Resume*). | |

### RemMessage

The function stops the processing of a transmitting object and removes it from the cyclic transmitting list.

```
HRESULT RemMessage ( UINT32 dwIndex );
```

| | | |
|---|---|---|
| **Parameter** | *dwIndex* | [in] List index of the transmitting object to be removed. The list index must originate from an earlier call of the function *AddMessage*. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | After calling the function the list index specified in *dwIndex* is invalid and must not be used any more. | |

### StartMessage

The function starts the processing of a transmitting object of the cyclic transmitting list.

```
HRESULT StartMessage ( UINT32 dwIndex, UINT16 dwCount );
```

| | | |
|---|---|---|
| **Parameter** | *dwIndex* | [in] List index of the transmitting object to be started. The list index must originate from an earlier call of the function `AddMessage`. |
| | *dwCount* | [in] Number of cyclic transmitting repetitions. With value 0 the transmitting is repeated endlessly. The specified value must be in between 0 and 65535. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The cyclic transmitting exclusively starts if the transmitting task is active when calling the function. If the transmitting task is inactive the transmitting is delayed until the next calling of function `Resume`. | |

### StopMessage

The function stops the processing of a transmitting object of the cyclic transmitting list.

```
HRESULT StopMessage ( UINT32 dwIndex );
```

| | | |
|---|---|---|
| **Parameter** | *dwIndex* | [in] List index of the transmitting object to be stopped. The list index must originate from an earlier call of the function `AddMessage`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |

## 6.6.8 ICanScheduler2

The interface provides functions to create, start and stop the cyclic transmitting list of an extended CAN controller. Basic information to the functionality of the component see *Cyclic Transmitting List, p. 44*. The ID of the interface is `IID_ICanScheduler`.

### Resume

The function starts the transmitting task of the cyclic transmitting list and therefore the transfer of all currently registered transmitting objects.

```
HRESULT Resume ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function can be used to start all registered transmitting objects simultaneously. Before calling the function all transmitting objects must be set in a started state with the function `StartMessage`. A subsequent call of this function then guarantees a simultaneous start of all registered transmitting objects. | |

## Suspend

The function stops the transmitting task of the cyclic transmitting list and therefore the transfer of all currently registered transmitting objects.

```
HRESULT Suspend ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function stops all transmitting object simultaneously. | |

## Reset

The function stops the transmitting task and removes all transmitting objects from the cyclic transmitting list.

```
HRESULT Reset ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

## GetStatus

The function determines the current state of the transmitting task and of all registered transmitting objects of a cyclic transmitting list.

```
HRESULT GetStatus ( PCANSCHEDULERSTATUS2 pStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pStatus* | [out] Pointer to structure of type CANSCHEDULERSTATUS2. If run successfully the function saves the current settings and the current state of all cyclic transmitting objects in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function returns in array CANSCHEDULERSTATUS2.abMsgStat the state of all transmitting objects. The list index returned by function AddMessage is used to request the state of a transmitting object, this means the array element abMsgStat[Index] contains the state of the transmitting object of the specified index. Further information about the data returned by this function see description of data structure *CANSCHEDULERSTATUS2*. | |

## AddMessage

The function adds a new transmitting object to the cyclic transmitting list.

```
HRESULT AddMessage (
  PCANCYCLICTXMSG2 pMessage,
  PUINT32          pdwIndex );
```

| | | |
|---|---|---|
| **Parameter** | *pMessage* | [in] Pointer to initialized structure of type *CANCYCLICTXMSG2* with the cyclic transmitting object. |
| | *pdwIndex* | [out] Pointer to variable of type UINT32. In case of successful execution the function returns the list index of the newly added transmitting object of this variable. In case of an error the variable is set to 0xFFFFFFFF. This index is required for all further callings of functions. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The cyclic transmitting of the newly added transmitting object starts after the successful calling of function StartMessage. The transmitting list must be simultaneously active (see *Resume*). | |

## RemMessage

The function stops the processing of a transmitting object and removes it from the cyclic transmitting list.

```
HRESULT RemMessage ( UINT32 dwIndex );
```

| | | |
|---|---|---|
| **Parameter** | *dwIndex* | [in] List index of the transmitting object to be removed. The list index must originate from an earlier call of the function *AddMessage*. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | After calling the function the list index specified in *dwIndex* is invalid and must not be used any more. | |

## StartMessage

The function starts the processing of a transmitting object of the cyclic transmitting list.

```
HRESULT StartMessage ( UINT32 dwIndex, UINT16 dwCount );
```

| | | |
|---|---|---|
| **Parameter** | *dwIndex* | [in] List index of the transmitting object to be started. The list index must originate from an earlier call of the function *AddMessage*. |
| | *dwCount* | [in] Number of cyclic transmitting repetitions. With value 0 the transmitting is repeated endlessly. The specified value must be in between 0 and 65535. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The cyclic transmitting exclusively starts if the transmitting task is active when calling the function. If the transmitting task is inactive the transmitting is delayed until the next calling of function *Resume*. | |

### StopMessage

The function stops the processing of a transmitting object of the cyclic transmitting list.

```
HRESULT StopMessage ( UINT32 dwIndex );
```

| | | |
|---|---|---|
| **Parameter** | *dwIndex* | [in] List index of the transmitting object to be stopped. The list index must originate from an earlier call of the function *AddMessage*. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

## 6.7 LIN Specific Interface

### 6.7.1 ILinSocket

The interface contains functions to request for the features and to create message monitors for a LIN controller. The ID of the interface is IID_ILinSocket.

### GetSocketInfo

The function determines general information about the bus controller.

```
HRESULT GetSocketInfo ( PBALSOCKETINFO pSocketInfo );
```

| | | |
|---|---|---|
| **Parameter** | *pSocketInfo* | [out] Pointer to structure of type BALSOCKETINFO. If run successfully the function stores the information about the bus controller in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Further information about the data returned by this function see description of data structure *BALSOCKETINFO*. | |

### GetCapabilities

The function determines the features of the LIN controller.

```
HRESULT GetCapabilities ( PLINCAPABILITIES pLinCaps );
```

| | | |
|---|---|---|
| **Parameter** | *pLinCaps* | [out] Pointer to structure of type LINCAPABILITIES. If run successfully the function stores the features of the LIN controller in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Further information about the data returned by this function see description of data structure *LINCAPABILITIES*. | |

### GetLineStatus

The function determines the current settings and the current state of the LIN Controller.

```
HRESULT GetLineStatus ( PLINLINESTATUS pLineStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pLineStatus* | [out] Pointer to memory area of type `LINLINESTATUS`. If run successfully the function saves the current settings and the current state of the controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about the data returned by this function see description of data structure *LINLINESTATUS*. | |

### CreateMonitor

The function creates the message monitor for the LIN controller.

```
HRESULT CreateMonitor (
  BOOL         fExclusive,
  PLINMONITOR* ppMonitor );
```

| | | |
|---|---|---|
| **Parameter** | *fExclusive* | [in] Determines if the controller is used exclusively for the newly created monitor. If value `TRUE` is specified no further message monitors can be created after the function ran successfully until the newly generated monitors released again. If value `FALSE` is specified as many message monitors as desired can be created for the LIN controller. |
| | *ppMonitor* | [out] Address of a variable to which a pointer to the interface `ILinMonitor` is assigned by the newly created monitor if run successfully. In case of an error the variable is set to `NULL`. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | For further information see *Message Monitors, p. 48*. | |

## 6.7.2 ILinControl

The interface provides functions to configure and control a LIN controller. Basic information about functionality of component see *Control Unit, p. 51*. The ID of the interface is `IID_ILinControl`.

### InitLine

The function specifies the operating mode and the bit rate of the LIN controller.

```
HRESULT InitLine ( PLININITLINE pInitParam );
```

| | | |
|---|---|---|
| **Parameter** | *pInitParam* | [in] Pointer to initialized structure of type `LININITLINE`. The field *bOpMode*determines the operating mode and the field *wBitrate* the bit rate of the LIN controller. Further information about the fields see description of data structure *LININITLINE*. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The function internally resets the controller hardware corresponding to the function *ResetLine* and initializes the LIN controller with the values specified in *pInitParam*. | |

## ResetLine

The function resets the LIN controller to initial state.

```
HRESULT ResetLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | Further information see *Control Unit, p. 51*. | |

## StartLine

The function starts the LIN controller.

```
HRESULT StartLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | A call of the function is exclusively successful if the LIN controller is configured with function *InitLine*. After the function is successful run the LIN controller is connected to the bus (online). Incoming messages are forwarded to all active message monitors resp. transmitting messages are transmitted to the bus. For further information see *Message Monitors, p. 48*. | |

## StopLine

The function stops the LIN controller.

```
HRESULT StopLine ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | For further information see *Message Monitors, p. 48*. | |

## WriteMessage

The function transmits the specified message either directly to the controller that is connected to the LIN bus or assigns the message to the response table of the controller.

```
HRESULT WriteMessage (BOOL fSend, PLINMSG pLinMsg );
```

| | | |
|---|---|---|
| **Parameter** | *fSend* | [in] Determines if a message is directly transmitted to the bus or if it is assigned to the response table of the controller. With TRUE the message is transmitted directly, with FALSE the message is assigned to the response table. |
| | *pLinMsg* | [in] Pointer to initialized structure of type *LINMSG* with the LIN message to be transmitted. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |

### GetLineStatus

The function determines the current settings and the current state of the LIN Controller.

```
HRESULT GetLineStatus ( PLINLINESTATUS pLineStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pLineStatus* | [out] Pointer to memory area of type `LINLINESTATUS`. If run successfully the function saves the current settings and the current state of the controller in this memory area. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | Further information about returned data see description of data structure *LINLINESTATUS*. | |

## 6.7.3 ILinMonitor

The interface provides functions to create a message monitor. For further information about the functionality of the component see *Message Monitors, p. 48*. The ID of the interface is `IID_ILinMonitor`.

### Initialize

The function initializes the receiving FIFO of the monitor.

