VCI: C-API

Software Version 3/4

SOFTWARE DESIGN GUIDE

Important User Information

Disclaimer

The information in this document is for informational purposes only. Please inform HMS Networks of any inaccuracies or omissions found in this document. HMS Networks disclaims any responsibility or liability for any errors that may appear in this document.

HMS Networks reserves the right to modify its products in line with its policy of continuous product development. The information in this document shall therefore not be construed as a commitment on the part of HMS Networks and is subject to change without notice. HMS Networks makes no commitment to update or keep current the information in this document.

The data, examples and illustrations found in this document are included for illustrative purposes and are only intended to help improve understanding of the functionality and handling of the product. In view of the wide range of possible applications of the product, and because of the many variables and requirements associated with any particular implementation, HMS Networks cannot assume responsibility or liability for actual use based on the data, examples or illustrations included in this document nor for any damages incurred during installation of the product. Those responsible for the use of the product must acquire sufficient knowledge in order to ensure that the product is used correctly in their specific application and that the application meets all performance and safety requirements including any applicable laws, regulations, codes and standards. Further, HMS Networks will under no circumstances assume liability or responsibility for any problems that may arise as a result from the use of undocumented features or functional side effects found outside the documented scope of the product. The effects caused by any direct or indirect use of such aspects of the product are undefined and may include e.g. compatibility issues and stability issues.

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   Related Documents

Document	Author
VCI: C++ Software Version 4 Software Design Guide	HMS
VCI Driver Installation Guide	HMS

1.2   Document History

Version	Date	Description
2.0	January 2018	Reworked and edited in new design
2.1	September 2018	Minor changes, added information about reception time of a CAN message
2.2	April 2019	Layout changes
2.3	September 2021	Information about error frame time stamps
2.4	October 2021	Minor corrections

1.3   Conventions

Instructions and results are structured as follows:

▶   instruction 1

▶   instruction 2

→   result 1

→   result 2

Lists are structured as follows:

•   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. 6*

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.

1.4 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. In this guide the C programming interface VCINPL.DLL is described. The programming interface connects the VCI server and the application programs using predefined components, interfaces and functions.
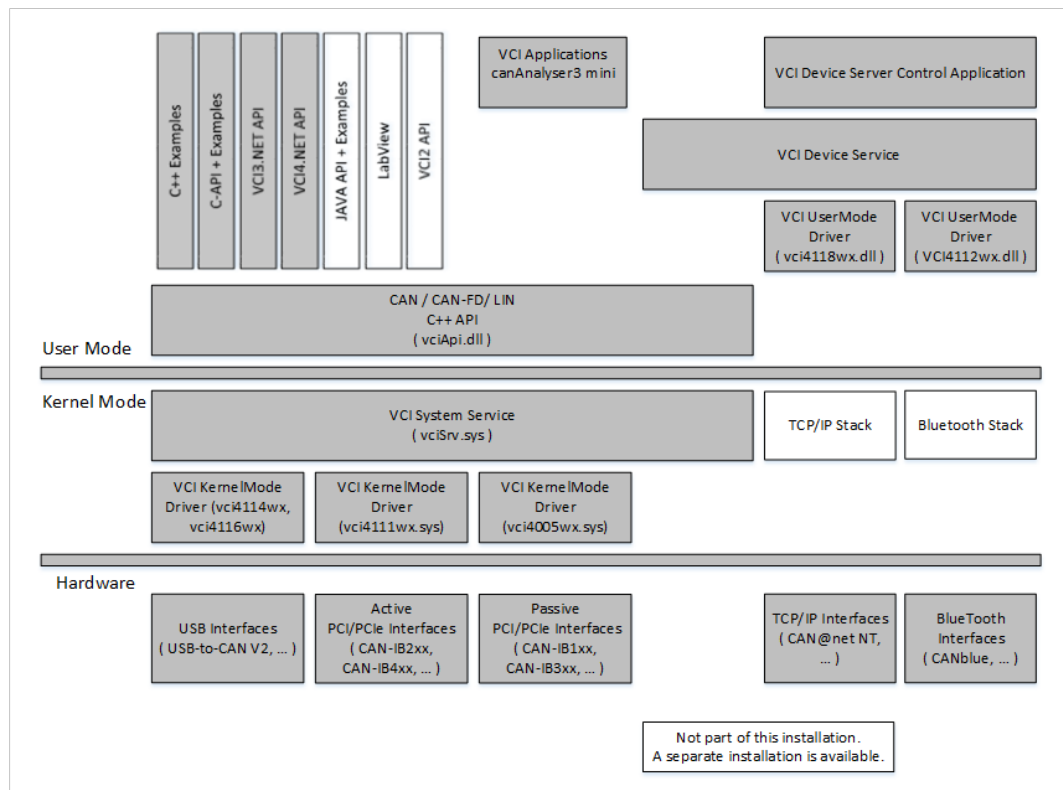

Fig. 1   **System structure and components**

2.1 Subcomponents and Functions of the Programming Interface

Native VCI Programming Interfaces (VCINPL.DLL)			
Device management and device access	**CAN control**	**CAN message channels**	**Cyclic CAN transmit list**
vciEnumDeviceOpen	canControlOpen	canChannelOpen	canSchedulerOpen
vciEnumDeviceClose	canControlClose	canChannelClose	canSchedulerClose
vciEnumDeviceNext	canControlGetCaps	canChannelGetCaps	canSchedulerGetCaps
vciEnumDeviceReset	canControlGetStatus	canChannelGetStatus	canSchedulerGetStatus
vciEnumDeviceWaitEvent	canControlDetectBitrate	canChannelInitialize	canSchedulerActivate
vciFindDeviceByHwid	canControlInitialize	canChannelActivate	canSchedulerReset
vciFindDeviceByClass	canControlReset	canChannelPeekMessage	canSchedulerAddMessage
vciSelectDeviceDlg	canControlStart	canChannelPostMessage	canSchedulerRemMessage
vciDeviceOpen	canControlSetAccFilter	canChannelWaitRxEvent	canSchedulerStartMessage
vciDeviceOpenDlg	canControlAddFilterIds	canChannelWaitTxEvent	canSchedulerStopMessage
vciDeviceClose	canControlRemFilterIds	canChannelReadMessage	
vciDeviceGetInfo		canChannelSendMessage	
vciDeviceGetCaps			
	LIN control	**LIN message monitors**	
	linControlOpen	linMonitorOpen	
	linControlClose	linMonitorClose	
	linControlGetCaps	linMonitorGetCaps	
	linControlGetStatus	linMonitorInitialize	
	linControlInitialize	linMonitorActivate	
	linControlReset	linMonitorPeekMessage	
	linControlStart	linMonitorWaitRxEvent	
	linControlWriteMessage	linMonitorReadMessage	

2.2 Programming Examples

With installing the VCI driver, programming examples are automatically installed in *c:\Users \Public\Documents\HMS\Ixxat VCI 4.0\Samples\Npl*.

3 Device Management and Device Access

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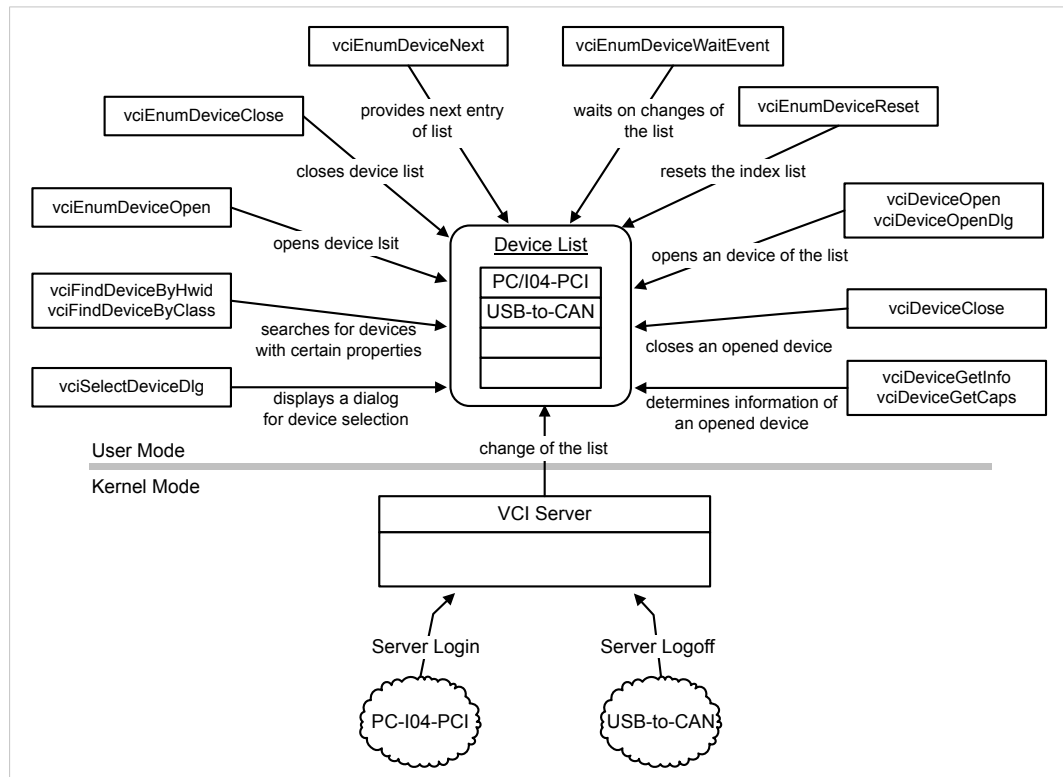

Fig. 2   Device management components

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, for example, because the connection is interrupted, the device is automatically removed from the device list.

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.

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 than PC-I04/PCI.
UniqueHardwareId	Hardware ID	Each device has a unique hardware ID. The ID can be used to differentiate between two 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 the global device list, call the function *vciEnumDeviceOpen*.

  →   Returns handle to the global device list.

With the handle 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

The application must provide the required memory as a structure of type *VCIDEVICEINFO*.

▶   Call the function *vciEnumDeviceNext*.

  →   Returns information about 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 the function *vciEnumDeviceNext* again.

  →   With each call information about the next device in the list is shown.

  →   When the list is run through, value VCI_E_NO_MORE_ITEMS is returned.

Reset Internal List Index

▶   Call the function *vciEnumDeviceReset*.

  →   Internal index of device list is reset.

  →   Subsequent call of function *vciEnumDeviceNext* provides information about the first device in the device list again.

Monitoring Changes in the Device List

▶   Call the function *vciEnumDeviceWaitEvent* and specify the handle of the device list in parameter *hEnum*.

  →   If the content of the device list changes, the function returns the value VCI_OK.

  →   Other return values indicate an error or signal that the waiting time specified for a function call is exceeded.

Closing Device List

To save system resources, it is recommended to close the device list if no further access is necessary.

▶   Call the function *vciEnumDeviceClose* specify the handle of the device list to be closed in parameter *hEnum*.

  →   Opened device list is closed.

  →   Specified handle is released.

3.2 Searching Individual Devices

Individual devices can be searched via the hardware ID, device class or a predefined dialog. For example, via the device class (*vciFindDeviceByClass*) an application can search for the first PC-104/PCI in the system.

▶ To search a device with a certain hardware ID, call the function *vciFindDeviceByHwid*.

▶ To search a device by device class (GUID), call the function *vciFindDeviceByClass*.

▶ Specify the device class (GUID) in parameter *rClass* and the instance number of the searched CAN interface in parameter *dwInst*.

▶ To display a predefined dialog that shows the device list, call the function *vciSelectDeviceDlg* and select the desired device.

→ If run successful, all functions return the device ID (VCIID) of the selected device.

ⓘ *The dialog via vciSelectDeviceDlg can also be used to find the hardware ID or the device class of a device.*

3.3 Accessing Devices

Accessing Individual Devices

► Call *vciDeviceOpen* and specify the device ID (VCIID) of the device to be opened in parameter *rVciid* (to determine the device ID see *Listing Available Devices, p. 11* and *Searching Individual Devices, p. 12*).

→ Returns handle to opened interface in parameter *phDevice*.

Accessing via Dialog

► To display a predefined dialog that shows the current device list, call the function *vciDeviceOpenDlg* and select the desired device.

→ Returns handle to opened interface.

Requesting Information About an Open Device

The application must provide the required memory as a structure of type *VCIDEVICEINFO*.

► Call the function *vciDeviceGetInfo*.

→ Returns information about the device in device list (see *Main Information about a Device, p. 10*).

Requesting Information About Technical Features of a Device

► Call the function *vciDeviceGetCaps*.

The function requires the handle of the device and the address of a structure of type *VCIDEVICECAPS*.

→ Returns required information in structure *VCIDEVICECAPS*.

→ Returned information informs how many bus controllers are available on a device.

→ Structure *VCIDEVICECAPS* contains a table with up to 32 entries, that describe the individual bus connection resp. controller. Entry 0 describes the bus connection 1, entry 1 bus connection 2 etc.
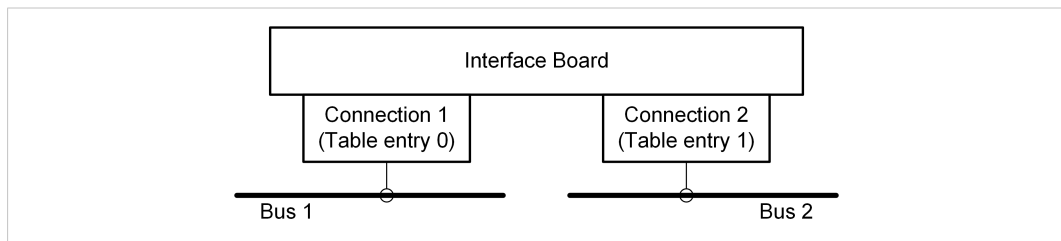

Fig. 3 **Interface with two bus connections**

Closing Devices

To save system resources, it is recommended to close the devices if no further access is necessary.

► Call the function *vciEnumDeviceClose*.

→ Opened device is closed.

→ Handle is released.

4 Accessing the Bus

4.1 Accessing the CAN Bus

Fig. 4 Components CAN controller and interface IDs

Each CAN connection can consist of up to three components:

- control unit (see *Control Unit, p. 20*)

- one or more message channels (see *Message Channels, p. 15*)

- cyclic transmitting list, optionally, only with devices with their own microprocessor (see *Cyclic Transmitting List, p. 24*)

The different functions to access the different components (*canControlOpen*, *canChannelOpen*, *canSchedulerOpen*) expect in the first parameter the handle of the CAN interface. To save system resources the handle of the CAN interface can be released after opening a component. For further access to the connection only the handle of component is required.

The functions *canControlOpen*, *canChannelOpen* and *canSchedulerOpen* can be called so that the user is presented with a dialog window to select the CAN interface and the CAN connection. It is accessed by entering the value 0xFFFFFFFF for the connection number. In this case, instead of the handle of the CAN interface , the functions expect in the first parameter the handle of the higher order window (parent), or the value ZERO if no higher order window is available.

4.1.1 Message Channels

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

Fig. 5   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. 6   CAN message distributor: possible configuration with three channels

Opening a Message Channel

Create or open a message channel with the function *canChannelOpen*.

▶ In parameter *hDevice* specify the handle of the CAN interface.

▶ In parameter *dwCanNo* specify the number of the CAN connection to be opened (0 for connection 1, 1 for connection 2 etc.).

▶ If the controller is used exclusively (exclusively with the first message channel, no further message channel can be opened) enter in parameter *fExclusive* value TRUE.

   or

   If the controller is used non-exclusively (further message channels can be created and opened) enter in parameter *fExclusive* value FALSE.

   →   If run successful, function returns a handle to the opened component.

Initializing the Message Channel

A newly generated message channel must be initialized before use.

Initialize with the function *canChannelInitialize*.

▶ In parameter *hCanChn* specify the handle of the opened message channel.

▶ Specify the size of the receive buffer in number of CAN messages in parameter *wRxFifoSize*.
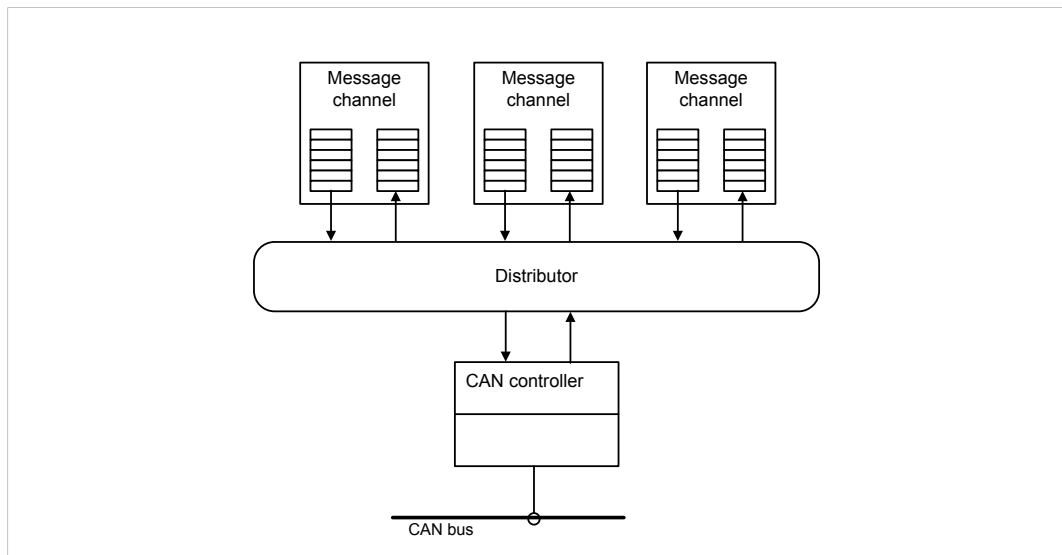
▶ Make sure that the value in parameter *wRxFifoSize* is higher than 0.

▶ Specify the number of messages the receive buffer must contain to trigger the receive event of a channel in *wRxThreshold*.

▶ Specify the size of the transmit buffer in number of CAN messages in parameter *wTxFifoSize*.

▶ Specify the number of messages the transmit buffer must have space for to trigger the transmit event in *wTxThreshold*.

▶ Call the function.

> (i) *The memory reserved for the receive and the transmit buffer comes from a limited system memory pool. The individual buffers of a messages channel can maximally contain up to approx. 2000 messages.*

Activating the Message Channel

A new message channel is inactive. Messages are only received and transmitted if the channel is active.

▶ Activate and deactivate the channel with function *canChannelActivate*.

▶ To activate the channel enter in parameter *fEnable* the value TRUE.

▶ To deactivate the channel enter in parameter *fEnable* the value FALSE.

Closing the Message Channel

Always close the message channel if it is no longer needed.

▶ To close a message channel, call the function *canChannelClose*.

Receiving CAN Messages

> ❗ Note that, when using interfaces with FPGA, error frames get the same time stamp as the last received CAN message.

There are different ways of reading received messages from the receive buffer.

▶ To read a received message call the function *canChannelReadMessage*.

→ If no messages are available in the receive buffer and no waiting time is defined the function waits until a new message is received.

▶ To define a maximum waiting time for the reading function, specify parameter *dwTimeout*.

→ If no messages are available the function waits only until the waiting time is expired.

▶ To get an immediate answer, call the function *canChannelPeekMessage*.

→ Next message in receive buffer is read.

→ If no message is available in the receive buffer, the function returns an error code.

▶ To wait for a new receive message or the next receive event, call the function *canChannelWaitRxEvent*.

The receive event is triggered when the receive buffer contains at least the number of messages specified in *wRxThreshold* when calling *canChannelInitialize* (see *Initializing the Message Channel, p. 16*).

Possible Use of `canChannelWaitRxEvent` and `canChannelPeekMessage`:

```
DWORD WINAPI ReceiveThreadProc( LPVOID lpParameter )
{
 HANDLE hChannel = (HANDLE) lpParameter;
 CANMSG sCanMsg;

 while (canChannelWaitRxEvent(hChannel, INFINITE) == VCI_OK)
 {
   while (canChannelPeekMessage(hChannel, &sCanMsg) == VCI_OK)
   {
       // processing of the message
   }
 }
  return 0;
}
```

Aborting the Thread Procedure

The thread procedure ends when the function *canChannelWaitRxEvent* returns an error code. When correctly called, all message channel specific functions only return an error code when a serious problem occurs. To abort the thread procedure the handle of the message channel must be closed from another thread, where all currently outstanding functions calls and new calls end with an error code. The disadvantage is that any transmit threads running simultaneously are also aborted.

Reception Time of a Message

The reception time of a message is noted in the field *dwTime* of structure *CANMSG*. The field contains the number of timer ticks that elapsed since the start of the timer. Dependent on the hardware the timer either starts with the start of the controller or with the start of the hardware. The time stamp of the CAN_INFO_START message (see type CAN_MSGTYPE_INFO of structure *CANMSGINFO*) , that is written to the receiving FIFOs of all active message channels when the control unit is started, contains the starting point of the controller.

To get the relative reception time of a message (in relation to the start of the controller) subtract the starting point of the controller (see *CANMSGINFO*) from the absolute reception time of the message (see *CANMSG*).

After an overrun of the counter the timer is reset.

Calculation of the relative reception time (T_{rx}) in ticks:

- T_{rx} = *dwTime* of message – *dwTime* of CAN_INFO_START (start of controller)

  Field dwTime of the message see *CANMSG*

  Field dwTime of CAN_INFO_START see CAN_MSGTYPE_INFO of structure *CANMSGINFO*

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*

Calculation of the reception time (T_{rx}) in seconds:

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

Transmitting CAN Messages

> **!** Note that, when using interfaces with FPGA, error frames get the same time stamp as the last received CAN message.

There are different ways of transmitting messages to the bus.

▶ To transmit a message, call the function *canChannelSendMessage*.

  → The function waits until a message channel is ready to receive a message and writes the message in the transmit buffer of the message channel.

▶ To define a maximum waiting time for sufficient space, specify parameter *dwTimeout*.

  → If no space is available before waiting time expires, the message is not written to the transmit buffer and the function returns VCI_E_TIMEOUT.

▶ To write the message immediately, call the function *canChannelPostMessage*.

  → If no space is available in the transmit buffer, the function returns an error code.

▶ To wait for the next transmit event, call the function *canChannelWaitTxEvent*.

  The transmit event is triggered when the transmit buffer has sufficient space for at least the number of messages specified in *wTxThreshold* when calling *canChannelInitialize* (see *Initializing the Message Channel, p. 16*).

Possible Use of `canChannelWaitTxEvent` and `canChannelPostMessage`:

```
HRESULT hResult;
HANDLE hChannel;
```

```
CANMSG sCanMsg;
.
.
.
hResult = canChannelPostMessage(hChannel, &sCanMsg);
if (hResult == VCI_E_TXQUEUE_FULL)
{
  canChannelWaitTxEvent(hChannel, INFINITE);
  hResult = canChannelPostMessage(hChannel, &sCanMsg);
}
.
.
```

Transmitting Messages Delayed

Connections with set bit `CAN_FEATURE_DELAYEDTX` in field *dwFeatures* of the structure *CANCAPABILITIES* 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 too much data in too short a time, which can cause data loss in slow nodes.

▶ In field *dwTime* of the structure *CANMSG* specify the number of ticks that have to pass at a minimum before the next message is forwarded to 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 the structure *CANCAPABILITIES*, the value in *dwTime* must not exceed the value in *dwMaxDtxTicks*.

Calculation of the resolution of a tick in seconds (s)

• Resolution [s] = dwDtxDivisor / dwClockFreq

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 another.

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

4.1.2   Control Unit

The control unit provides the following functions:

• configuration of the CAN controller

• configuration of the transmitting features of the CAN controller

• configuration of CAN message filters

• requesting of current operating state

The control unit can be opened by several application simultaneously to determine the status and the features of the CAN controller.

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

Opening and Closing the Control Unit

▶ Open the control unit with the function *canControlOpen*.

▶ In parameter *hDevice* specify the handle of the CAN interface.

▶ In parameter *dwCanNo* specify the number of the CAN connection to be opened (0 for connection 1, 1 for connection 2 etc.).

  → The application that calls first gets the exclusive control over the CAN controller.

  → If run successful, the function returns a handle to the opened component.

▶ With *canControlClose* close the control unit and release for access by other applications.

ⓘ *Before another application can get the exclusive control, all applications have to close the parallel opened control unit with* `canControlClose`.

Controller States

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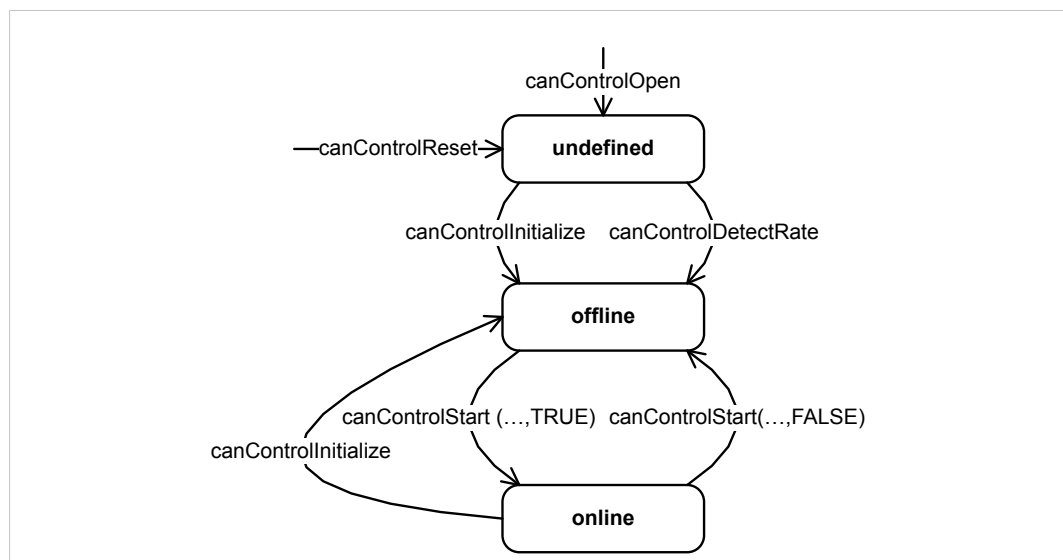

Fig. 7   Controller states

Initializing the Controller

After the first opening of the control unit the controller is in a non-initialized state.

▶ To leave the non-initialized state, call the function *canControlInitialize* or *canControlDetectBitrate*.

→ Controller is in state *offline*.

→ If the function *canControlInitialize* returns an access denied error code, the CAN controller is already used by another application.

▶ With *canControlInitialize* specify the operating mode in parameter *bMode*.