```
HRESULT Initialize ( UINT16 wRxSize );
```

| | | |
|---|---|---|
| **Parameter** | *wRxSize* | [in] Size of receiving FIFO in number of LIN messages. |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `VCI_E_INVALIDARG` | Value in parameter *wRxSize* must be higher than 0. |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | The value specified in parameter *wRxSize* defines the lower limit for the size of the FIFO. The actual size is eventually bigger as specified, because the memory for the FIFOs is reserved page by page and the pages are always used completely. The size of a element in the FIFO always conforms to the size of the structure *LINMSG*. | |

### Activate

The function activates the message monitor and connects the receiving FIFO to the LIN controller.

```
HRESULT Activate ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | `VCI_OK` | Function succeeded |
| | `!=VCI_OK` | Error, further information about error code provides the function `VciFormatError` |
| **Remark** | By default the message monitor is deactivated and disconnected from the bus after it is created resp. initialized. To connect the monitor to the bus, the bus must be activated. For further information see *Message Monitors, p. 48*. | |

## Deactivate

The function deactivates the message monitor and connects the receiving FIFO to the LIN controller.

```
HRESULT Deactivate ( void );
```

| | | |
|---|---|---|
| **Parameter** | - | |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | If the monitor is deactivated it doesn't receive any message from the LIN controller. | |

## GetReader

The function determines a pointer to the interface *IFifoReader* of the receiving FIFO of the message monitor.

```
HRESULT GetReader ( IFifoReader** ppReader );
```

| | | |
|---|---|---|
| **Parameter** | *ppReader* | [out] Address of a variable to which a pointer to the interface *IFifoReader* is assigned by the receiving FIFO if run successfully. In case of an error the variable is set to NULL. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function succeeds if the monitor is initialized with the function *Initialize*. If the pointer returned by the function is not required any more the pointer must be released with *Release*. | |

## GetStatus

The function determines the current state of the message monitor and the LIN controller that is connected to the message monitor.

```
HRESULT GetStatus ( PLINMONITORSTATUS pStatus );
```

| | | |
|---|---|---|
| **Parameter** | *pStatus* | [out] Pointer to memory area of type LINMONITORSTATUS. If run successfully the function saves the current settings and the current state of the message monitor in this memory area. |
| **Possible Responses** | VCI_OK | Function succeeded |
| | !=VCI_OK | Error, further information about error code provides the function VciFormatError |
| **Remark** | The function can always be called, even before the first call of one of the function *Initialize*. Further information about returned data see description of data structure *LINMONITORSTATUS*. | |

# 7 Data Structures

## 7.1 VCI Specific Data Types

The declaration of all VCI specific data types and constants is stored in the files *vcitype.h* and *restype.h*.

### 7.1.1 VCIID

The data type describes a VCI-wide unique ID resp. a VCI-specific unique ID (LUID) and has the following setup:

```
typedef union _VCIID
{
   LUID AsLuid;
   INT64 AsInt64
} VCIID, *PVCIID;
```

| | |
|---|---|
| *AsLuid* | ID in form of a LUID. Data type LUID is defined in Windows. |
| *AsInt64* | ID as a signed 64 bit integer. |

### 7.1.2 VCIVERSIONINFO

The structure `VciGetVersion` describes version information and has the following structure:

```
typedef struct _VCIVERSIONINFO
{
  UINT32 VciMajorVersion;
  UINT32 VciMinorVersion;
  UINT32 VciReleaseNumber;
  UINT32 VciBuildNumber;
  UINT32 OsMajorVersion;
  UINT32 OsMinorVersion;
  UINT32 OsBuildNumber;
  UINT32 OsPlatformId;
} VCIVERSIONINFO, *PVCIVERSIONINFO;
```

| | |
|---|---|
| *VciMajorVersion* | [out] Major version number |
| *VciMinorVersion* | [out] Minor version number |
| *VciRevNumber* | [out] Revision number |
| *VciBuildNumber* | [out] Build number |
| *OsMajorVersion* | [out] Major version number of operating system |
| *OsMinorVersion* | [out] Minor version number of operating system |
| *OsBuildNumber* | [out] Build version number of operating system |
| *OsPlatformId* | [out] Platform ID |

### 7.1.3 VCIDEVICEINFO

The structure describes the general information about a device and has the following setup:

```
typedef struct _VCIDEVICEINFO
{
  VCIID VciObjectId;
  GUID DeviceClass;

  UINT8 DriverMajorVersion;
  UINT8 DriverMinorVersion;
  UINT16 DriverBuildVersion

  UINT8 HardwareBranchVersion
  UINT8 HardwareMajorVersion;
  UINT8 HardwareMinorVersion;
  UINT8 HardwareBuildVersion

  union _UniqueHardwareId
  {
    CHAR AsChar[16];
    GUID AsGuid;
  } UniqueHardwareId;

  CHAR Description[128];
  CHAR Manufacturer[126];
  UINT16 DriverReleaseVersion

} VCIDEVICEINFO, *PVCIDEVICEINFO;
```

| | |
|---|---|
| *VciObjectId* | [out] Unique VCI ID of device. The VCI assigns a system-wide ID to every started device for the runtime. This ID serves as key for later access to the device. |
| *DeviceClass* | [out] ID of device class. Every device driver specifies its device class in form of a globally unique ID (`GUID`). Different types of devices belong to different categories. |
| *DriverMajorVersion* | [out] Major version number of device driver |
| *DriverMinorVersion* | [out] Minor version number of device driver |
| *DriverReleaseVersion* | [out] Release number of device driver |
| *DriverBuildVersion* | [out] Build number of device driver |
| *HardwareBranchVersion* | [out] Branch version number of hardware |
| *HardwareMajorVersion* | [out] Major version number of hardware |
| *HardwareMinorVersion* | [out] Minor version number of hardware |
| *HardwareBuildVersion* | [out] Build version number of hardware |
| *UniqueHardwareId* | [out] Unique ID of device. Every device has a unique ID resp. serial number which for example can be used to distinguish between two different cards of the same class. The value can be either interpreted as `GUID` or as character string. If the first two bytes contain the characters HW it is a serial number in form of a ASCII character string according to ISO-8859-1 (Latin-1). |
| *Description* | [out] Further description of device in form of a 0-terminated ASCII character string according to ISO-8859-1 (Latin-1). |
| *Manufacturer* | [out] Manufacturer ID in form of a 0-terminated ASCII character string according to ISO-8859-1 (Latin-1). |

### 7.1.4 VCIDEVICECAPS

The structure describes the technical features of a device and has the following setup:

```
typedef struct _VCIDEVICECAPS
{
  UINT16 BusCtrlCount;
  UINT16 BusCtrlTypes[VCI_MAX_BUSCTRL];
} VCIDEVICECAPS, *PVCIDEVICECAPS;
```

| | |
|---|---|
| *BusCtrlCount* | [out] Number of available bus controller. |
| *BusCtrlTypes* | [out] Table with up to `VCI_MAX_BUSCTRL` 16 bit values that describe the type of the respective controller. Valid entries in the table are in a range of 0 to *BusCtrlCount*-1. The upper 8 bits of every value of the table define the type of the supported bus, the lower 8 bits define the type of controller that is used. With the in *vcitype.h* defined makros `VCI_BUS_TYPE` resp. `VCI_CTL_TYPE` the type of the bus resp. the type of the controller can be extracted. For predefined constants for all types of bus and controller types see in *vcitype.h*. |

## 7.2 BAL Specific Data Types

The declaration of all BAL specific data types and constants is stored in the files *vcitype.h*.

### 7.2.1 BALFEATURES

The data type describes the features of the Bus Access Layer (BAL) of a controller and has the following setup:

```
typedef struct _BALFEATURES
{
  UINT16 FwMajorVersion;
  UINT16 FwMinorVersion;
  UINT16 BusSocketCount;
  UINT16 BusSocketType[BAL_MAX_SOCKETS];
} BALFEATURES, *PBALFEATURES;
```

| | |
|---|---|
| *FwMajorVersion* | [out] Major version number of BAL firmware |
| *FwMinorVersion* | [out] Minor version number of BAL firmware |
| *BusSocketCount* | [out] Number of available bus controller. |
| *BusSocketType* | [out] Table with up to `BAL_MAX_SOCKETSL` 16 bit values that describe the type of the respective controller. Valid entries in the table are in a range of 0 to *BusSocketCount*-1. The upper 8 bits of every value of the table define the type of the supported bus, the lower 8 bits define the type of controller. With the in *vcitype.h* defined macros `VCI_BUS_TYPE` resp. `VCI_CTL_TYPE` the type of the bus resp. the type of the controller can be extracted. Additionally the file contains the predefined constants for each type of possible bus and controller types. |

*If the value in field BusSocketCount doesn't coincide with the value in field `VCIDEVICECAPS.BusCtrlCount` the BAL doesn't provide all controllers that are available on the device.*

### 7.2.2 BALSOCKETINFO

The data type describes the Information about the opened bus controller and has the following structure:

```
typedef struct _BALSOCKETINFO
{
  VCIID Obid;
  UINT16 Type;
  UINT16 BusNo;
} BALSOCKETINFO, *PBALSOCKETINFO;
```

| | |
|---|---|
| *Obid* | [out] Unique ID of controller. The ID is only valid until the last reference to the superior BAL object is released. |
| *Type* | [out] Type of connection. The upper 8 bit of the value define the type of the bus the lower 8 bit the type of the controller. With the in *vcitype.h* defined macros `VCI_BUS_CTRL` resp. `VCI_CTL_TYPE` the type of the bus resp. the type of the controller can be extracted. The file *vcitype.h* contains several predefined constants for each type of possible bus and controller types |
| *BusNo* | [out] Number of bus controller. Valid values are in the range of 0 to `BALFEATURES.BusSocketCount`-1. |

## 7.3 CAN Specific Data Types

The declaration of all CAN specific data types and constants is stored in the files *cantype.h*.