▶ With *canControlInitialize* set the values for bit timing register in parameters *bBtr0* and *bBtr1* (values corresponding 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 CiA or CANopen compliant bit timing register BTR0 and BTR1:			
Bit rate (KBit)	**Predefined constants for BTR0, BTR1**	**BTR0**	**BTR1**
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.

▶ To detect the bitrate of a running system, call the function *canControlDetectBitrate*.

→ Bus timing values are determined by the function and can be applied into the function *canControlInitialize*.

Starting the Controller

▶ Make sure, that the controller is initialized.

▶ To start the controller call function *canControlStart* with the value `TRUE` in parameter *fStart*.

→ Controller is in state *online*.

→ Controller is actively connected to the bus.

→ Incoming messages are forwarded to all active message channels.

→ Transmitting messages are transferred to the bus.

Stopping (resp. Reset) the Controller

▶ Call the function *canControlStart* with the value FALSE in parameter *fStart*.

→ Controller is in state *offline*.

→ Data transfer is stopped.

→ Controller is disabled.

or

▶ Call the function *canControlReset*.

→ Controller is in state *not initialized*.

→ Controller hardware and set message filters are reset to the predefined initial state.

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

4.1.3 Message Filter

All control units have a two-level message filter to filter the data messages received from the bus. The data messages are exclusively filtered by the ID. Data bytes are not considered.

If the self reception request bit on a transmit message is set, the message is entered in the receive buffer as soon as it is transmitted on the bus. In this case the message filter is bypassed.

Fig. 8   Filtering mechanism

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.

If the first filter level does not accept the ID it is forwarded to the second filter level. The second filter level 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.

Setting the Filter

The CAN controller has 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 are filtered by the 29 bit filter.

When the controller is reset or initialized the filters are set to let every message pass.

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

▶ Make sure that the control unit is in state *offline*.

▶ To set the filter, call the function `canControlSetAccFilter`.

▶ Add individual IDs or groups of IDs to the filter list with the function `canControlAddFilterIds` and remove with `canControlRemFilterIds`.

▶ In parameter *fExtend* set FALSE for 11 bit filter or TRUE for 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* correspond to the bits 0 to 10 resp. 0 to 28 of the ID of a CAN message. Bit 0 always corresponds to the Remote Transmission Request bit (RTR) of the 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 (with RTR bit not set) 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 do not 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. Each list can contain up to 2048 resp. 4096 entries.

▶ Register individual or groups of IDs with function *canControlAddFilterIds*.

▶ If necessary remove from list with the function *canControlRemFilterIds*.

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

If *canControlAddFilterIds* is called with the same values as in the example above 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 the acceptance filters completely, when calling the function *canControlSetAccFilter* 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

▶ To open the acceptance filter completely, when calling *canControlSetAccFilter* enter the values CAN_ACC_CODE_ALL and CAN_ACC_MASK_ALL.

  → Acceptance filter accepts all IDs and ID list is ineffective.

4.1.4 Cyclic Transmitting List

With the optionally provided transmitting list of the controller up to 16 messages can be transmitted cyclically. It is possible that after each transmit process a certain part of a CAN message is automatically incremented. The access to this list is limited to one application and therefore can not be used by several programs simultaneously.

Open the interface with the function *canSchedulerOpen*.

▶ In parameter *hDevice* specify the handle of the CAN interface.

▶ In parameter *dwCanNo* specify the number of the CAN connection to be opened (0 for connection 1, 1 for connection 2 etc.).

  → The application that calls first gets the exclusive control over the CAN controller.

  → If run successful, the function returns a handle to the opened component.

  → 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 close an opened transmit list and release it for access by other applications, call *canSchedulerClose*.

▶ To add a message object to the list, call the function *canSchedulerAddMessage*. The function expects a pointer to a structure of type *CANCYCLICTXMSG* that specifies the transmit object that is to be added to the list.

  → If run successfully, the function returns list index of the added transmit object.

▶ Specify the cycle time of a message in number of ticks in field *wCycleTime* of the structure *CANCYCLICTXMSG*.

▶ Make sure that the specified value is higher than 0 but less than or equal the value in field *dwCmsMaxTicks* of the structure *CANCAPABILITIES*.

- ► Calculate the length of a tick resp. the cycle time of the transmitting list (t_z) with values in fields *dwClockFreq* and *dwCmsDivisor* with the following formula:

$$t_z \text{ [s]} = (dwCmsDivisor \,/\, dwClockFreq)$$

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 resp. the cycle time (t_z).
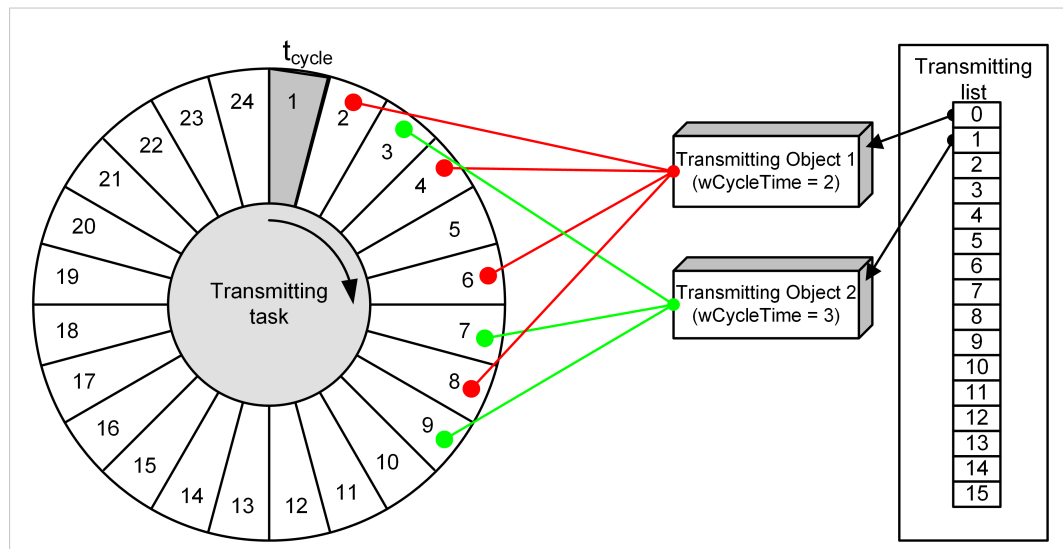

Fig. 9   **Transmitting task of the cyclic transmitting list with 24 time frames**

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 cycle time of 1 all time frames are occupied and no other objects can be created. The more transmitting objects are created, the larger their cycle 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 cycle time of the second object therefore is not met exactly and in this case leads to an 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` 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 to 0 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 the structure *CANCYCLICTXMSG* determines the starting position of the data value.

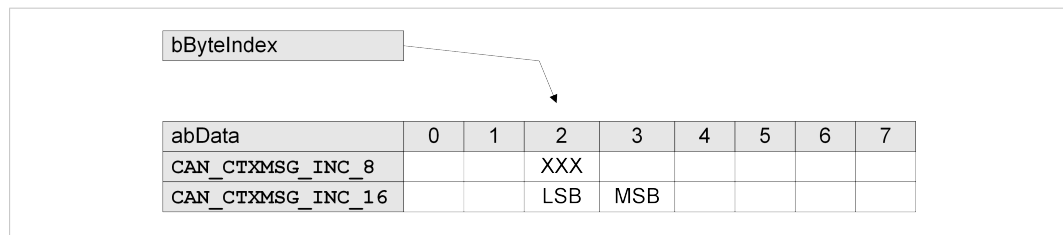| abData	0	1	2	3	4	5	6	7
CAN_CTXMSG_INC_8			XXX					
CAN_CTXMSG_INC_16			LSB	MSB				

bByteIndex

Fig. 10   Auto increment of data fields

Regarding 16 bit values, the low byte (LSB) is located in the field *abData[bByteIndex]* and the high byte (MSB) in the 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 the transmitting object from the list with the function *canSchedulerRemMessage*. The function expects the list index of the object to remove returned by *canSchedulerAddMessage*.

▸ To transmit the newly created transmitting object, call the function *canSchedulerStartMessage*.

▸ If necessary, stop transmitting with the function *canSchedulerStopMessage*.

▸ To get the status of transmitting task and of all created transmitting objects, call the function *canSchedulerGetStatus*. The required memory is provided as structure of type *CANSCHEDULERSTATUS* by the application.

→ If run successfully, 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 *canSchedulerAddMessage* 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, first start all transmitting objects with the function *canSchedulerStartMessage*.

▸ Activate the transmit task of the transmitting list with the function *canSchedulerActivate*.

▸ To stop all transmit objects simultaneously, disable the transmit task.

▸ To reset a transmitting task call the function *canSchedulerReset*.

→ Transmitting task is stopped.

→ All registered transmitting objects are removed from the specified cyclic transmitting list.
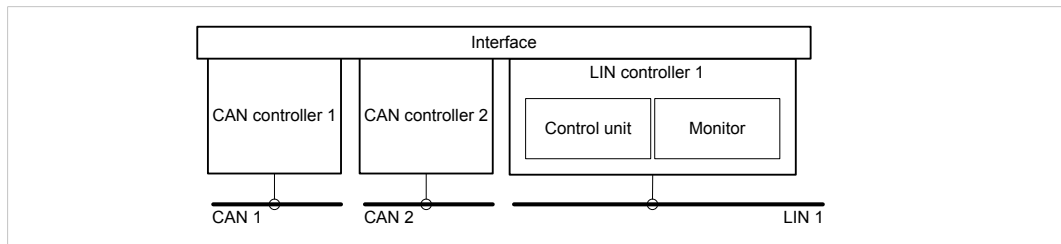
4.2   Accessing the LIN Bus

Fig. 11   Components LIN controller

Each LIN connections consists of the following components:

• control unit (see *Control Unit, p. 30*)

• one or more message monitors (see *Message Monitors, p. 27*)

The different functions to access the different components (`linControlOpen`, `linMonitorOpen`) expect in the first parameter the handle of the interface. To save system resources the handle of the interface can be released after opening a component. For further access to the connection only the handle of component is required.

The functions `linControlOpen` and `linMonitorOpen` can be called so that the user is presented with a dialog window to select the interface and the LIN connection. It is accessed by entering the value 0xFFFFFFFF for the connection number. In this case, instead of the handle of the interface, the functions expect in the first parameter the handle of the higher order window (parent), or the value ZERO if no higher order window is available.

4.2.1   Message Monitors

The basic functionality of a message monitor is the same, irrespective whether the connection is used exclusively or not. In case of exclusive use, the message channel is directly connected to the controller. If the LIN connection is not used exclusively, theoretically any number of message monitors can be created.

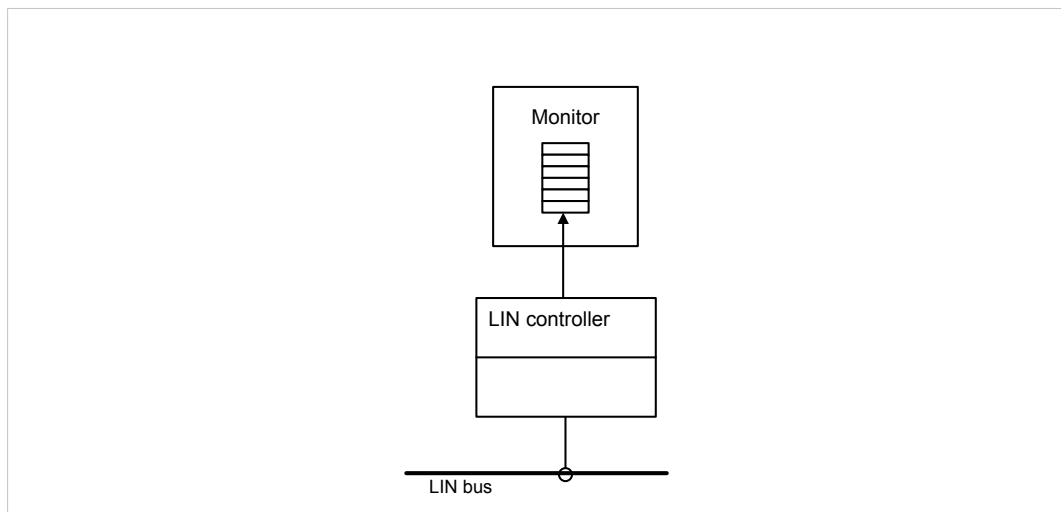**Fig. 12   Exclusive usage**

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

The distributor transfers all received messages to all active monitors and parallel the transmitted messages to the controller. No monitor is prioritized i. e. the algorithm used by the distributor is designed to treat all monitors as equal as possible.
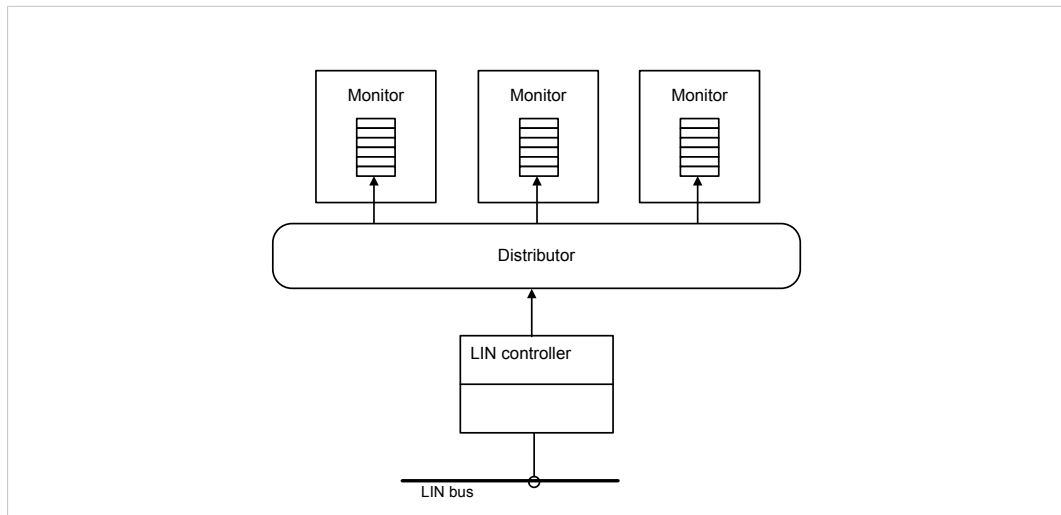
Fig. 13   Non-exclusive usage (with distributor)

Opening a Message Monitor

Create or open a message monitor with the function *linMonitorOpen*.

▶   In parameter *hDevice* specify the handle of the opened LIN monitor.

▶   In parameter *dwLinNo* specify the number of the LIN connection to be opened (0 for connection 1, 1 for connection 2 etc.).

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

or

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

→   Function returns a handle to the opened component.

Initializing the Message Monitor

A newly generated message monitor has to be initialized before use.

Initialize with the function *linMonitorInitialize*.

▶   In parameter *hLinMon* specify the handle of the opened LIN monitor.

▶   Specify the size of the receive buffer in number of messages in parameter *wFifoSize*.

▶   Make sure that the value in parameter *wFifoSize* is higher than 0.

▶   Specify the number of messages the receive buffer must contain to trigger the receive event of a monitor in *wThreshold*.

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*.

ⓘ   *The memory reserved for the receive and the transmit buffer comes from a limited system memory pool. The individual buffers of a messages channel can maximally contain up to approx. 2000 messages.*

Activating the Message Monitor

A new monitor is deactivated. Messages are only received and transmitted if the monitor is active and if the LIN controller is started. For further information about LIN controllers see chapter *Control Unit, p. 30*.

▸   Activate and deactivate the message monitor with the function *linMonitorActivate*.

▸   To activate the monitor enter in parameter *fEnable* the value `TRUE`.

▸   To deactivate the monitor enter in parameter *fEnable* the value `FALSE`.

Closing the Message Monitor

Always close the message monitor if it is no longer needed.

▸   To close a message monitor call the function *linMonitorClose*.

Receiving LIN Messages

There are different ways of reading received messages from the receive buffer.

▸   To read a received message call the function *linMonitorReadMessage*.

→   If no messages are available in the receive buffer and no waiting time is defined the function waits until a new message is received.

▸   To define a maximum waiting time for the reading function, specify parameter *dwTimeout*.

→   If no messages are available the function waits only until the waiting time is expired.

▸   To get an immediate answer, call the function *linMonitorPeekMessage*.

→   Next message in receive buffer is read.

→   If no message is available in the receive buffer, the function returns an error code.

▸   To wait for a new receive message or the next receive event, call the function *linMonitorWaitRxEvent*.

The receive event is triggered when the receive buffer contains at least the number of messages specified in *wThreshold* when calling *linMonitorInitialize* (see *Initializing the Message Monitor, p. 28*).

Possible Use of `linMonitorWaitRxEvent` and `linMonitorPeekMessage`:

```
DWORD WINAPI ReceiveThreadProc( LPVOID lpParameter )
{
 HANDLE hLinMon = (HANDLE) lpParameter;
 LINMSG sLinMsg;

 while (linMonitorWaitRxEvent(hLinMon, INFINITE) == VCI_OK)
 {
   while (linMonitorPeekMessage(hLinMon, &sLinMsg) == VCI_OK)
   {
       // processing of the message
   }
 }
 return 0;
}
```

Aborting the Thread Procedure

The thread procedure ends when the function *linMonitorWaitRxEvent* returns an error code. When correctly called all message monitor specific functions only return an error code when a serious problem occurs. To abort the thread procedure the handle of the message monitor must be closed from another thread, where all currently outstanding functions calls and

new calls end with an error code. The disadvantage is that any transmit threads running simultaneously are also aborted.

4.2.2 Control Unit

The control unit provides the following functions:

- configuration of the LIN controller
- configuration of transmitting features of the LIN controller
- requesting of current controller state

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

Opening and Closing the Control Unit

- ► Open with the function *linControlOpen*.
- ► In parameter *hDevice* specify the handle of the LIN controller.
- ► In parameter *dwLinNo* specify the number of the connection to be opened (0 for connection 1, 1 for connection 2 etc.).
    - → If run successfully, function returns the handle of the interface.
    - → If the function returns an error code respective *access denied* the component is already used by another program.
- ► With *linControlClose* close the control unit and release for access by other applications. Only release the control unit when it is no longer required.
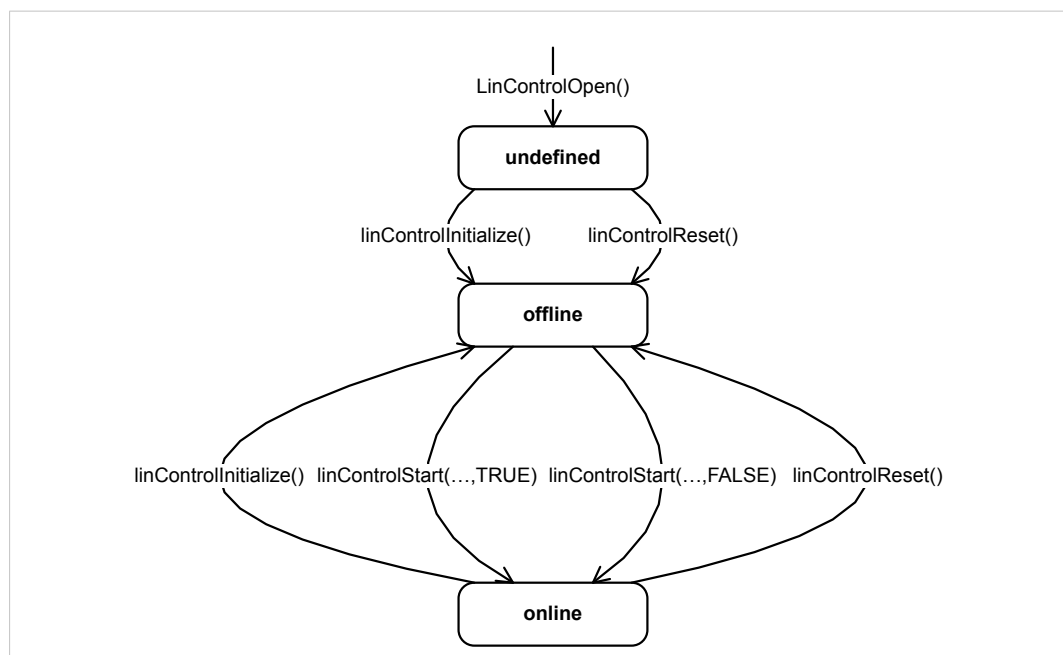

Fig. 14   LIN controller states

Initializing the Controller

After the first opening of the control unit the controller is in a non-initialized state.

- ► To leave the non-initzialized state, call the function *linControlInitialize*.
- ► In parameter *hLinCtl* specify the handle of the LIN controller.
    - → Controller is in state *offline*.

▶ With *linControlInitialize* specify the operating mode in parameter *bMode*.

▶ With *linControlInitialize* specify the bitrate in bits per second in parameter *wBitrate*.

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 bitrate detection, enter `LIN_BITRATE_AUTO` in the field *wBitrate* to activate the automatic bitrate detection.

Recommended Bitrates		
Slow (bit/sec)	Medium (bit/sec)	Fast (bit/sec)
2400	9600	19200

Starting and Stopping the Controller

▶ To start the LIN controller, call the function *linControlStart* with the value `TRUE` in parameter *fStart*.

→ LIN controller is in state *online*.

→ LIN controller is actively connected to bus.

→ Incoming messages are forwarded to all active message monitors.

▶ To stop the LIN controller, call the function *linControlStart* with the value `FALSE` in parameter *fStart*.

→ LIN controller is in state *offline*.

→ Message transfer to the monitor is interrupted and controller is deactivated.

▶ Call the function *linControlReset* to shift the controller to state *offline* and to reset the controller hardware.

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

Transmitting LIN Messages

Messages can be transmitted directly or can be registered in a response table in the controller.
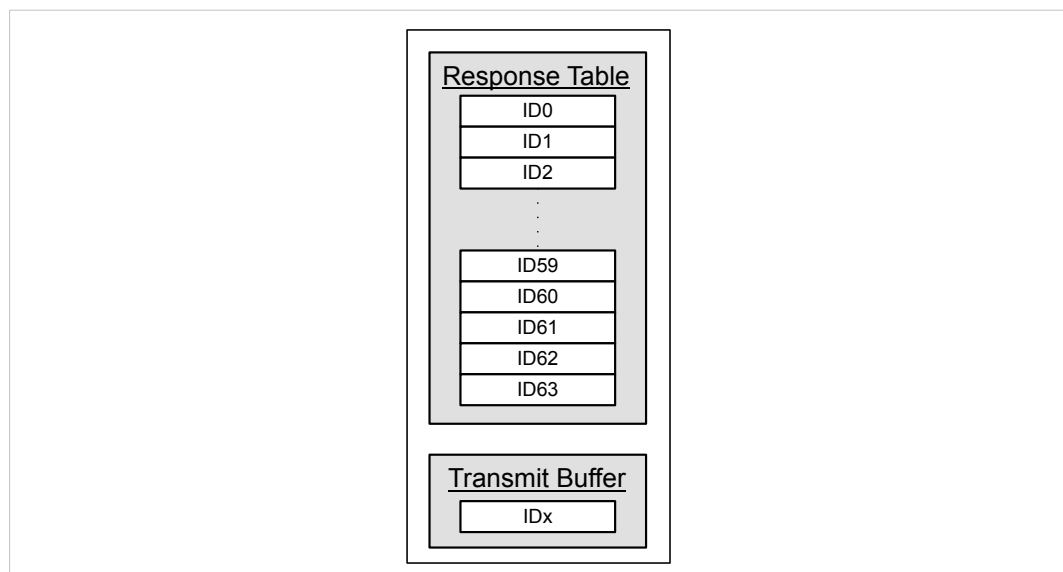

Fig. 15 Internal structure of a control unit

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.

Change and update the content of the response table with the function *linControlWriteMessage*:

- ► In parameter *hLinCtl* specify the handle of the opened LIN controller.

- ► In parameter *fSend* enter value FALSE.

  - → The message with the response data in the field *abData* of the structure *LINMSG* is transferred to the function in parameter *pLinMsg*.

- ► To clear the response table, call the function *linControlReset*.

The LIN message in field *abData* of the structure *LINMSG* has to be of type LIN_MSGTYPE_ DATA and has to contain an ID in the range 0 to 63. Irrespective of the operating mode (master or slave) the table has to be initialized before the controller is started. It can be updated at any time without stopping the controller.

Transmit messages directly to the bus with the function *linControlWriteMessage*:

- ► In parameter *hLinCtl* specify the handle of the opened LIN controller.

- ► In parameter *fSend* enter value 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 connection is operated as master, control messages LIN_MSGTYPE_SLEEP and LIN_ MSGTYPE_WAKEUP and data messages of the type LIN_MSGTYPE_DATA can be transmitted directly. If the connection 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 the type LIN_MSGTYPE_SLEEP generates a goto-Sleep frame, a message of the type LIN_MSGTYPE_WAKEUP a wake-up frame on the bus. For further information see chapter Network Management in LIN specifications.

In master mode the function *linControlWriteMessage* also serves for transmitting IDs. For this a message of the 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 *LINMSG*).

Irrespective of the value of the parameter *fSend linControlWriteMessage* 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.

5   Functions

5.1   General Functions

5.1.1   vciInitialize

Initializes the VCINPL for the calling process.

```
HRESULT EXTERN_C vciInitialize (
);
```

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

The function must be called at the beginning of a program in order to initialize the DLL for the calling process.