### 7.3.1 CANCAPABILITIES

The data type describes the features of a CAN connection and has the following setup:

```
typedef struct _CANCAPABILITIES
{
  UINT16 wCtrlType;
  UINT16 wBusCoupling;
  UINT16 dwFeatures;
  UINT32 dwClockFreq;
  UINT32 dwTscDivisor
  UINT32 dwCmsDivisor;
  UINT32 dwMaxCmsTicks;
  UINT32 dwDtxDivisor;
  UINT32 dwMaxDtxTicks;
} CANCAPABILITIES1, *PCANCAPABILITIES;
```

| | | |
|---|---|---|
| *wCtrlType* | [out] Type of CAN controller. The value of this field is corresponding to a `CAN_TYPE_` constant defined in *cantype.h*. | |
| *wBusCoupling* | [out] Type of bus coupling. For the bus coupling the following values are defined: | |
| | `CAN_BUSC_LOWSPEED` | CAN-Controller has a low speed coupling. |
| | `CAN_BUSC_HIGHSPEED` | CAN-Controller has a high speed coupling. |
| *dwFeatures* | [out] Supported features. Value is a logical combination of one or more of the following constants: | |
| | `CAN_FEATURE_STDOREXT` | CAN controller supports 11 or 29 bit messages, but not both formats simultaneously. |
| | `CAN_FEATURE_STDANDEXT` | CAN controller supports 11 or 29 bit messages simultaneously. |
| | `CAN_FEATURE_RMTFRAME` | CAN controller supports remote transmission request (RTR) messages. |
| | `CAN_FEATURE_ERRFRAME` | CAN controller returns error messages. |

| | | |
|---|---|---|
| | CAN_FEATURE_BUSLOAD | CAN controller supports calculation of the bus load. |
| | CAN_FEATURE_IDFILTER | CAN controller allows exact filtering of messages. |
| | CAN_FEATURE_LISTONLY | CAN controller supports operating mode *Listen Only*. |
| | CAN_FEATURE_SCHEDULER | Cyclic transmission list provided. |
| | CAN_FEATURE_GENERRFRM | CAN controller supports generating of error frames. |
| | CAN_FEATURE_DELAYEDTX | CAN controller supports delayed transmitting of messages. |
| | CAN_FEATURE_ SINGLESHOT | CAN controller supports messages of type *Single Shot*. If a message is of type *Single Shot* the controller doesn't try to transmit again if the message is not transmitted with the first attempt. |
| | CAN_FEATURE_HIGHPRIOR | CAN controller supports transmitting of messages with high priority. Messages with high priority are assigned to a transmitting buffer by the controller, the transmitting buffer is prior to messages in the normal transmitting buffer. Messages of high priority are transmitted with priority to the bus. |
| | CAN_FEATURE_AUTOBAUD | CAN controller supports the automatic detection of the bit rate regarding the hardware. If this bit is set and the controller is connected to a running system, the controller detects the bit rate autonomously and it can be initialized without specifying bit timing parameters (see *CANINITLINE*). |
| *dwClockFreq* | [out] Frequency in hertz of the primary clock generator | |
| *dwTscDivisor* | [out] Divisor for the time stamp counter. Resolution of the time stamps of CAN messages is calculated by the values specified here divided by the frequency of the primary clock generator. | |
| *dwCmsDivisor* | [out] Divisor for the clock generator of the cyclic transmitting list. Frequency of cyclic transmitting list is calculated by the frequency of the primary clock generator divided by the value specified here. If no cyclic transmitting list is available the field contains value 0. | |
| *dwCmsMaxTicks* | [out] Maximum cyclic time of the cyclic transmitting list in timer ticks. If no cyclic transmitting list is available the field contains value 0. | |
| *dwDtxDivisor* | [out] Divisor for the clock generator for delayed transmitting of CAN messages. The resolution of the timer for delayed transmission is calculated by the values specified here divided by the frequency of the primary clock generator. If delayed transmitting is not supported the field contains value 0. | |
| *dwDtxMaxTicks* | [out] Maximum delay time in number of timer ticks. If delayed transmitting is not supported the field contains value 0. | |

## 7.3.2 CANCAPABILITIES2

The data type describes the features of a extended CAN connection and has the following setup:

```
typedef struct _CANCAPABILITIES2
{
  UINT16 wCtrlType;
  UINT16 wBusCoupling;
  UINT32 dwFeatures;

  UINT32 dwCanClkFreq;
  CANBTP sSdrRangeMin;
  CANBTP sSdrRangeMax;
  CANBTP sFdrRangeMin;
  CANBTP sFdrRangeMax;

  UINT32 dwTscClkFreq;
  UINT32 dwTscDivisor;

  UINT32 dwCmsClkFreq;
  UINT32 dwCmsDivisor;
  UINT32 dwCmsMaxTicks;

  UINT32 dwDtxClkFreq;
  UINT32 dwDtxDivisor;
  UINT32 dwDtxMaxTicks;
} CANCAPABILITIES2, *PCANCAPABILITIES2;
```

| | | |
|---|---|---|
| *CtrlType* | [out] Type of CAN controller. The value of this fields is corresponding to a `CAN_TYPE_` constant defined in *cantype.h*. | |
| *wBusCoupling* | [out] Type of bus coupling. For the bus coupling the following values are defined: | |
| | `CAN_BUSC_LOWSPEED` | CAN Controller has a low speed coupling. |
| | `CAN_BUSC_HIGHSPEED` | CAN Controller has a high speed coupling. |
| *dwFeatures* | [out] Supported features. Value is a logical combination of one or more of the following constants: | |
| | `CAN_FEATURE_STDOREXT` | CAN controller supports 11 or 29 bit messages, but not both formats simultaneously. |
| | `CAN_FEATURE_STDANDEXT` | CAN controller supports 11 or 29 bit messages simultaneously. |
| | `CAN_FEATURE_RMTFRAME` | CAN controller supports remote transmission request (RTR) messages. |
| | `CAN_FEATURE_ERRFRAME` | CAN controller returns error messages. |
| | `CAN_FEATURE_BUSLOAD` | CAN controller supports calculations of the bus load. |
| | `CAN_FEATURE_IDFILTER` | CAN controller allows exact filtering of messages. |
| | `CAN_FEATURE_LISTONLY` | CAN controller supports operating mode *Listen Only*. |
| | `CAN_FEATURE_SCHEDULER` | Cyclic transmitting list provided. |
| | `CAN_FEATURE_GENERRFRM` | CAN controller supports generating of error frames. |
| | `CAN_FEATURE_DELAYEDTX` | CAN controller supports delayed transmitting of messages. |
| | `CAN_FEATURE_SINGLESHOT` | CAN controller supports messages of type *Single Shot*. If a message is of type *Single Shot* the controller doesn't try to transmit again if the message is not transmitted with the first attempt. |

| | | |
|---|---|---|
| | CAN_FEATURE_HIGHPRIOR | CAN controller supports transmitting of messages with high priority. Messages with high priority are assigned to a transmitting buffer by the controller, the transmitting buffer is prior to messages in the normal transmitting buffer. Messages of high priority are transmitted with priority to the bus. |
| | CAN_FEATURE_AUTOBAUD | CAN controller supports the automatic detection of the bit rate. If this bit is set, the controller detects if it is connected to a running system, the bit rate autonomously and it can be initialized without specifying bit timing parameters (see *CANINITLINE2*). |
| | CAN_FEATURE_EXTDATA | CAN controller provides messages with extended data field. If this bit is not set at a CAN FD controller it supports maximally 8 byte in the data field. |
| | CAN_FEATURE_FASTDATA | CAN controller supports transmission with high data bit rate. |
| | CAN_FEATURE_ISOFRAME | CAN controller supports ISO-conform frames (exclusively CAN FD). |
| | CAN_FEATURE_NONISOFRM | CAN controller supports non-ISO-conform frames (exclusively CAN FD). |
| *dwCanClockFreq* | [out] Frequency in Hertz of bit rate generator. The bit rate generator defines together with the values in the structure *CANBTP* the bit transmission rate for the standard resp. for the nominal arbitration bit rate and the high data bit rate. | |
| *sSdrRangeMin* | [out] Lower limit value to specify the bit rate generator for the standard resp. the nominal arbitration bit rate. | |
| *sSdrRangeMax* | [out] Upper limit value to specify the bit rate generator for the standard resp. the nominal arbitration bit rate. | |
| *sFdrRangeMin* | [out] Lower limit value to specify the bit rate generator for the high data bit rate. All fields of the structure contain the value 0 if the controller do not support a high data bit rate. See CAN_FEATURE_FASTDATA. | |
| *sFdrRangeMax* | [out] Upper limit value to specify the bit rate generator for the high data bit rate. All fields of the structure contain the value 0 if the controller do not support a high data bit rate. See CAN_FEATURE_FASTDATA. | |
| *dwTscClockFreq* | [out] Frequency in Hertz of clock generator which is used to create the time stamps of CAN messages (Time Stamp Counter). | |
| *dwTscDivisor* | [out] Divisor for clock generator for creating time stamps. Resolution of the time stamps of CAN messages is calculated by the values specified here divided by the frequency of the time stamp counter. | |
| *dwCmsClockFreq* | [out] Frequency in Hertz of the clock generator of the cyclic transmitting list (Cyclic Message Timer). If no cyclic transmitting list is available the field contains value 0. | |
| *dwCmsDivisor* | [out] Divisor for the clock generator of the cyclic transmitting list. Frequency of cyclic transmitting list is calculated by the frequency of the cyclic message timer divided by the value specified here. If no cyclic transmitting list is available the field contains value 0. | |
| *dwCmsMaxTicks* | [out] Maximum cyclic time of the cyclic transmitting list in timer ticks. If no cyclic transmitting list is available the field contains value 0. | |
| *dwDtxClockFreq* | [out] Frequency in Hertz of clock generator, that is used for delayed transmission of CAN messages (Delay Timer). If delayed transmission is not supported the field contains value 0. | |
| *dwDtxDivisor* | [out] Divisor for the clock generator for delayed transmission of messages. The resolution of the timer for delayed transmission of messages is calculated by the values specified here divided by the frequency of the delay timer. If delayed transmission is not supported the field contains value 0. | |
| *dwDtxMaxTicks* | [out] Maximum delay time in number of timer ticks. If delayed transmission is not supported the field contains value 0. | |