5.1.2   vciGetVersion

Gets the version number of the installed VCI.

```
HRESULT EXTERN_C vciGetVersion (
     PUINT32 pdwMajorVersion,
     PUINT32 pdwMinorVersion
);
```

Parameter

Parameter	Dir.	Description
pdwMajorVersion	[out]	Address of a variable of type UINT32. If run successfully, the function returns the major version number of the VCI in this variable.
pdwMinorVersion	[out]	Address of a variable of type UINT32. If run successfully, the function returns the minor version number of the VCI in this variable.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.1.3   vciGetVersionEx

Gets the version number of the installed VCI.

```
HRESULT EXTERN_C vciGetVersionEx (
      PUINT32 pdwMajorVersion,
      PUINT32 pdwMinorVersion,
      PUINT32 pdwRevNumber,
      PUINT32 pdwBuildNumber
);
```

Parameter

Parameter	Dir.	Description
pdwMajorVersion	[out]	Address of a variable of type UINT32. If run successfully, the function returns the major version number of the VCI in this variable.
pdwMinorVersion	[out]	Address of a variable of type UINT32. If run successfully, the function returns the minor version number of the VCI in this variable.
pdwRevNumber	[out]	Address of a variable of type UINT32. If run successfully, the function returns the revision number of the VCI in this variable.
pdwBuildNumber	[out]	Address of a variable of type UINT32. If run successfully, the function returns the build number of the VCI in this variable.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.1.4   vciFormatErrorA

Formats VCI error code as text.

```
HRESULT EXTERN_C vciFormatErrorA (
      HRESULT hrError,
      PCHAR   pszText,
      UINT32  dwLength
);
```

Parameter

Parameter	Dir.	Description
hrError	[in]	Error code that is to be converted into text.
pszText	[out]	Pointer to a buffer for the text string. The buffer must provide space for at least *dwLength* characters. The function saves the error text including a final 0 character in the specified memory area.
dwLength	[in]	Size of the buffer specified in *pszText* in number of characters.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.1.5 vciFormatErrorW

Formats VCI error code as text (wide character version).

```
HRESULT EXTERN_C vciFormatErrorW (
    HRESULT hrError,
    PWCHAR  pszText,
    UINT32  dwLength
);
```

Parameter

Parameter	Dir.	Description
hrError	[in]	Error code that is to be converted into text.
pszText	[out]	Pointer to a buffer for the text string. The buffer must provide space for at least *dwLength* characters. The function saves the error text including a final 0 character in the specified memory area.
dwLength	[in]	Size of the buffer specified in *pszText* in number of characters.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.1.6 vciDisplayErrorA

Displays VCI error code in message box.

```
void EXTERN_C vciDisplayErrorA (
    HWND    hwndParent,
    PCHAR   pszCaption,
    HRESULT hrError
);
```

Parameter

Parameter	Dir.	Description
hwndParent	[in]	Handle of the higher order window. If value ZERO is specified here, the message window has no higher order window.
pszCaption	[in]	Pointer to a 0-terminated character string with the text for the title line of the message window. If value ZERO is specified here, a pre-defined title line text is displayed.
hrError	[in]	Error code for which the message is to be displayed.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.1.7 vciDisplayErrorW

Displays VCI error code in message box (wide character version).

```
void EXTERN_C vciDisplayErrorW (
    HWND     hwndParent,
    PWCHAR   pszCaption,
    HRESULT  hrError
);
```

Parameter

Parameter	Dir.	Description
hwndParent	[in]	Handle of the higher order window. If value ZERO is specified here, the message window has no higher order window.
pszCaption	[in]	Pointer to a 0-terminated character string with the text for the title line of the message window. If value ZERO is specified here, a pre-defined title line text is displayed.
hrError	[in]	Error code for which the message is to be displayed.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.1.8 vciCreateLuid

Creates a locally unique VCI ID.

```
HRESULT EXTERN_C vciCreateLuid (
    PVCIID pVciid
);
```

Parameter

Parameter	Dir.	Description
pVciid	[out]	Pointer to buffer for the locally unique VCI ID.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.1.9 vciLuidToCharA

Converts a locally unique ID (VCIID) to a character string.

```
HRESULT EXTERN_C vciLuidToCharA (
     REFVCIID rVciid,
     PCHAR    pszLuid,
     LONG     cbSize
);
```

Parameter

Parameter	Dir.	Description
rVciid	[in]	Reference to the locally unique VCI ID to be converted into a character string.
pszLuid	[out]	Pointer to a buffer for the 0-terminated character string. If run successfully, the function saves the converted VCI ID in the memory area specified here. The buffer must provide space for at least 17 characters including the final 0-character.
cbSize	[in]	Size of the buffer specified in *pszLuid* in bytes.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pszLuid* points to an invalid buffer.
VCI_E_BUFFER_OVERFLOW	Buffer specified in *pszLuid* is not large enough for the character string.

5.1.10 vciLuidToCharW

Converts a locally unique ID (VCIID) to a wide character string.

```
HRESULT EXTERN_C vciLuidToCharW (
     REFVCIID rVciid,
     PWCHAR   pwszLuid,
     LONG     cbSize
);
```

Parameter

Parameter	Dir.	Description
rVciid	[in]	Reference to the locally unique VCI ID to be converted into a character string.
pwszLuid	[out]	Pointer to a buffer for the 0-terminated wide character string. If run successfully, the function saves the converted VCI ID in the memory area specified here. The buffer must provide space for at least 17 characters including the final 0-character.
cbSize	[in]	Size of the buffer specified in *pszLuid* in bytes.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pszLuid* points to an invalid buffer.
VCI_E_BUFFER_OVERFLOW	Buffer specified in *pszLuid* is not large enough for the character string.

5.1.11   vciCharToLuidA

Converts a 0-terminated character string to a locally unique VCI ID (VCIID).

```
HRESULT EXTERN_C vciCharToLuidA (
    PCHAR   pszLuid,
    PVCIID  pVciid
);
```

Parameter

Parameter	Dir.	Description
pszLuid	[in]	Pointer to the 0-terminated character string to be converted.
pVciid	[out]	Address of a variable of type VCIID. If run successfully, the function returns the converted ID in this variable.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pszLuid* or *pVciid* points to an invalid buffer.
VCI_E_FAIL	Character string specified in *pszLuid* can not be converted into a valid ID.

5.1.12   vciCharToLuidW

Converts a 0-terminated wide character string to a locally unique VCI ID (VCIID).

```
HRESULT EXTERN_C vciCharToLuidW (
    PWCHAR pwszLuid,
    PVCIID pVciid
);
```

Parameter

Parameter	Dir.	Description
pwszLuid	[in]	Pointer to the 0-terminated wide character string to be converted.
pVciid	[out]	Address of a variable of type VCIID. If run successfully, the function returns the converted ID in this variable.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pszLuid* or *pVciid* points to an invalid buffer.
VCI_E_FAIL	Character string specified in *pszLuid* can not be converted into a valid ID.

5.1.13   vciGuidToCharA

Converts a globally unique ID (GUID) into a character string.

```
HRESULT EXTERN_C vciGuidToCharA (
     REFGUID rGuid,
     PCHAR   pszGuid,
     LONG    cbSize
);
```

Parameter

Parameter	Dir.	Description
rGuid	[in]	Reference to the globally unique ID that is to be converted into a character string.
pszGuid	[out]	Pointer to the buffer for the 0-terminated character string. If run successfully, the function saves the converted GUID in the specified memory area. The buffer must have space for at least 39 characters including the final 0-character.
cbSize	[in]	Size of the in *pszGuid* specified buffer in Bytes.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pszGuid* points to an invalid buffer.
VCI_E_BUFFER_OVERFLOW	Buffer specified in *pszGuid* is not large enough for the character string.

5.1.14   vciGuidToCharW

Converts a globally unique ID (GUID) into a character string.

```
HRESULT EXTERN_C vciGuidToCharW (
     REFGUID rGuid,
     PWCHAR  pwszGuid,
     LONG    cbSize
);
```

Parameter

Parameter	Dir.	Description
rGuid	[in]	Reference to the globally unique ID that is to be converted into a character string.
pwszGuid	[out]	Pointer to the buffer for the 0-terminated character string. If run successfully, the function saves the converted GUID in the specified memory area. The buffer must have space for at least 39 characters including the final 0-character.
cbSize	[in]	Size of the in *pszGuid* specified buffer in Bytes.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pwszGuid* points to an invalid buffer.
VCI_E_BUFFER_OVERFLOW	Buffer specified in *pwszGuid* is not large enough for the character string.

5.1.15   vciCharToGuidA

Converts a 0-terminated character string into a globally unique ID (GUID).

```
HRESULT EXTERN_C vciCharToGuidA (
      PCHAR pszGuid,
      PGUID pGuid
);
```

Parameter

Parameter	Dir.	Description
pszGuid	[in]	Pointer to the 0-terminated character string to be converted.
pGuid	[out]	Address of a variable of type GUID. If run successfully, the function returns the converted ID in this variable.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pszGuid* or *pGuid* points to an invalid buffer.
VCI_E_FAIL	Character string specified in *pszGuid* can not be converted into a valid ID.

5.1.16   vciCharToGuidW

Converts a 0-terminated wide character string into a globally unique ID (GUID).

```
HRESULT EXTERN_C vciCharToGuidW (
      PWCHAR pwszGuid,
      PGUID  pGuid
);
```

Parameter

Parameter	Dir.	Description
pwszGuid	[in]	Pointer to the 0-terminated character string to be converted.
pGuid	[out]	Address of a variable of type GUID. If run successfully, the function returns the converted ID in this variable.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALIDARG	Parameter *pwszGuid* or *pGuid* points to an invalid buffer.
VCI_E_FAIL	Character string specified in *pszGuid* can not be converted into a valid ID.

5.2   Functions for the Device Management

5.2.1   Functions for Accessing the Device List

vciEnumDeviceOpen

Opens the list of all fieldbus adapters registered with the VCI.

```
HRESULT EXTERN_C vciEnumDeviceOpen (
     PHANDLE hEnum
);
```

Parameter

Parameter	Dir.	Description
hEnum	[out]	Address of a variable of type HANDLE. If run successfully, the function returns the handle of the opened device list in this variable. In the case of an error, the variable is set to ZERO.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

vciEnumDeviceClose

Closes the device list opened with the function vciEnumDeviceOpen.

```
HRESULT EXTERN_C vciEnumDeviceClose (
     HANDLE hEnum
);
```

Parameter

Parameter	Dir.	Description
hEnum	[in]	Handle of the device list to be closed

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

After the function is called, the handle that is specified in *hEnum* is no longer valid and must no longer be used.

vciEnumDeviceNext

Determines the description of a fieldbus adapter of the device list and increases the internal list index so that a subsequent call of the function supplies the description to the next adapter.

```
HRESULT EXTERN_C vciEnumDeviceNext (
    HANDLE        hEnum,
    PVCIDEVICEINFO pInfo
);
```

Parameter

Parameter	Dir.	Description
hEnum	[in]	Handle to the opened device list.
pInfo	[out]	Address of a data structure of type *VCIDEVICEINFO*. If run successfully, the function saves information on the adapter in the memory area specified here.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_NO_MORE_ITEMS`	ist does not contain any more entries.
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

vciEnumDeviceReset

Resets the internal list index of the device list, so that a subsequent call of vciEnumDeviceNext returns the first entry of the list again.

```
HRESULT EXTERN_C vciEnumDeviceReset (
    HANDLE hEnum
);
```

Parameter

Parameter	Dir.	Description
hEnum	[in]	Handle of the opened device list

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

vciEnumDeviceWaitEvent

Waits until the content of the device list changes, or a given waiting time has elapsed.

```
HRESULT EXTERN_C vciEnumDeviceWaitEvent (
    HANDLE hEnum,
    UINT32 dwTimeout
);
```

Parameter

Parameter	Dir.	Description
hEnum	[in]	Handle of the opened device list
dwTimeout	[in]	Specifies the timeout interval, in milliseconds. The function returns if the interval elapses, even if the device list has not changed. If *dwTimeout* is zero, the function tests the state of the device list and returns immediately. If *dwTimeout* is INFINITE (0xFFFFFFFF), the function's timeout interval never elapses.

Return Value

Return value	Description
`VCI_OK`	Contents of the device list changed since the last call of `vciEnumDeviceWaitEvent`
`VCI_E_TIMEOUT`	Contents of the device list have not changed and the timeout period specified in parameter *dwTimeout* elapsed.

Remark

The contents of the device list changes when an adapter is added or removed.

vciFindDeviceByHwid

Searches for an adapter with a certain hardware ID.

```
HRESULT EXTERN_C vciFindDeviceByHwid (
    REFGUID rHwid,
    PVCIID  pVciid
);
```

Parameter

Parameter	Dir.	Description
rHwid	[in]	Reference to the unique hardware ID of the adapter to search for
pVciid	[out]	Address of a variable type VCIID. If run successfully, the function returns the device ID of the found adapter in this variable.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The device ID returned by this function can be used to open the adapter with the function *vciDeviceOpen*. Each adapter has a unique hardware ID, which also remains valid after a restart of the system.

vciFindDeviceByClass

Searches for an adapter with a certain device class.

```
HRESULT EXTERN_C vciFindDeviceByClass (
    REFGUID rClass,
    UINT32  dwInst,
    PVCIID  pVciid
);
```

Parameter

Parameter	Dir.	Description
rClass	[in]	Reference to the device class of the adapter to search for
dwInst	[in]	Instance number of the adapter to search for. If more than one adapter of the same class is available, this value defines the number of the adapter to search for in the device list. Value 0 selects the first adapter of the specified device class.
pVciid	[out]	Address of a variable type VCIID. If run successfully, the function returns the device ID of the found adapter in this variable.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The device ID returned by this function can be used to open the adapter with the function *vciDeviceOpen*.

vciSelectDeviceDlg

Displays a dialog window to select an adapter from the current device list on the screen.

```
HRESULT EXTERN_C vciSelectDeviceDlg (
    HWND   hwndParent,
    PVCIID pVciid
);
```

Parameter

Parameter	Dir.	Description
hwndParent	[in]	Handle of the higher order window. If value ZERO is specified here, the dialog window has no higher order window.
pVciid	[out]	Address of a variable type VCIID. If run successfully, the function returns the device ID of the selected adapter in this variable.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_ABORT`	Dialog window closed without having selected a CAN interface
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The device ID returned by this function can be used to open the adapter with the function *vciDeviceOpen*.

5.2.2   Functions for Accessing VCI Devices

vciDeviceOpen

Opens the fieldbus adapter with the specified device ID.

```
HRESULT EXTERN_C vciDeviceOpen (
     REFVCIID rVciid,
     PHANDLE  phDevice
);
```

Parameter

Parameter	Dir.	Description
rVciid	[in]	Device ID of the adapter to be opened
phDevice	[out]	Address of a variable of type HANDLE. If run successfully, the function returns the handle of the opened adapter in this variable. In the event of an error, the variable is set to ZERO.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

vciDeviceOpenDlg

Displays a dialog window to select a fieldbus adapter on the screen and opens the adapter selected by the user.

```
HRESULT EXTERN_C vciDeviceOpenDlg (
     HWND    hwndParent,
     PHANDLE phDevice
);
```

Parameter

Parameter	Dir.	Description
hwndParent	[in]	Handle of the higher order window. If value ZERO is specified here, the dialog window has no higher order window.
phDevice	[out]	Address of a variable of type HANDLE. If run successfully, the function saves the handle of the selected and opened adapter in this variable. In the event of an error the variable is set to ZERO.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

vciDeviceClose

Closes an opened fieldbus adapter.

```
HRESULT EXTERN_C vciDeviceClose (
     HANDLE hDevice
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the adapter to be closed. The specified handle must come from a call of one of the functions *vciEnumDeviceOpen* or *vciDeviceOpenDlg*.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

After the function is called, the handle specified in *hDevice* is no longer valid and must no longer be used.

vciDeviceGetInfo

Determines general information on a fieldbus adapter.

```
HRESULT EXTERN_C vciDeviceGetInfo (
     HANDLE        hDevice,
     PVCIDEVICEINFO pInfo
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the opened adapter
pInfo	[out]	Address of a structure of type *VCIDEVICEINFO*. If run successfully, the function saves information on the adapter in the memory area specified here.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

vciDeviceGetCaps

Determines information on the technical equipment of a fieldbus adapter.

```
HRESULT EXTERN_C vciDeviceGetCaps (
    HANDLE        hDevice,
    PVCIDEVICECAPS pCaps
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the opened adapter
pCaps	[out]	Address of a structure of type *VCIDEVICECAPS*. If run successfully, the function saves the information on the technical equipment in the memory area specified here.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

5.3 Functions for CAN Access

5.3.1 Control Unit

canControlOpen

Opens the control unit of a CAN connection on a fieldbus adapter.

```
HRESULT EXTERN_C canControlOpen (
    HANDLE  hDevice,
    UINT32  dwCanNo,
    PHANDLE phCanCtl
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the fieldbus adapter
dwCanNo	[in]	Number of the CAN connection of the control unit to be opened. Value 0 selects the first connection, value 1 the second connection and so on.
phCanCtl	[out]	Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened CAN controller in this variable. In the event of an error, the variable is set to ZERO.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

If the value 0xFFFFFFFF is specified in the parameter *dwCanNo*, the function displays a dialog window to select an adapter and a CAN connection on the screen. In this case the function expects the handle of a higher order window, or the value ZERO if no higher order window is available, in the parameter *hDevice* instead of the handle of the adapter.

canControlClose

Closes an opened CAN controller.

```
HRESULT EXTERN_C canControlClose (
    HANDLE hCanCtl
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the CAN controller to be closed. The specified handle must come from a call of the function `canControlOpen`.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

After the function is called, the handle specified in *hCanCtl* is no longer valid and must no longer be used.

canControlGetCaps

Determines the features of a CAN connection.

```
HRESULT EXTERN_C canControlGetCaps (
    HANDLE          hCanCtl,
    PCANCAPABILITIES pCanCaps
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller.
pCanCaps	[out]	Pointer to a structure of type *CANCAPABILITIES*. If run successfully, the function saves the features of the CAN connection in the memory area specified here.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

canControlGetStatus

Determines the current settings and the current status of the controller of a CAN connection.

```
HRESULT EXTERN_C canControlGetStatus (
    HANDLE         hCanCtl,
    PCANLINESTATUS pStatus
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller.
pStatus	[out]	Pointer to a structure of type *CANLINESTATUS*. If run successfully, the function saves the current settings and the status of the controller in the memory area specified here.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

canControlDetectBitrate

Determines the current bit rate of the bus to which the CAN connection is connected.

```
HRESULT EXTERN_C canControlDetectBitrate (
     HANDLE hCanCtl,
     UINT16 wTimeout,
     UINT32 dwCount,
     PUINT8 pabBtr0,
     PUINT8 pabBtr1,
     PINT32 plIndex
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller
wTimeout	[in]	Maximum waiting time in milliseconds between two messages on the bus
dwCount	[in]	Number of elements in the bit timing tables *pabBtr0* or *pabBtr1*
pabBtr0	[in]	Pointer to a table with the values to be tested for the bus timing register 0. The value of an entry corresponds to the BT0 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz. The table must contain at least dwCount elements.
pabBtr1	[in]	Pointer to a table with the values to be tested for the bus timing register 1. The value of an entry corresponds to the BT1 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz. The table must contain at least dwCount elements.
plIndex	[out]	Pointer to a variable of type INT32. If run successfully, the function returns the table index of the found bit timing values in this variable.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_TIMEOUT`	Bit rate detection failed due to timeout, no message transmitted within the time specified in *wTimeout*

Remark

For more information on the bus timing values in the tables *pabBtr0* and *pabBtr1* see chapter *Initializing the Controller*. To detect the bit rate, the CAN controller is operated in "Listen only" mode. It is therefore necessary for two further bus nodes to transmit messages when the function is called. If no messages are transmitted within the time specified in *wTimeout*, the function returns the value `VCI_E_TIMEOUT`. If run successfully, the function receives the variables to which the parameter *plIndex* shows the index (including 0) of the found values in the bus timing tables. The corresponding table values can then be used to initialize the CAN controller with the function *canControlInitialize*. The function can be called in the undefined and stopped status.

canControlInitialize

Sets the operating mode and bit rate of a CAN connection.

```
HRESULT EXTERN_C canControlInitialize (
     HANDLE hCanCtl,
     UINT8  bMode,
     UINT8  bBtr0,
     UINT8  bBtr1
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller
bMode	[in]	Operating mode of the CAN controller `CAN_OPMODE_STANDARD`: reception of 11-bit ID messages `CAN_OPMODE_EXTENDED`: reception of 29-bit ID messages `CAN_OPMODE_ERRFRAME`: error are indicated to the application via special CAN messages `CAN_OPMODE_LISTONLY`: listen only mode (TX passive) `CAN_OPMODE_LOWSPEED`: use of low speed bus interface `CAN_OPMODE_AUTOBAUD`: automatic bit rate detection
bBtr0	[in]	Value for the bus timing register 0 of the CAN controller. Value corresponds to the BTR0 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz.
bBtr1	[in]	Value for the bus timing register 1 of the CAN controller. Value corresponds to the BTR1 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The function resets the controller hardware internally according to the function *canControlReset* and then initializes the controller with the specified parameters. The function can be called from every controller status. For more information on the bus timing values in the parameters *bBtr0* and *bBtr1* see chapter *Initializing the Controller*.

canControlReset

Resets the controller hardware and resets the message filters of a CAN connection.

```
HRESULT EXTERN_C canControlReset (
     HANDLE hCanCtl
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

The function resets the controller hardware, removes the set acceptance filter, deletes the contents of the filter lists and switches the controller "offline". At the same time, the message flow between the controller and the connected message channels is interrupted. When the function is called, a currently active transmit process of the controller is aborted. This may lead to transmission errors or to a faulty message telegram on the bus.

canControlStart

Starts or stops the controller of a CAN connection.

```
HRESULT EXTERN_C canControlStart (
    HANDLE hCanCtl,
    BOOL   fStart
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller
fStart	[in]	Value TRUE starts and value FALSE stops the CAN controller.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

A call of the function is only successful when the CAN controller was previously configured with the function *canControlInitialize*. After a successful start of the CAN controller, it is actively connected to the bus. Incoming CAN messages are forwarded to all configured and activated message channels, or transmit messages issued by the message channels to the bus. A call of the function with the value FALSE in the parameter *fStart* switches the CAN controller "offline". The message transfer is thus interrupted and the CAN controller switched to passive status. Unlike the function *canControlReset*, the set acceptance filter and filter lists are not altered with a stop. Neither does the function simply stop a running transmit process of the controller but ends it in such a way that no faulty telegram is transferred to the bus.

canControlSetAccFilter

Sets the 11- or 29-bit acceptance filter of a CAN connection.

```
HRESULT EXTERN_C canControlSetAccFilter (
     HANDLE hCanCtl,
     BOOL   fExtend,
     UINT32 dwCode,
     UINT32 dwMask
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller
fExtend	[in]	Selection of the acceptance filter. The 11-bit acceptance filter is selected with value `FALSE` and the 29-bit acceptance filter with value `TRUE`.
dwCode	[in]	Bit sample of the identifier(s) to be accepted including RTR-bit.
dwMask	[in]	Bit sample of the relevant bits in *dwCode*. If a bit has the value 0 in dwMask, the corresponding bit in *dwCode* is not used for the comparison. If a bit has the value 1, it is relevant for the comparison.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

canControlAddFilterIds

Enters one or more IDs (CAN IDs) in the 11- or 29-bit filter list of a CAN connection.

```
HRESULT EXTERN_C canControlAddFilterIds (
     HANDLE hCanCtl,
     BOOL   fExtend,
     UINT32 dwCode,
     UINT32 dwMask
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller
fExtend	[in]	Selection of the filter list. The 11-bit filter list is selected with value `FALSE` and the 29-bit filter list with value `TRUE`.
dwCode	[in]	Bit sample of the identifier(s) to be identified including RTR-bit.
dwMask	[in]	Bit sample of the relevant bits in *dwCode*. If a bit has the value 0 in *dwMask*, the corresponding bit in *dwCode* is ignored. If a bit has the value 1, it is relevant.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

canControlRemFilterIds

Removes one or more IDs (CAN IDs) from the 11- or 29-bit filter list of a CAN connection.

```
HRESULT EXTERN_C canControlRemFilterIds (
     HANDLE hCanCtl,
     BOOL   fExtend,
     UINT32 dwCode,
     UINT32 dwMask
);
```

Parameter

Parameter	Dir.	Description
hCanCtl	[in]	Handle of the opened CAN controller
fExtend	[in]	Selection of the filter list. The 11-bit filter list is selected with value `FALSE` and the 29-bit filter list with value `TRUE`.
dwCode	[in]	Bit sample of the identifier(s) to be removed including RTR-bit.
dwMask	[in]	Bit sample of the relevant bits in *dwCode*. If a bit has the value 0 in *dwMask*, the corresponding bit in *dwCode* is ignored. If a bit has the value 1, it is relevant.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

5.3.2   Message Channel

canChannelOpen

Opens or creates a message channel for a CAN connection of a fieldbus adapter.

```
HRESULT EXTERN_C canChannelOpen (
     HANDLE  hDevice,
     UINT32  dwCanNo,
     BOOL    fExclusive,
     PHANDLE phCanChn
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the fieldbus adapter
dwCanNo	[in]	Number of the CAN connection for which a message channel is to be opened. Value 0 selects the first connection, value 1 the second connection and so on.
fExclusive	[in]	Defines whether the connection is used exclusively for the channel to be opened. If value `TRUE` is specified here, the CAN connection is used exclusively for the new message channel. With value `FALSE`, more than one message channel can be opened for the CAN connection.
phCanChn	[out]	Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened CAN message channel in this variable. In the event of an error, the variable is set to ZERO.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

If the value `TRUE` is specified in the parameter *fExclusive*, no more message channels can be opened after a successful call of the function. This means that the program that first calls the function with the value `TRUE` in the parameter *fExclusive* has exclusive control over the message flow on the CAN connection. If the value 0xFFFFFFFF is specified in the parameter *dwCanNo*, the function displays a dialog window to select an adapter and a CAN connection on the screen. In this case the function expects the handle of a higher order window, or the value ZERO if no higher order window is available, in the parameter *hDevice* and not the handle of the adapter. If the message channel is no longer required, the handle returned in *phCanChn* should be released again with the function <u>canChannelClose</u>.

canChannelClose

Closes an opened message channel.

```
HRESULT EXTERN_C canChannelClose (
    HANDLE hCanChn
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the message channel to be closed. The specified handle must come from a call of the function canChannelOpen.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

After the function is called, the handle specified in *hCanChn* is no longer valid and must no longer be used.

canChannelGetCaps

Determines the features of a CAN connection.

```
HRESULT EXTERN_C canChannelGetCaps (
    HANDLE          hCanChn,
    PCANCAPABILITIES pCanCaps
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
pCanCaps	[out]	Pointer to a structure of type *CANCAPABILITIES*. If run successfully, the function saves the features of the CAN connection in the memory area specified here.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

canChannelGetStatus

Determines the current status of a message channel as well as the current settings and the current status of the controller that is connected to the channel.

```
HRESULT EXTERN_C canChannelGetStatus (
    HANDLE         hCanChn,
    PCANCHANSTATUS pStatus
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
pStatus	[out]	Pointer to a structure of type *CANCHANSTATUS*. If run successfully, the function saves the current status of the channel and controller in the memory area specified here.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

canChannelInitialize

Initializes the receive and transmit buffers of a message channel.