### 7.3.3 CANBTRTABLE

The data structure serves to determine the bit rate and is used by the function `ICanControl::DetectBaud`. The structure has the following setup:

```
typedef struct _CANBTRTABLE
{
  UINT8 bCount;
  UINT8 bIndex;
  UINT8 abBtr0[64];
  UINT8 abBtr1[64];
} CANBTRTABLE, *PCANBTRTABLE;
```

| | |
|---|---|
| *bCount* | [in] Number of valid entries in the tables *abBtr0* and *abBtr1*. The first valid value has to be set in *abBtr0[0]* resp. *abBtr1[0]*. |
| *bIndex* | [in/out] If run successfully `DetectBaud` returns in this field the table index of the detected bus timing values. Before calling additional characters for the CAN operating mode used in `DetectBaud` can be specified here. Valid are exclusively `CAN_OPMODE_LOWSPEED` or 0, if no low speed coupling is desired. |
| *abBtr0* | [in] Table with up to 64 values for the bus timing register 0. This values are used to determine the actual transmission rate on the bus. The value of an entry corresponds to the *BT0* register of Philips SJA 1000 CAN controller with a clock frequency of 16 MHz. |
| *abBtr1* | [in] Table with up to 64 values for the bus timing register 1. This values are used to determine the actual transmission rate on the bus. The value of an entry corresponds to the *BT1* register of Philips SJA 1000 CAN controller with a clock frequency of 16 MHz. |

### 7.3.4 CANBTP

The data structure defines the parameters to specify the bit transmission rate and the sampling point and has the following setup.

```
typedef struct _CANBTP
{
   UINT32 dwMode;
   UINT32 dwBPS;
   UINT16 wTS1;
   UINT16 wTS2;
   UINT16 wSJW;
   UINT16 wTDO;
} CANBTP, *PCANBTP;
```

| | | |
|---|---|---|
| *dwMode* | [in] Operating mode. This bit field determines how the following fields are interpreted. For the operating mode a logical combination of one or more of the following constants can be specified: | |
| | `CAN_BTMODE_RAW` | Native mode. The fields *dwBPS*, *wTS1*, *wTS2*, *wSJW* and *wTDO* contain hardware specific values for the corresponding registers of the controller. The values of these fields must be inside the limits which are determined by the fields *sSdrRangeMin* resp. *sFdrRangeMin* and *sSdrRangeMax* resp. *sFdrRangeMax* of structure *CANCAPABILITIES2*. |
| | `CAN_BTMODE_TSM` | Activating triple sampling mode |
| *dwBPS* | [in] Transmitting rate in bits per second. If in field *dwMode* the bit `CAN_BTMODE_RAW` is set, the hardware specific value for the baud rate prescaler register is expected here. If not, the bit rate in bits per second is expected. | |
| *wTS1* | [in] Length of time segment 1. If in field *dwMode* the bit `CAN_BTMODE_RAW` is set, the hardware specific number of time quanta for the time segment 1 is expected here. If not, the value defines the length of this time segment in relation to the total number of time quanta per bit. | |
| *wTS2* | [in] Length of time segment 2. If in field *dwMode* the bit `CAN_BTMODE_RAW` is set, the hardware specific number of time quanta for the time segment 2 is expected here. If not, the value defines the length of this time segment in relation to the total number of time quanta per bit. | |
| *wSJW* | [in] Jump width for re-synchronization. If in field *dwMode* the bit `CAN_BTMODE_RAW` is set, the hardware specific number of time quanta for the re-synchronization is expected here. If not, the value defines the length of the jumping width in relation to the total number of time quanta per bit. | |
| *wTDO* | [in] Offset to the transceiver delay automatically determined by the controller. If in field *dwMode* the bit `CAN_BTMODE_RAW` is set, the hardware specific number of time quanta for the transceiver delay is expected here. If not, the value defines the length of the transceiver delay offset in relation to the total number of time quanta per bit. This value is exclusively relevant in case of fast data bit rate. | |

### 7.3.5 CANBTPTABLE

The data structure serves to detect the nominal bit rate and the fast bit rate if supported by the computer. Structure is used by function `ICanControl2::DetectBaud` and has the following setup:

```
typedef struct _CANBTPTABLE
{
  UINT8 bCount;
  UINT8 bIndex;
  struct
  {
    CANBTP sSdr;
    CANBTP sFdr;
  } asBTP[64];
} CANBTPTABLE, *PCANBTPTABLE;
```

| | |
|---|---|
| *bCount* | [in] Number of valid values in table *asBTP*. The first valid values have to be set in *asBtP[0]*. |
| *bIndex* | [out] Table index with the bit timing parameters of the detected bit rate. |
| *asBTP* | [in] Table with up to 64 values of different default values, that can be used to determine the actual bit transmission rate on the bus. The table contains the paired bit timing parameters for the default and the nominal bit rate in field *sSdr* and the fast bit rate in field *sFdr*. The values for the fast data bit rate are only relevant if the controller supports this and if in parameter *bExMode* the value `CAN_EXMODE_FASTDATA` is specified when calling the function `ICanControl2::DetectBaud`. Otherwise the values in *sFdr* have no significance. |

### 7.3.6 CANINITLINE

The structure is used to initialize the CAN control unit and has the following setup:

```
typedef struct _CANINITLINE
{
  UINT8 bOpMode;
  UINT8 bReserved;
  UINT8 bBtReg0; UINT8 bBtReg1;
} CANINITLINE, *PCANINITLINE;
```

| | | |
|---|---|---|
| *bOpMode* | [in] Operating mode of controller. For the operating mode a logical combination of one or more of the following constants can be specified: | |
| | `CAN_OPMODE_STANDARD` | Controller accepts messages with 11 bit identifier. |
| | `CAN_OPMODE_EXTENDED` | Controller accepts messages with 29 bit identifier. |
| | `CAN_OPMODE_LISTONLY` | Controller is used in *Listen Only* mode. |
| | `CAN_OPMODE_ERRFRAME` | Errors are reported to the application via special messages. |
| | `CAN_OPMODE_LOWSPEED` | Controller uses low speed bus coupling. |
| | `CAN_OPMODE_AUTOBAUD` | If supported by the controller the controller performs an automatic detection of the bit rate during the initialization. Controller must be connected with running system. If this bit is set the bit timing parameters specified in the fields *bBtReg0* and *bBtReg1* are ignored. |
| *bReserved* | [in] Reserved. Value must be initialized with 0. | |
| *bBtReg0* | [in] Value for the bus timing register 0 of the controller. Value corresponds to BTR0 register of Philips SJA 1000 CAN controllers with a clock frequency of 16 MHz. Further information see data sheet of SJA 1000. | |
| *bBtReg1* | [in] Value for the bus timing register 1 of the controller. Value corresponds to BTR1 register of Philips SJA 1000 CAN controllers with a clock frequency of 16 MHz. Further information see data sheet of SJA 1000. | |

### 7.3.7 CANINITLINE2

The structure is used to initialize the extended CAN control unit and has the following setup:

```
typedef struct _CANINITLINE2
 {
   UINT8  bOpMode;
   UINT8  bExMode;
   UINT8  bSFMode;
   UINT8  bEFMode;
   UINT32 dwSFIds;
   UINT32 dwEFIds;
   CANBTP sBtpSdr;
   CANBTP sBtpFdr;
 } CANINITLINE2, *PCANINITLINE2;
```

| | | |
|---|---|---|
| *bOpMode* | [in] Operating mode of controller. For the operating mode a logical combination of one or more of the following constants can be specified: | |
| | CAN_OPMODE_STANDARD | Controller accepts messages with 11 bit identifier. |
| | CAN_OPMODE_EXTENDED | Controller accepts messages with 29 bit identifier. |
| | CAN_OPMODE_LISTONLY | Controller is used in *Listen Only* mode. |
| | CAN_OPMODE_ERRFRAME | Errors are reported to the application via special messages. |
| | CAN_OPMODE_LOWSPEED | Controller uses low speed bus coupling. |
| | CAN_OPMODE_AUTOBAUD | If supported by the controller the controller performs an automatic detection of the bit rate during the initialization. Controller must be connected with running system. If this bit is set the bit timing parameters specified in the fields *sBtpSdr* and *sBtpFdr* are ignored. |
| *bExMode* | [in] Extended operating mode. If supported by the controller, a logical combination of one or more of the following constants can be specified: | |
| | CAN_EXMODE_EXTDATA | Allows messages with extended data length up to 64 bytes. |
| | CAN_EXMODE_FASTDATA | Allows higher bit rates for the data field. |
| | CAN_EXMODE_NONISO: | Usage of non-ISO-conform message frames. This option is exclusively available with older CAN FD controller with the feature CAN_FEATURE_NONISOFRM. |
| | If the value CAN_EXMODE_DISABLED is specified no further operating mode is activated. The value also must be specified with all other controllers that do not support CAN FD operating mode. Further information see description of field *dwFeatures* of structure *CANCAPABILITIES2*. | |
| *bSFMode* | [in] Default value for the operating mode of 11 bit filter. Operating mode can also be changed with function ICanControl2::SetFilterMode. | |
| *bEFMode* | [in] Default value for the operating mode of 29 bit filter. Operating mode can also be changed with function ICanControl2::SetFilterMode. | |
| *dwSFIds* | [in] Number of CAN IDs supported by the 11 bit filter. With value 0 not filter is specified. Controller allows all messages with 11 bit ID to pass. The operating mode specified in *bSFMode* is not considered. | |
| *dwEFIds* | [in] Number of CAN IDs supported by the 29 bit filter. With value 0 not filter is specified. Controller allows all messages with 29 bit ID to pass. The operating mode specified in *bEFMode* is not considered. | |
| *sBtpSdr* | [in] Bit timing parameter for default or nominal bit rate resp. for bit rate during the arbitration phase. For further information see description of data type *CANBTP*. | |
| *sBtpFdr* | [in] Bit timing parameter for fast data bit rate. Field is exclusively relevant if the controller supports the fast data transmission and if constant CAN_EXMODE_FASTDATA in field *bExMode* is specified. For further information see description of data type *CANBTP*. | |

### 7.3.8 CANLINESTATUS

The data type describes the current status of a CAN control unit and has the following structure:

```
typedef struct _CANLINESTATUS
{
  UINT8 bOpMode;
  UINT8 bBtReg0;
  UINT8 bBtReg1;
  UINT8 bBusLoad;
  UINT32 dwStatus;
} CANLINESTATUS, *PCANLINESTATUS;
```

| | | |
|---|---|---|
| *bOpMode* | [in] Current operating mode of controller. Value is a logical combination of one or more in *cantype.h* defined constants `CAN_OPMODE_` and corresponds to the value of the field *bOpMode* specified in parameter *pInitLine*when calling the function `ICanControl::InitLine`. | |
| *bBtReg0* | [out] Current value Bus-Timing-Register 0. Value corresponds to *BTR0* register of Philips SJA 1000 CAN controllers with a clock frequency of 16 MHz. Further information see data sheet of SJA 1000. | |
| *bBtReg1* | [out] Current value bus timing register 1. Value corresponds to *BTR1* register of Philips SJA 1000 CAN controllers with a clock frequency of 16 MHz. Further information see data sheet of SJA 1000. | |
| *bBusLoad* | [out] Current bus load in percentage (0 to 100). Value is exclusively valid if calculation of bus load is supported by the controller. For further information see *CANCAPABILITIES*. | |
| *dwStatus* | [out] Current status of CAN controller. Value is a logical combination of one or more of the following constants: | |
| | `CAN_STATUS_TXPEND` | CAN controller is currently transmitting a message to the bus. |
| | `CAN_STATUS_OVRRUN` | Data overflow in the receiving buffer of the CAN controller had happened. |
| | `CAN_STATUS_ERRLIM` | Overflow of an error counter of the CAN controller has happened. |
| | `CAN_STATUS_BUSOFF` | CAN controller has shifted to state *BUS-OFF*. |
| | `CAN_STATUS_ININIT` | CAN controller is in stopped state. |
| | `CAN_STATUS_BUSCERR` | Faulty bus coupling |

### 7.3.9 CANLINESTATUS2

The data type describes the current status of a CAN control unit and has the following structure:

```
typedef struct _CANLINESTATUS
{
  UINT8 bOpMode;
  UINT8 bExMOde;
  UINT8 bBusLoad;
  UINT8 bReserved;
  CANBTP sBtpSdr;
  CANBTP sBtpFdr;
  UINT32 dwStatus;
} CANLINESTATUS2, *PCANLINESTATUS2;;
```

| | | |
|---|---|---|
| *bOpMode* | [in] Current operating mode of controller. Value is a logical combination of one or more in *cantype.h* defined constants `CAN_OPMODE_` and corresponds to the value of the field *bOpMode* specified in parameter *pInitLine*when calling the function `ICanControl2::InitLine`. | |
| *bExMode* | [in] Current extended operating mode of controller. Value is a logical combination of one or more in *cantype.h* defined constants `CAN_EXMODE_` and corresponds to the value of the field *bExMode* specified in parameter *pInitLine*when calling the function `ICanControl2::InitLine`. | |
| *bBusLoad* | [out] Current bus load in percentage (0 to 100). Value is exclusively valid if calculation of bus load is supported by the controller. For further information see *CANCAPABILITIES2*. | |
| *bReserved* | Not used (normally 0). | |
| *sBtpSdr* | [out] Current bit timing parameter for nominal bit rate resp. for bit rate during the arbitration phase. | |
| *sBtpFdr* | [out] Current bit timing parameter for fast data bit rate. | |
| *dwStatus* | [out] Current status of CAN controller. Value is a logical combination of one or more of the following constants: | |
| | `CAN_STATUS_TXPEND` | CAN controller is currently transmitting a message to the bus. |
| | `CAN_STATUS_OVRRUN` | Data overflow in the receiving buffer of the CAN controller had happened. |
| | `CAN_STATUS_ERRLIM` | Overflow of an error counter of the CAN controller has happened. |
| | `CAN_STATUS_BUSOFF` | CAN controller has shifted to state *BUS-OFF*. |
| | `CAN_STATUS_ININIT` | CAN controller is in stopped state. |
| | `CAN_STATUS_BUSCERR` | Faulty bus coupling |

### 7.3.10 CANCHANSTATUS

The data type describes the current status of the CAN message channel and has the following setup:

```
typedef struct _CANLINESTATUS
{
  CANLINESTATUS sLineStatus;
  BOOL32 fActivated;
  BOOL32 fRxOverrun;
  UINT8 bRxFifoLoad;
  UINT8 bTxFifoLoad;
} CANCHANSTATUS, *PCANCHANSTATUS;
```

| | |
|---|---|
| *sLineStatus* | [out] Current status of CAN controller. For further information see *CANLINESTATUS*. |
| *fActivated* | [out] Shows if message channel is active (TRUE) or inactive (FALSE). |
| *fRxOverrun* | [out] Signalizes an overflow in the receiving buffer with the value TRUE. |
| *bRxFifoLoad* | [out] Current filling level of receiving FIFO in percentage |
| *bTxFifoLoad* | [out] Current filling level of transmitting FIFO in percentage |

### 7.3.11 CANCHANSTATUS2

The data type describes the current status of the CAN message channel with extended interface and has the following setup:

```
typedef struct _CANCHANSTATUS2
{
  CANLINESTATUS sLineStatus;
  BOOL8 fActivated;
  BOOL8 fRxOverrun;
  UINT8 bRxFifoLoad;
  UINT8 bTxFifoLoad;
} CANCHANSTATUS, *PCANCHANSTATUS2;
```

| | |
|---|---|
| *sLineStatus* | [out] Current status of CAN controller. For further information see *CANLINESTATUS2*. |
| *fActivated* | [out] Shows if message channel is active(TRUE) or inactive (FALSE). |
| *fRxOverrun* | [out] Signalizes an overflow in the receiving buffer with the value TRUE. |
| *bRxFifoLoad* | [out] Current filling level of receiving FIFO in percentage |
| *bTxFifoLoad* | [out] Current filling level of transmitting FIFO in percentage |

## 7.3.12 CANSCHEDULERSTATUS

The data type describes the current status of the cyclic transmitting list and has the following setup:

```
typedef struct _CANSCHEDULERSTATUS
{
  UINT8 bTaskStat;
  UINT8 abMsgStat[16];
} CANSCHEDULERSTATUS, *PCANSCHEDULERSTATUS;
```

| | | |
|---|---|---|
| *bTaskStat* | [out] Current status of transmitting task | |
| | CAN_CTXTSK_STAT_STOPPED | Transmitting task is stopped resp. deactivated. |
| | CAN_CTXTSK_STAT_RUNNING | Transmitting task is performed resp. is active. |
| *abMsgStat* | Table with status of all 16 transmitting objects. Each table entry can take one of the following values: | |
| | CAN_CTXMSG_STAT_EMPTY | The entry is not assigned to a transmitting object resp. the entry is currently not used. |
| | CAN_CTXMSG_STAT_BUSY | Transmitting object is currently processed. |
| | CAN_CTXMSG_STAT_DONE | Processing of transmitting object is finished. |