```
HRESULT EXTERN_C canChannelInitialize (
    HANDLE hCanChn,
    UINT16 wRxFifoSize,
    UINT16 wRxThreshold,
    UINT16 wTxFifoSize,
    UINT16 wTxThreshold
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
wRxFifoSize	[in]	Size of the receive buffer in number of CAN messages
wRxThreshold	[in]	Threshold value for the receive event. The event is triggered when the number of messages in the receive buffer reaches or exceeds the number specified here.
wTxFifoSize	[in]	Size of the transmit buffer in number of CAN messages
wTxThreshold	[in]	Threshold value for the transmit event. The event is triggered when the number of free entries in the transmit buffer reaches or exceeds the number specified here.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

A value greater than 0 must be specified for the size of the receive and of the transmit buffer, otherwise the function returns an error code according to "Invalid parameter". The values

specified in the parameters *wRxFifoSize* and *wTxFifoSize* define the lower limit for the size of the buffers. The actual size of a buffer may be larger than the specified value, as the memory used for this is reserved page-wise. If the function is called for an already initialized channel, the function first deactivates the channel, then releases the available FIFOs and creates two new FIFOs with the required dimensions.

canChannelActivate

Activates or deactivates a message channel.

```
HRESULT EXTERN_C canChannelActivate (
     HANDLE hCanChn,
     BOOL   fEnable
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
fEnable	[in]	With value `TRUE`, the function activates the message flow between the CAN controller and the message channel, with value `FALSE` the function deactivates the message flow.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

As a default setting, the message channel is deactivated after opening or initializing. For the channel to receive messages from the bus, or send messages to the bus, the bus must be activated. At the same time, the CAN controller must be in the "online" status. For more information see function *linControlStart* and chapter *Initializing the Controller*. After activation of the channel, messages can be written in the transmit buffer with *canChannelPostMessage* or *canChannelSendMessage*, or read from the receive buffer with the functions *canChannelPeekMessage* and *canChannelReadMessage*.

canChannelPeekMessage

Reads the next CAN message from the receive buffer of a message channel.

```
HRESULT EXTERN_C canChannelPeekMessage (
     HANDLE  hCanChn,
     PCANMSG pCanMsg
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
pCanMsg	[out]	Pointer to a CANMSG structure where the function stores the retrieved CAN message. If this parameter is set to NULL, the function simply removes the next CAN message from the receive FIFO.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_RXQUEUE_EMPTY`	Currently no CAN message available in receive FIFO
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The function returns immediately to the calling program if no message is available for reading.

canChannelPeekMsgMult

Retrieves the next CAN messages from the receive FIFO of the specified CAN channel. The function does not wait for messages to be received from the CAN bus.

```
HRESULT EXTERN_C canChannelPeekMsgMult (
     HANDLE   hCanChn,
     PCANMSG2 paCanMsg,
     UINT32   dwCount,
     PUINT32  pdwDone
);
```

timeout

timeout	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
paCanMsg	[out]	Array of buffers where the function stores the retrieved CAN messages. If this timeout is set NULL, the function simply removes the specified number of CAN messages from the receive FIFO.
dwCount	[in]	Number of messages available in buffer
pdwDone	[out]	Pointer to a variable where the function stores the number of CAN messages actually read. This timeout is optional and can be NULL.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_RXQUEUE_EMPTY`	Currently no CAN message available in receive FIFO
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

canChannelPostMessage

Writes a CAN message in the transmit buffer of the specified message channel.

```
HRESULT EXTERN_C canChannelPostMessage (
     HANDLE  hCanChn,
     PCANMSG pCanMsg
);
```

timeout

timeout	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
pCanMsg	[in]	Pointer to an initialized structure of type *CANMSG* with the CAN message to be transmitted.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_TXQUEUE_FULL`	Not enough free space available within the transmit FIFO
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The function does not wait for the message to be transmitted on the bus.

canChannelPostMsgMult

Places the specified CAN messages in the transmit FIFO of the CAN channel without waiting for the messages to be transmitted over the CAN bus.

```
HRESULT EXTERN_C canChannelPostMsgMult (
     HANDLE  hCanChn,
     PCANMSG paCanMsg,
     UINT32  dwCount,
     PUINT32 pdwDone
);
```

timeout

timeout	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
paCanMsg	[in]	Pointer to array of transmit messages
dwCount	[in]	Number of valid messages in buffer
pdwDone	[out]	Pointer to a variable where the function stores the number of CAN messages written. This timeout is optional and can be NULL.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_TXQUEUE_FULL`	Not enough free space available within the transmit FIFO
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

canChannelWaitRxEvent

Waits until a CAN message is received from the CAN bus or the timeout interval elapses.

```
HRESULT EXTERN_C canChannelWaitRxEvent (
     HANDLE hCanChn,
     UINT32 dwTimeout
);
```

timeout

timeout	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
dwTimeout	[in]	Maximum waiting time in milliseconds. The function returns to the caller with the error code `VCI_E_TIMEOUT` if the receive event has not occurred in the time specified here. With value INFINITE (0xFFFFFFFF), the function waits until the receive event has occurred.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_TIMEOUT`	Timeout interval elapsed
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The receive event is triggered as soon as the number of messages in the receive buffer reaches or exceeds the set threshold. See the description of the function *canChannelInitialize*.

To check whether the receive event has already occurred without blocking the calling program, the value 0 can be specified in the timeout *dwTimeout* when calling the function. If the handle specified in *hCanChn* is closed from another thread, the function ends the current function control and returns with a return value not equal to `VCI_OK`.

canChannelWaitTxEvent

Waits until a CAN message can be written to the transmit FIFO or the timeout interval elapses.

```
HRESULT EXTERN_C canChannelWaitTxEvent (
     HANDLE hCanChn,
     UINT32 dwTimeout
);
```

timeout

timeout	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
dwTimeout	[in]	Timeout interval, in milliseconds. The function returns if the interval elapses, even if no message can be written to the transmit FIFO. If this timeout is zero, the function tests if a message can be written and returns immediately. If this timeout is INFINITE (0xFFFFFFFF), the timeout interval of the function never elapses.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_TIMEOUT`	Timeout interval elapsed
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The transmit event is triggered as soon as the transmit buffer contains the same number of free entries as the set threshold or more. See the description of the function *canChannelInitialize*. To check whether the transmit event has already occurred without blocking the calling program, the value 0 can be specified in the timeout *dwTimeout* when the function is called. If the handle specified in *hCanChn* is closed from another thread, the function ends the current function control and returns with a return value not equal to `VCI_OK`.

canChannelReadMessage

Retrieves the next CAN message from the receive FIFO of the specified CAN channel. The function waits for a message to be received from the CAN bus.

```
HRESULT EXTERN_C canChannelReadMessage (
     HANDLE  hCanChn,
     UINT32  dwTimeout,
     PCANMSG pCanMsg
);
```

timeout

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
dwTimeout	[in]	Maximum waiting time in milliseconds. The function returns to the caller with the error code `VCI_E_TIMEOUT` if no message is read or received within the specified time. With value INFINITE (0xFFFFFFFF), the function waits until a message has been read.
pCanMsg	[out]	Pointer to a *CANMSG* structure where the function stores the retrieved CAN message. If this parameter is set to NULL, the function simply removes the next CAN message from the FIFO.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_RXQUEUE_EMPTY`	Currently no CAN message available in receive FIFO
`VCI_E_TIMEOUT`	Timeout interval elapses without a CAN message available.
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

If the handle specified in *hCanChn* is closed from another thread, the function ends the current function control and returns with a return value not equal to `VCI_OK`.

canChannelReadMsgMult

Read the next CAN messages from the receive FIFO of the specified CAN channel. The function waits for CAN messages to be received from the CAN bus.

```
HRESULT EXTERN_C canChannelReadMsgMult (
    HANDLE  hCanChn,
    UINT32  dwTimeout,
    PCANMSG paCanMsg,
    UINT32  dwCount,
    PUINT32 pdwDone
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
dwTimeout	[in]	Maximum waiting time in milliseconds. The function returns to the caller with the error code `VCI_E_TIMEOUT` if no message is read or received within the specified time. With value INFINITE (0xFFFFFFFF), the function waits until a message has been read.
paCanMsg	[out]	Pointer to message buffer
dwCount	[in]	Number of slots in message buffer
pdwDone	[out]	Number of received CAN messages

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_RXQUEUE_EMPTY`	Currently no CAN message available in receive FIFO
`VCI_E_TIMEOUT`	Timeout interval elapses without a CAN message available.
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

If the handle specified in *hCanChn* is closed from another thread, the function ends the current function control and returns with a return value not equal to `VCI_OK`.

canChannelSendMessage

Places the specified CAN message in the transmit FIFO. The function waits until the message is placed in the transmit FIFO, but does not wait for the message to be transmitted over the CAN bus.

```
HRESULT EXTERN_C canChannelSendMessage (
    HANDLE  hCanChn,
    UINT32  dwTimeout,
    PCANMSG pCanMsg
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
dwTimeout	[in]	Timeout interval, in milliseconds. The function returns if the interval elapses, even if no message has been written to the transmit FIFO. If this parameter is zero, the function tries to write a message to the transmit FIFO and returns immediately. If this parameter is INFINITE (0xFFFFFFFF), the timeout interval of the function never elapses.
pCanMsg	[in]	Pointer to an initialized structure of type *CANMSG* with the CAN message to be transmitted.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_TXQUEUE_FULL`	*dwTimeout* is zero and there is currently no free space available within the transmit FIFO.
`VCI_E_TIMEOUT`	Specified timeout interval elapsed and there is no free space available within the transmit FIFO.
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The function only waits until the message is written in the transmit buffer, but not until the message is transmitted on the bus. If the handle specified in *hCanChn* is closed from another thread, the function ends the current function control and returns with a return value not equal to `VCI_OK`.

canChannelSendMsgMult

Places the specified CAN messages in the transmit FIFO. The function waits until the messages can be placed into the transmit FIFO, but does not wait for the messages to be transmitted over the CAN bus.

```
HRESULT EXTERN_C canChannelSendMsgMult (
      HANDLE  hCanChn,
      UINT32  dwTimeout,
      PCANMSG paCanMsg,
      UINT32  dwCount,
      PUINT32 pdwDone
);
```

Parameter

Parameter	Dir.	Description
hCanChn	[in]	Handle of the opened message channel
dwTimeout	[in]	Timeout interval, in milliseconds. The function returns if the interval elapses, even if no message has been written to the transmit FIFO. If this parameter is zero, the function tries to write a message to the transmit FIFO and returns immediately. If this parameter is INFINITE (0xFFFFFFFF), the timeout interval of the function never elapses.
paCanMsg	[in]	Pointer to array of CAN message to transmit.
dwCount	[in]	Number of valid entries in message array
pdwDone	[out]	Number of sent CAN messages

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_TXQUEUE_FULL`	*dwTimeout* is zero and there is currently no free space available within the transmit FIFO.
`VCI_E_TIMEOUT`	Specified timeout interval elapsed and there is no free space available within the transmit FIFO.
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The function only waits until the last message is written in the transmit buffer, but not until the last message is transmitted on the bus. If the handle specified in *hCanChn* is closed from another thread, the function ends the current function control and returns with a return value not equal to `VCI_OK`.

5.3.3 Cyclic Transmit List

canSchedulerOpen

Opens the cyclic transmit list of a CAN connection on a fieldbus adapter.

```
HRESULT EXTERN_C canSchedulerOpen (
    HANDLE  hDevice,
    UINT32  dwCanNo,
    PHANDLE phCanShd
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the fieldbus adapter
dwCanNo	[in]	Number of the CAN connection of the transmit list to be opened. Value 0 selects the first connection, value 1 the second connection and so on.
phCanShd	[out]	Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened transmit list in this variable. In the event of an error, the variable is set to ZERO.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

If the value 0xFFFFFFFF is specified in the parameter *dwCanNo*, the function displays a dialog window to select an adapter and a CAN connection on the screen. In this case the function expects the handle of a higher order window, or the value ZERO if no higher order window is available, in the parameter *hDevice* and not the handle of the adapter.

canSchedulerClose

Closes an opened cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerClose (
    HANDLE hCanShd
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the transmit list to be closed. The specified handle must come from a call of the function *canSchedulerOpen*.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

After the function is called, the handle specified in *hCanShd* is no longer valid and must no longer be used.

canSchedulerGetCaps

Determines the features of the CAN connection of the specified cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerGetCaps (
    HANDLE          hCanShd,
    PCANCAPABILITIES pCanCaps
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list
pCanCaps	[out]	Pointer to a structure of type CANCAPABILITIES. If run successfully, the function saves the features of the CAN connection in the memory area specified here.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

canSchedulerGetStatus

Determines the current status of the transmit task and of all registered transmit objects of a cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerGetStatus (
    HANDLE             hCanShd,
    PCANSCHEDULERSTATUS pStatus
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list
pStatus	[out]	Pointer to a structure of type CANSCHEDULERSTATUS. If run successfully, the function saves the current status of all cyclic transmit objects in the memory area specified here.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

The function returns the current status of all 16 transmit objects in the table CANSCHEDULERSTATUS.abMsgStat. The list index provided by the function canSchedulerAddMessage can be used to request the status of an individual transmit object, i.e. abMsgStat[Index] contains the status of the transmit object with the specified index.

canSchedulerActivate

Starts or stops the transmit task of the cyclic transmit list and thus the cyclic transmit process of all currently registered transmit objects.