## 7.3.13 CANSCHEDULERSTATUS2

The data type describes the current status of the cyclic transmitting list and has the following setup:

```
typedef struct _CANSCHEDULERSTATUS2
 {
 CANLINESTATUS2 sLineStatus;
 UINT8 bTaskStat;
 UINT8 abMsgStat[16];
 }
 CANSCHEDULERSTATUS2, *PCANSCHEDULERSTATUS2;
```

| | | |
|---|---|---|
| *SLineStatus* | [out] Current state of CAN controller. For further information see data structure *CANLINESTATUS2*. | |
| *bTaskStat* | [out] Current status of transmitting task | |
| | CAN_CTXTSK_STAT_STOPPED | Transmitting task is stopped resp. deactivated. |
| | CAN_CTXTSK_STAT_RUNNING | Transmitting task is performed resp. is active. |
| *abMsgStat* | Table with status of all 16 transmitting objects. Each table entry can take one of the following values: | |
| | CAN_CTXMSG_STAT_EMPTY | The entry is not assigned to a transmitting object resp. the entry is currently not used. |
| | CAN_CTXMSG_STAT_BUSY | Transmitting object is currently processed. |
| | CAN_CTXMSG_STAT_DONE | Processing of transmitting object is finished. |

### 7.3.14 CANMSGINFO

The data type summarizes different information about CAN messages in a union. The individual values can either be addresses via byte fields or via bus bit fields.

```
typedef union _CANMSGINFO
{
  struct
  {
    UINT8 bType;
    union
    {
      UINT8 bReserved;
      UINT8 bFlags2;
    };
    UINT8 bFlags;
    UINT8 bAccept;
  } Bytes;

  struct
  {
    UINT32 type: 8;

    UINT32 ssm : 1;
    UINT32 hpm : 1;
    UINT32 edl : 1;
    UINT32 fdr : 1;
    UINT32 esi : 1;
    UINT32 res : 3;

    UINT32 dlc : 4;
    UINT32 ovr : 1;
    UINT32 srr : 1;
    UINT32 rtr : 1;
    UINT32 ext : 1;

    UINT32 afc : 8;
  } Bits;

}CANMSGINFO, *PCANMSGINFO;
```

The information are accessed byte by byte via the following byte fields:

| | |
|---|---|
| *Bytes.bType* | [in/out] Type of message. See *bits.type*. |
| *Bytes.bReserved* | Reserved. Due to compatibility reasons set field always to 0. See *bits.res*. |
| *Bytes.bFlags2* | [in/out] Extended message flags. Meaning of the field is described in description of the respective bit fields *Bits.ssm, Bits.hpm, Bits.edl, Bits.fdr, Bits.esi and Bits.res*. |
| *Bytes.bFlags2* | [in/out] Standard message flags. Meaning of the field is described in description of the respective bit fields *Bits.dlc*, *Bits.ovr*, *Bits.srr*, *Bits.rtr* and *Bits.ext*. |
| *Bytes.bAccept* | [out] Shows in receive messages which filter has accepted the message. See *bits. afc*. |

The information are accessed byte by byte via the following bit fields:

| | | |
|---|---|---|
| *Bits.type* | [in/out] Type of message. For the receiving messages the following types are defined: | |
| | CAN_MSGTYPE_DATA | Standard data message. In receive messages the field *dwMsgId* contains the ID of the message and the field *dwTime* the receiving time in ticks. The field *abData* contains depending on the length (see *bits.dlc*) the data bytes of the message.<br>In transmit messages the field *dwMsgId* contains the message ID and the fields *abData* the data bytes to be transmitted. The field *dwTime* is set to 0. If the message is to be transmitted delayed to the message transmitted before specify desired delay time in ticks. See *Transmitting Messages Delayed, p. 30*. |
| | CAN_MSGTYPE_INFO | Information message. This message type is generated by certain events resp. state changes of the control unit and registered in the receiving buffers of all active message channels. The field *dwMsgId* of the message always contains the value *CAN_MSGID_INFO*. Additionally the field *abData[0]* contains one of the following values: |
| | | **Constant** / **Meaning**<br>CAN_INFO_START — Controller is started. Field *dwTime* contains the relative starting point.<br>CAN_INFO_STOP — Controller is stopped. Field *dwTime* contains value 0.<br>CAN_INFO_RESET — Controller is reset. Field *dwTime* contains value 0. |
| | CAN_MSGTYPE_ERROR | Error message. This message type is generated if a bus error occurs and is registered in the receiving buffers off all active message channels, provided that the flag CAN_OPMODE_ERRFRAME is set during the initialization of the CAN controller. The field *dwMsgId* of the message contains the value CAN_MSGID_ERROR. The time of the event is noted in field *dwTime*. The field *abData[0]* contains one of the following values: |
| | | **Constant** / **Meaning**<br>CAN_ERROR_STUFF — Stuff error<br>CAN_ERROR_FORM — Format error<br>CAN_ERROR_ACK — Acknowledge error<br>CAN_ERROR_BIT — Bit error<br>CAN_ERROR_FDB — Fast data bit error<br>(CANFD) CAN_ERROR_CRC — CRC error<br>CAN_ERROR_DLC — Faulty data length<br>CAN_ERROR_OTHER — Another, not specified error |
| | | Additionally the field *abData[1]* contains the low byte of the current CAN state. See description of field *dwStatus* of structure *CANCAPABILITIES* or *CANCAPABILITIES2*. The content of all other data fields is undefined. |
| | CAN_MSGTYPE_STATUS | Status message. This message type is generated by state changes of the CAN controller and registered in the receiving buffers of all active message channels. The field dwMsgId contains the value CAN_MSGID_STATUS. The time of the event is noted in field *dwTime*. Additionally the field *abData[0]* contains the low byte of the current CAN state. The content of the other data fields is undefined. |
| | CAN_MSGTYPE_WAKEUP | This message type is not used resp. reserved for future expansions. |
| | CAN_MSGTYPE_TIMEOVR | Messages of this type are generated by the time stamp counter with every overflow and registered in the receiving buffer of all active message channels. The field *dwTime* of the message contains the time of the event and the field *dwMsgId* the number of occurred overflows (normally 1). The content of the data fields *abData* is undefined. |
| | CAN_MSGTYPE_TIMERST | This message type is not used resp. reserved for future expansions.<br>For transmitting messages exclusively the message type CAN_MSGTYPE_DATA is defined, other values are not valid. |
| *Bits.ssm* | [in] Single shot message. If this bit is set in transmitting messages the controller tries to transmit the message only once. If the message loses its arbitration during the first transmitting attempt, the controller rejects the message and no further automatic transmitting attempt follows. If this bit is 0 no transmitting is attempted until the message has been transmitted over the bus. Regarding receiving messages this bit has no significance. | |

| | |
|---|---|
| *Bits.hpm* | [in] High priority message. Transmitting messages with high priority are assigned to a transmitting buffer by the controller, the transmitting buffer is prior to messages in the normal transmitting buffer. Messages of high priority are transmitted with priority to the bus. Regarding receiving messages this bit has no significance. |
| *Bits.edl* | [in/out] Message with extended data field. For further information see description of data length field *Bits.dlc*. The bit is exclusively valid with extended controller operating mode CAN_EXMODE_EXTDATA. |
| *Bits.fdr* | [in/out] This bit can be set in transmitting messages to transfer the data bytes and bis from the DLC field with high bit rate on the bus. If this bit is set the RTR bit is ignored. See description of *bits.rtr* The bit is exclusively valid with extended controller operating mode CAN_EXMODE_FASTDATA. |
| *Bits.esi* | [out] Error state indicator. Nodes that are *error active* transmit this bit dominant (0), nodes that are *error passive* recessive (1). This bit is exclusively considered in receiving messages. In transmitting messages it is has no significance and must be set to 0. |
| *Bits.res* | Reserved for further extensions. Due to compatibility reasons set field always to 0. |
| *Bits.dlc* | [in/out] Data length code. The value defines the number of valid data bytes in field *abData* of a message. The following assignment applies: |

| ***dlc*** | **Number of data bytes** |
|---|---|
| 0...8 | 0...8 |
| 9 | 12 |
| 10 | 16 |
| 11 | 20 |
| 12 | 24 |
| 13 | 32 |
| 14 | 48 |
| 15 | 64 |

A value higher than 8 is exclusively allowed in messages with extended data field (see *CANMSG2*). To transmit a message with more than 8 byte the CAN must be used in the operating mode CAN_EXMODE_EXTDATA and additionally the bit *edl* of the message to be transmitted must be set to 1. Basically this is exclusively possible with controllers with extended functionality (CAN FD).

| | |
|---|---|
| *Bits.ovr* | [out] Data overrun. The bit is set to 1 in receiving messages if an overflow of the receiving FIFO took place. |
| *Bits.srr* | [in/out] Self reception request. If the bit is set in transmitting messages the message is assigned to the receiving FIFO as soon as it was transmitted to the bus. In receiving messages a set bit indicates that it is a self reception message. This bit must not be mistaken as substitute remote request (SRR) bit of CAN FD. |
| *Bits.rtr* | [in/out] Remote transmission request. This bit is set in transmitting messages to scan other bus participants specifically for certain messages. Observe that the bit is ignored if one of the bits *edl* or*fdr* is also set. RTR messages are not possible with CAN FD. |
| *Bits.ext* | [in/out] Messages with extended 29 bit ID. |
| *Bits.afc* | [out] Acceptance filter code. In receiving messages this field determines the filter that lets pass the message. The following values are defined: |

| | |
|---|---|
| CAN_ACCEPT_ALWAYS | The message is always accepted. All other messages than these of type CAN_MSGTYPE_DATA contain this value. |
| CAN_ACCEPT_FILTER1 resp. CAN_ACCEPT_FILTER2 | The message has either been accepted by the acceptance filter (CAN_ACCEPT_FILTER1) or by the filter list (CAN_ACCEPT_FILTER2). Exclusively messages of type CAN_MSGTYPE_DATA contain this value. The filter must be used in operating mode CAN_FILTER_INCL. |
| CAN_ACCEPT_EXCL | This value is used in the filter operating mode CAN_FILTER_EXCL if a message of type CAN_MSGTYPE_DATA has been accepted.<br>Detailed information about functionality of message filters and the different operating modes see *Message Filter, p. 40*. |

### 7.3.15 CANMSG

The data type describes the structure of CAN message telegrams.

```
typedef struct _CANMSG
{
  UINT32 dwTime;
  UINT32 dwMsgId;
  CANMSGINFO uMsgInfo;
  UINT8 abData[8];
} CANMSG, *PCANMSG;
```

| | |
|---|---|
| *dwTime* | In receiving messages this field contains the relative starting point of the message in ticks. In transmitting messages this fields determines with how many ticks delay the message is transmitted after the message sent before. For further information see *Message Channels, p. 23*. |
| *dwMsgId* | CAN ID of the message in Intel format (aligned right) without RTR bit. |
| *uMsgInfo* | Bit field with information about the message type. For detailed description of bit field see *CANMSGINFO*. |
| *abData* | Array for up to 8 data bytes. Number of valid data bytes is defined by field *uMsgInfo.Bits.dlc*. |

### 7.3.16 CANMSG2

The data type describes the structure of extended CAN message telegrams.

```
typedef struct _CANMSG2
{
  UINT32 dwTime;
  UINT32 dwMsgId;
  CANMSGINFO uMsgInfo;
  UINT8 abData[64];
} CANMSG2, *PCANMSG2;
```

| | |
|---|---|
| *dwTime* | In receiving messages this field contains the relative receiving point of the message in ticks. In transmitting messages this fields determines with how many ticks delay the message is transmitted after the message sent before. For further information see *Message Channels, p. 23*. |
| *dwMsgId* | CAN ID of the message in Intel format (aligned right) without RTR bit. |
| *uMsgInfo* | Bit field with information about the message type. For detailed description of bit field see *CANMSGINFO*. |
| *abData* | Array for up to 64 data bytes. Number of valid data bytes is defined by field *uMsgInfo.Bits.dlc*. |

## 7.3.17 CANCYCLICTXMSG

This data type describes the structure of a cyclic transmitting list.

```
typedef struct _CANCYCLICTXMSG
{
  UINT16 wCycleTime;
  UINT8 bIncrMode;
  UINT8 bByteIndex;
  UINT32 dwMsgId;
  CANMSGINFO uMsgInfo;
  UINT8 abData[8];
} CANCYCLICTXMSG, *PCANCYCLICTXMSG;
```

| | | |
|---|---|---|
| *wCycleTime* | Cycle time of the message in number ticks. The cycle time can be calculated in the fields *dwClockFreq* and *dwCmsDivisor* of structure *CANCAPABILITIES* with the following formula.<br>$T_{cycle}$ [s] = (*dwCmsDivisor* / *dwClockFreq*) * *wCycleTime*<br>The maximum value for the field is limited to the value in field *dwCmsMaxTicks* of structure *CANCAPABILITIES*. | |
| *bIncrMode* | This field determines if a part of the cyclic transmitting list is automatically incremented after each transmitting. | |
| | CAN_CTXMSG_INC_NO | The message field is not incremented automatically. |
| | CAN_CTXMSG_INC_ID | Increments the field *dwMsgId* by 1 after every transmission. If the field reaches the value 2048 (11 bit ID) resp. 536.870.912 (29 bit ID) an overflow automatically takes place. |
| | CAN_CTXMSG_INC_8 | Increments an 8 bit value in the data field *abData* of the message. The data byte to be incremented is determined via the parameter *bByteIndex*. If the maximum value 255 is exceeded an overflow to 0 takes place. |
| | CAN_CTXMSG_INC_16 | Increments a 16 bit value in the data field *abData* of the message. The low byte of the 16 bit value to be incremented is determined via the field *bByteIndex*. The high byte is in the data field on position *bByteIndex*+1. If the maximum value 655350 is exceeded an overflow to 0 takes place. |
| *bByteIndex* | Field determines the byte resp. the low byte (LSB) of the 16 bit value in data field *abData*, that is automatically incremented after each transmission. The value range of the field is limited by the data length specified in the field *uMsgInfo.Bits.dlc* of structure *CANMSGINFO* and it is limited to the range 0 to (*dlc*–1) in case of 8 bit increment and 0 to (*dlc*–2) in case of 16 bit increment. | |
| *dwMsgId* | CAN ID of the message in Intel format (aligned right) without RTR bit. | |
| *uMsgInfo* | Bit field with information about the message type. For description of bit field see *CANMSGINFO*. | |
| *abData* | Array for up to 8 data bytes. Number of valid data bytes is defined by field *uMsgInfo.Bits.dlc*. | |

## 7.3.18 CANCYCLICTXMSG2

This data type describes the structure of an extended cyclic transmitting list.

```
typedef struct _CANCYCLICTXMSG2
{
  UINT16 wCycleTime;
  UINT8 bIncrMode;
  UINT8 bByteIndex;
  UINT32 dwMsgId;
  CANMSGINFO uMsgInfo;
  UINT8 abData[64];
} CANCYCLICTXMSG2, *PCANCYCLICTXMSG2;
```

| | | |
|---|---|---|
| *wCycleTime* | Cycle time of the message in number ticks. The cycle time can be calculated in the fields *dwClockFreq* and *dwCmsDivisor* of structure *CANCAPABILITIES2* with the following formula.<br>$T_{cycle}$ [s] = (*dwCmsDivisor* / *dwClockFreq*) * *wCycleTime*<br>The maximum value for the field is limited to the value in field *dwCmsMaxTicks* of structure *CANCAPABILITIES2*. | |
| *bIncrMode* | This field determines if a part of the cyclic transmitting list is automatically incremented after each transmitting. | |
| | CAN_CTXMSG_INC_NO | The message field is not incremented automatically. |
| | CAN_CTXMSG_INC_ID | Increments the field *dwMsgId* by 1 after every transmission. If the field reaches the value 2048 (11 bit ID) resp. 536.870.912 (29 bit ID) an overflow automatically takes place. |
| | CAN_CTXMSG_INC_8 | Increments an 8 bit value in the data field *abData* of the message. The data byte to be incremented is determined via the parameter *bByteIndex*. If the maximum value 255 is exceeded an overflow to 0 takes place. |
| | CAN_CTXMSG_INC_16 | Increments a 16 bit value in the data field *abData* of the message. The low byte of the 16 bit value to be incremented is determined via the field *bByteIndex*. The high byte is in the data field on position *bByteIndex*+1. If the maximum value 655350 is exceeded an overflow to 0 takes place. |
| *bByteIndex* | Field determines the byte resp. the low byte (LSB) of the 16 bit value in data field *abData*, that is automatically incremented after each transmission. The value range of the field is limited by the data length specified in the field *uMsgInfo.Bits.dlc* of structure *CANMSGINFO* and it is limited to the range 0 to (*dlc*–1) in case of 8 bit increment and 0 to (*dlc*–2) in case of 16 bit increment. | |
| *dwMsgId* | CAN ID of the message in Intel format (aligned right) without RTR bit. | |
| *uMsgInfo* | Bit field with information about the message type. For description of bit field see *CANMSGINFO*. | |
| *abData* | Array for up to 64 data bytes. Number of valid data bytes is defined by field *uMsgInfo.Bits.dlc*. | |

# 7.4 LIN Specific Data Types

The declaration of all LIN specific data types and constants is stored in the files *lintype.h*.

## 7.4.1 LINCAPABILITIES

The data type describes the features of a LIN connection and has the following setup:

```
typedef struct _LINCAPABILITIES
{
  UINT16 dwFeatures;
  UINT32 dwClockFreq;
  UINT32 dwTscDivisor;
} LINCAPABILITIES, *PLINCAPABILITIES;
```

| | | |
|---|---|---|
| *dwFeatures* | [out] Supported features. Value is a logical combination of one or more of the following constants: | |
| | LIN_FEATURE_MASTER | LIN controller supports the operating mode *Master*. |
| | LIN_FEATURE_AUTORATE | LIN controller supports the automatic detection of the bit rate. |
| | LIN_FEATURE_ERRFRAME | LIN controller returns error messages. |
| | LIN_FEATURE_BUSLOAD | LIN controller supports calculation of the bus load. |
| | LIN_FEATURE_SLEEP | LIN controller supports SLEEP messages (exclusively Master). |
| | LIN_FEATURE_WAKEUP | LIN controller supports WAKEUP messages. |
| *dwClockFreq* | [out] Frequency in hertz of the primary timer. | |
| *dwTscDivisor* | [out] Divisor for the time stamp counter. The time stamp counter returns the time stamp for LIN messages. Frequency of time stamp counter is calculated by the frequency of the primary timer divided by the value specified here. | |

## 7.4.2 LININITLINE

The structure is used to initialize a LIN controller and determines the operating mode and the transmission rate. The structure has the following setup:

```
typedef struct _LININITLINE
{
  UINT8 bOpMode;
  UINT8 bReserved;
  UINT16 wBitrate;
} LININITLINE, *PLININITLINE;
```

| | | |
|---|---|---|
| *bOpMode* | [in] Operating mode of LIN controller. For the operating mode a logical combination of one or more of the following constants can be specified: | |
| | LIN_OPMODE_SLAVE | Slave mode. By default the operating mode is always active. |
| | LIN_OPMODE_MASTER | Activate master mode (if supported see *LINCAPABILITIES*). |
| | LIN_OPMODE_ERRORS | Errors are reported to the application via special LIN messages. |
| *bReserved* | [in] Reserved. Value must be initialized with 0. | |
| *wBitrate* | [in] Transmitting rate in bits per second. The specified value must be in between the limits that are determined by the constants LIN_BITRATE_MIN and LIN_BITRATE_MAX. If the controller is used as slave and supports an automatic bit detection the bit rate can be determined automatically by setting the value LIN_BITRATE_AUTO. | |

### 7.4.3 LINLINESTATUS

The data type describes the current status of the LIN message and has the following setup:

```
typedef struct _LINLINESTATUS
{
  UINT8 bOpMode;
  UINT8 bReserved;
  UINT16 wBitrate;
  UINT32 dwStatus;
} LINLINESTATUS, *PLINLINESTATUS;
```

| | | |
|---|---|---|
| *bOpMode* | [in] Current operating mode of controller. Value is a logical combination of one or more in `lintype.h` defined constants *LIN_OPMODE_* and corresponds to the values specified in `ILinControl::InitLine`. | |
| *bReserved* | [out] Not used. | |
| *wBitrate* | [out] Currently specified transmission rate in bits per second. | |
| *dwStatus* | [out] Current status of LIN controller. Value is a logical combination of one or more of the following constants: | |
| | `LIN_STATUS_TXPEND` | Controller is currently transmitting a message to the bus. |
| | `LIN_STATUS_OVRRUN` | Data overflow in the receiving buffer of the controller had happened. |
| | `LIN_STATUS_ERRLIM` | Overflow of an error counter of the controller has happened. |
| | `LIN_STATUS_BUSOFF` | Controller has shifted to state *BUS-OFF*. |
| | `LIN_STATUS_ININIT` | Controller is in stopped state. |

### 7.4.4 LINMONITORSTATUS

The data type describes the current status of the LIN message monitor and has the following setup:

```
typedef struct _LINMONITORSTATUS
{
  LINLINESTATUS sLineStatus;
  BOOL32 fActivated;
  BOOL32 fRxOverrun;
  UINT8 bRxFifoLoad;
} LINMONITORSTATUS, *PLINMONITORSTATUS;
```

| | |
|---|---|
| *sLineStatus* | [out] Current status of LIN controller. Further information see description of data structure *LINLINESTATUS*. |
| *fActivated* | [out] Shows if message monitor is active(`TRUE`) or inactive (`FALSE`). |
| *fRxOverrun* | [out] Signalizes an overflow in the receiving buffer with the value `TRUE`. |
| *bRxFifoLoad* | [out] Current filling level of receiving FIFO in percentage. |

### 7.4.5 LINMSGINFO

The data type summarizes different information about LIN messages in a 32 bit value. The value can be assigned byte by byte or via individual bit fields.

```
typedef union _LINMSGINFO
{
  struct
  {
    UINT8 bPid;
    UINT8 bType;
    UINT8 bDlen;
    UINT8 bFlags; } Bytes;

  struct
  {
    UINT32 pid : 8;
    UINT32 type : 8;
    UINT32 dlen : 8;
    UINT32 ecs : 1;
    UINT32 sor : 1;
    UINT32 ovr : 1;
    UINT32 ido : 1;
    UINT32 res : 4;
  } Bits;

} LINMSGINFO, *PLINMSGINFO;
```

The information of a LIN message can be accessed byte by byte via the structure element *Bytes*. The following fields are defined:

| | |
|---|---|
| *Bytes.bPid* | [in/out] Protected identifier. See *bits.pid*. |
| *Bytes.bType* | [in/out] Type of message. See *bits-type* and *bits.ecs*. |
| *Bytes.bDlen* | [in/out] Data length, see *bits.dlen*. |
| *Bytes.bFlags* | [in/out] Different flags, see *bits.ecs*, *bits.sor*, *bits.ovr* and *bits.ido*. |

The information of a LIN message can be accessed by the bit via the structure element *Bits*. The following bit fields are defined:

| | | |
|---|---|---|
| *Bits.pid* | [in/out] Protected identifier of the message. | |
| *Bits.type* | [in/out] Type of message. For the receiving messages the following types are defined: | |
| | `LIN_MSGTYPE_DATA` | Standard message. All regular receiving channels are of this type. In field *bPid* is the ID of the message, in field *dwTime* the receiving time. The field *abData* contains depending on the length (see *bits.dlen*) the data bytes of the message. In the master operating mode messages of this type can also be transmitted. Therefore the ID must be specified in field *bPid* and in field *abData* depending on the length (*bits.dlen*) the data to be transmitted. The field *dwTime* is set to 0. To transmit exclusively the ID without data *Bits.ido* is set to 1. |

| | | |
|---|---|---|
| *Bits.type* | LIN_MSGTYPE_INFO | Information message. This message type is assigned in the receiving buffers of all active message monitors if certain events happen or the status of the controller is changed. The field *bPid* of the message contains the value 0xFF. The field *abData[0]* contains one of the following values:<br>**Constant** — **Meaning**<br>LIN_INFO_START — Controller is started. Field *dwTime* contains the relative starting point (normally 0).<br>LIN_INFO_STOP — Controller is stopped. Field *dwTime* contains value 0.<br>LIN_INFO_RESET — Controller is reset. Field *dwTime* contains value 0. |
| | LIN_MSGTYPE_ERROR: | Error message. This message type is generated if a bus error occurs and is registered in the receiving buffers off all active message channels, provided that the flag CAN_OPMODE_ERRORS is set during the initialization of the controller. The field *bPid* of the message has the value 0xFF. The time of the event is noted in field *dwTime*. The field *abData[0]* contains one of the following values:<br>**Constant** — **Meaning**<br>LIN_ERROR_BIT — Bit error<br>LIN_ERROR_CHKSUM — Check sum error<br>LIN_ERROR_PARITY — Parity error of identifier<br>LIN_ERROR_SLNORE — Slave doesn't answer.<br>LIN_ERROR_SYNC — Invalid synchronization field.<br>LIN_ERROR_NOBUS — No bus activity.<br>LIN_ERROR_OTHER — Another, not specified error<br>The field *abData[1]* of the message contains the low byte of the current status (see LINLINESTATUS.dwStatus). The content of the other data fields is unefined. |
| | LIN_MSGTYPE_STATUS | Status message. This message type is assigned in the receiving buffers of all active message channels if the status of the controller is changed. The field *bPid* of the message contains the value 0xFF. The time of the event is noted in field *dwTime*. The field *abData[0]* contains the low byte of the current status. The content of the other data fields is undefined. (See LINLINESTATUS.dwStatus) |
| | LIN_MSGTYPE_WAKEUP | Exclusively for transmitting messages. Messages of this type generate a *Wake-Up* signal on the bus. The fields *dwTime*, *bPid* and *bDlen* have no significance. |
| | LIN_MSGTYPE_TMOVR | Counter overflow. Messages of this type are generated by LIN messages if an overflow of the 32 bit time stamp takes place. In field *dwTime* of the message the time of the event (standard 0) and in field *bDlen* the number of timer overflows. The content of the data fields *abData* is undefined, the field *bPid* has the value 0xFF. |
| | LIN_MSGTYPE_SLEEP | *Go-to-Sleep* message. The fields *dwTime*, *bPid* and *bDlen* have no significance.<br>For transmitting messages exclusively LIN_MSGTYPE_DATA, LIN_MSGTYPE_SLEEP and LIN_MSGTYPE_WAKEUP are defined, other values are not allowed. |
| Bits.dlen | [in/out] Number of valid data bytes in field *abData* of the message. | |
| *Bits.ecs* | [in/out] Enhanced check sum. Bit is set to 1, if it is a message with extended check sum according to LIN 2.0. | |
| *Bits.sor* | [out] Sender of response. Bit is set in messages that are transmitted by the LIN controller, i. e. in messages for which the controller has an entry in the response table. | |
| *Bits.ovr* | [out] Data overrun. Bit is set to 1 if the receiving FIFO is crowded after this message is assigned. | |
| *Bits.ido* | [in] ID only. Bit is exclusively in messages of type LIN_MSGTYPE_DATA relevant that are directly transmitted. If the bit in transmitting messages is set to 1 exclusively the ID without data is transmitted and serves in the master operating mode to send the IDs. Regarding all other message types this bit has no significance. | |
| *Bits.res* | Reserved for further extensions. This field is 0. | |

### 7.4.6 LINMSG

The data type describes the structure of LIN message telegrams.

```
typedef struct _LINMSG
{
  UINT32 dwTime;
  LINMSGINFO uMsgInfo;
  UINT8 abData[8];
} LINMSG, *PLINMSG;
```

| | |
|---|---|
| `dwTime` | In receiving messages this field contains the relative receiving point of the message in timer ticks. The resolution of timer tick can be calculated with the fields *dwClock-Freq* and *dwTscDivisor* of structure `LINCAPABILITIES` with the following formula: Resolution[s] = *dwTscDivisor* / *dwClockFreq* |
| *uMsgInfo* | Bit field with information about the message. For detailed description of bit field see `LINMSGINFO`. |
| *abData* | Array for up to 8 data bytes. Number of valid data bytes is determined by the field *uMsgInfo.Bits.dlen*. |

This page intentionally left blank

**HMS Industrial Networks AB**
Box 4126
300 04 Halmstad, Sweden

info@hms.se


4.02.0250.20022 1.1.3109 / 2017-01-27 10:15