```
HRESULT EXTERN_C canSchedulerActivate (
     HANDLE hCanShd,
     BOOL   fEnable
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list
fEnable	[in]	With value TRUE the function activates, and with value FALSE deactivates, the cyclic transmit process of all currently registered transmit objects.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

The function can be used to start all registered transmit objects simultaneously. For this, all transmit objects are first set to started status with the function *canSchedulerStartMessage*. A subsequent call of this function with the value TRUE for the parameter *fEnable* then guarantees a simultaneous start. If the function is called with the value FALSE for the parameter *fEnable*, processing of all registered transmit objects is stopped simultaneously.

canSchedulerReset

Stops the transmit task and removes all transmit objects from the specified cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerReset (
     HANDLE hCanShd
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

canSchedulerAddMessage

Adds a new transmit object to the specified cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerAddMessage (
    HANDLE         hCanShd,
    PCANCYCLICTXMSG pMessage,
    PUINT32        pdwIndex
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list
pMessage	[in]	Pointer to an initialized structure of type *CANCYCLICTXMSG* with the transmit object.
pdwIndex	[out]	Pointer to a variable of type UINT32. If run successfully, the function returns the list index of the newly added transmit object in this variable. In the event of an error, the variable is set to the value 0xFFFFFFFF (-1). This index is required for all further function calls.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

The cyclic transmit process of the newly added transmit object begins only after a successful call of the function *canSchedulerStartMessage*. In addition, the transmit list must be active (see *canSchedulerActivate*).

canSchedulerRemMessage

Stops processing of a transmit object and removes it from the specified cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerRemMessage (
    HANDLE hCanShd,
    UINT32 dwIndex
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list
dwIndex	[in]	List index of the transmit object to be removed. The list index must come from a previous call of the function *canSchedulerAddMessage*.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

After the function is called, the list index specified in *dwIndex* is invalid and must no longer be used.

canSchedulerStartMessage

Starts a transmit object of the specified cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerStartMessage (
     HANDLE hCanShd,
     UINT32 dwIndex,
     UINT16 wRepeat
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list
dwIndex	[in]	List index of the transmit object to be started. The list index must come from a previous call of the function *canSchedulerAddMessage*.
wRepeat	[in]	Number of the cyclic transmit repeats. With value 0, the transmit process is repeated infinitely. The specified value must be in the range 0 to 65535.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

The cyclic transmit process only starts if the transmit task is active when the function is called. If the transmit task is inactive, the transmit process is delayed until the next call of the function *canSchedulerActivate*.

canSchedulerStopMessage

Stops a transmit object of the specified cyclic transmit list.

```
HRESULT EXTERN_C canSchedulerStopMessage (
     HANDLE hCanShd,
     UINT32 dwIndex
);
```

Parameter

Parameter	Dir.	Description
hCanShd	[in]	Handle of the opened transmit list
dwIndex	[in]	List index of the transmit object to be stopped. The list index must come from a previous call of the function *canSchedulerAddMessage*.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

5.4   Functions for LIN Access

5.4.1   Control Unit

linControlOpen

Opens the control unit of a LIN connection on a fieldbus adapter.

```
HRESULT EXTERN_C linControlOpen (
      HANDLE  hDevice,
      UINT32  dwLinNo,
      PHANDLE phLinCtl
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the fieldbus adapter
dwLinNo	[in]	Number of the LIN connection of the control unit to be opened. Value 0 selects the first connection, value 1 the second connection and so on.
phLinCtl	[out]	Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened LIN controller in this variable. In the event of an error, the variable is set to ZERO.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linControlClose

Closes an opened LIN controller.

```
HRESULT EXTERN_C linControlClose (
      HANDLE hLinCtl
);
```

Parameter

Parameter	Dir.	Description
hLinCtl	[in]	Handle of the LIN controller to be closed. The specified handle must come from a call of the function *canControlOpen*.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

After the function is called, the handle specified in *hLinCtl* is no longer valid and must no longer be used.

linControlGetCaps

Determines the features of a LIN connection.

```
HRESULT EXTERN_C linControlGetCaps (
    HANDLE          hLinCtl,
    PLINCAPABILITIES pLinCaps
);
```

Parameter

Parameter	Dir.	Description
hLinCtl	[in]	Handle of the opened LIN controller
pLinCaps	[out]	Pointer to a structure of type *LINCAPABILITIES*. If run successfully, the function saves the features of the LIN connection in the memory area specified here.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linControlGetStatus

Determines the current settings and the current status of the controller of a LIN connection.

```
HRESULT EXTERN_C linControlGetStatus (
    HANDLE          hLinCtl,
    PLINLINESTATUS pStatus
);
```

Parameter

Parameter	Dir.	Description
hLinCtl	[in]	Handle of the opened LIN controller
pStatus	[out]	Pointer to a structure of type *LINLINESTATUS*. If run successfully, the function saves the current settings and the status of the controller in the memory area specified here.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linControlInitialize

Sets the operating mode and bit rate of a LIN connection.

```
HRESULT EXTERN_C linControlInitialize (
    HANDLE hLinCtl,
    UINT8  bMode,
    UINT16 wBitrate
);
```

Parameter

Parameter	Dir.	Description
hLinCtl	[in]	Handle of the opened LIN controller
bMode	[in]	Operating mode of the LIN controller `LIN_OPMODE_SLAVE`: Slave mode, activated by default `LIN_OPMODE_MASTER`: Master mode (if supported, see *LINCAPABILITIES*) `LIN_OPMODE_ERRORS`: error are indicated to the application via special LIN messages
wBitrate	[in]	Bit rate of the LIN controller. Valid values are between 1000 and 20000 bit/s, resp. between the values that are specified by `LIN_BITRATE_MIN` and `LIN_BITRATE_MAX`. If the controller supports automatic bit rate detection, enter `LIN_BITRATE_AUTO` to activate the automatic bit rate detection.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linControlReset

Resets the specified LIN controller to its initial state. The function aborts the current message transmission and switches the LIN controller into INIT mode.

```
HRESULT EXTERN_C linControlReset (
    HANDLE hLinCtl
);
```

Parameter

Parameter	Dir.	Description
hLinCtl	[in]	Handle of the opened LIN controller

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linControlStart

Starts or stops the controller of a LIN connection.

```
HRESULT EXTERN_C linControlStart (
     HANDLE hLinCtl,
     BOOL   fStart
);
```

Parameter

Parameter	Dir.	Description
hLinCtl	[in]	Handle of the opened LIN controller
fStart	[in]	Value `TRUE` starts and value `FALSE` stops the LIN controller.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linControlWriteMessage

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 EXTERN_C linControlWriteMessage (
     HANDLE  hLinCtl,
     BOOL    fSend,
     PLINMSG pLinMsg
);
```

Parameter

Parameter	Dir.	Description
hLinCtl	[in]	Handle of the opened LIN controller
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

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

5.4.2 Message Monitor

The interface provides functions to install a message monitor between the application and the LIN bus.

linMonitorOpen

Opens a LIN message monitor on the specified LIN controller.

```
HRESULT EXTERN_C linMonitorOpen (
     HANDLE  hDevice,
     UINT32  dwLinNo,
     BOOL    fExclusive,
     PHANDLE phLinMon
);
```

Parameter

Parameter	Dir.	Description
hDevice	[in]	Handle of the device where the LIN controller is located.
dwLinNo	[in]	Number of the LIN controller to open, value 0 selects the first LIN connection, value 1 the second LIN connection, etc. (see Remarks).
fExclusive	[in]	If this parameter is set to TRUE the function tries to acquire exclusive access to the LIN message monitor, and no further monitors can be created. If set to FALSE the function opens the monitor in shared mode and any number of message monitors can be created for the LIN connection.
phLinMon	[out]	Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened LIN controller in this variable. In the event of an error, the variable is set to ZERO.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

If *dwLinNo* is set to 0xFFFFFFFF, the function shows a dialog box which allows the user to select the VCI device and LIN controller. In this case *hDevice* must contain the handle to the window that owns the dialog box.

linMonitorClose

Closes an opened LIN monitor.

```
HRESULT EXTERN_C linMonitorClose (
     HANDLE hLinMon
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

The handle specified by the parameter *hLinMon* is not longer valid after the function returns and must not be used any longer.

linMonitorGetCaps

Determines the features of a LIN connection.

```
HRESULT EXTERN_C linMonitorGetCaps (
     HANDLE          hLinMon,
     PLINCAPABILITIES pLinCaps
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
pLinCaps	[out]	Points to a *LINCAPABILITIES* structure where the function stores the capabilities of the LIN controller.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

linMonitorGetStatus

Determines the current settings and the current status of the controller of a LIN connection.

```
HRESULT EXTERN_C linMonitorGetStatus (
     HANDLE          hLinMon,
     PLINMONITORSTATUS pStatus
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
pStatus	[out]	Points to a *LINMONITORSTATUS* structure where the function stores the current status of the LIN monitor.

Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

linMonitorInitialize

Initializes the FIFO size of a LIN monitor.

```
HRESULT EXTERN_C linMonitorInitialize (
     HANDLE hLinMon,
     UINT16 wFifoSize,
     UINT16 wThreshold
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
wFifoSize	[in]	Size of receive FIFO in number of LIN messages
wThreshold	[in]	Threshold value for the receive event. Event is triggered when the number of messages in the receive FIFO reaches the defined number.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linMonitorActivate

This function activates or deactivates the LIN monitor. After activating the monitor, LIN messages are received from the LIN bus by calling the receive functions. After deactivating the monitor, no further messages are received from the LIN bus.

```
HRESULT EXTERN_C linMonitorActivate (
     HANDLE hLinMon,
     BOOL   fEnable
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
fEnable	[in]	`TRUE` activates the connection between LIN controller and message monitor, `FALSE` deactivates the connection (default: `FALSE`).

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The LIN controller must be started, otherwise no messages are received from the LIN bus (see *linControlStart*).

linMonitorPeekMessage

Retrieves the next LIN message from the receive FIFO of the specified monitor. The function does not wait for a message to be received from the LIN bus.

```
HRESULT EXTERN_C linMonitorPeekMessage (
     HANDLE  hLinMon,
     PLINMSG pLinMsg
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
pLinMsg	[out]	Pointer to a *LINMSG* structure where the function stores the retrieved LIN message. If this parameter is set to NULL, the function simply removes the next LIN message from the receive FIFO.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_RXQUEUE_EMPTY`	Currently no CAN message available in receive FIFO
`VCI_E_TIMEOUT`	Timeout interval elapses without a CAN message available
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linMonitorPeekMsgMult

Retrieves the next LIN messages from the receive FIFO of the specified LIN monitor. The function does not wait for messages to be received from the LIN bus.

```
HRESULT EXTERN_C linMonitorPeekMsgMult (
     HANDLE  hLinMon,
     PLINMSG paLinMsg,
     UINT32  dwCount,
     PUINT32 pdwDone
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
paLinMsg	[out]	Array of buffers where the function stores the retrieved LIN messages. If this parameter is set NULL, the function simply removes the specified number of LIN messages from the receive FIFO.
dwCount	[in]	Number of available slots in LIN message buffer
pdwDone	[out]	Pointer to a variable where the function stores the number of LIN messages actually read. This parameter is optional and can be NULL.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`VCI_E_RXQUEUE_EMPTY`	Currently no LIN message available in receive FIFO
`VCI_E_TIMEOUT`	Timeout interval elapses without a LIN message available.
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

linMonitorWaitRxEvent

This function waits until a LIN message is received from the LIN bus or the timeout interval elapses.

```
HRESULT EXTERN_C linMonitorWaitRxEvent (
      HANDLE hLinMon,
      UINT32 dwTimeout
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
dwTimeout	[in]	Maximum waiting time in milliseconds. The function returns to the caller with the error code `VCI_E_TIMEOUT` if the receive event has not occurred in the time specified here. With value INFINITE (0xFFFFFFFF), the function waits until the receive event has occurred.

Return Value

Return value	Description
`VCI_OK`	Function succeeded
`!=VCI_OK`	Error, more information about error code provides the function VciFormatError

Remark

The transmit event is triggered as soon as the transmit buffer contains the same number of free entries as the set threshold or more. See the description of the function *linMonitorInitialize*. To check whether the transmit event has already occurred without blocking the calling program, the value 0 can be specified in the parameter *dwTimeout* when the function is called. If the handle specified in *hLinMon* is closed from another thread, the function ends the current function control and returns with a return value not equal to `VCI_OK`.

linMonitorReadMessage

Reads the next LIN message from the receive buffer of a LIN message monitor.

```
HRESULT EXTERN_C linMonitorReadMessage (
      HANDLE  hLinMon,
      UINT32  dwTimeout,
      PLINMSG pLinMsg
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
dwTimeout	[in]	Timeout interval, in milliseconds. The function returns if the interval elapses, even if no message is received from the LIN bus. If this parameter is zero, the function tests if a message is available and returns immediately. If this parameter is INFINITE (0xFFFFFFFF), the timeout interval of the function never elapses.
pLinMsg	[out]	Pointer to a *LINMSG* structure where the function stores the retrieved LIN message. If this parameter is set to NULL, the function simply removes the next LIN message from the FIFO.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_RXQUEUE_EMPTY	Currently no LIN message available in receive FIFO
VCI_E_TIMEOUT	Timeout interval elapses without a CAN message available.
!=VCI_OK	Error, more information about error code provides the function VciFormatError

Remark

If the handle specified in *hLinMon* is closed from another thread, the function ends the current function control and returns with a return value not equal to VCI_OK.

linMonitorReadMsgMult

Read the next LIN messages from the receive FIFO of the specified LIN monitor. The function waits for LIN messages to be received from the LIN bus.

```
HRESULT EXTERN_C linMonitorReadMsgMult (
    HANDLE  hLinMon,
    UINT32  dwTimeout,
    PLINMSG paLinMsg,
    UINT32  dwCount,
    PUINT32 pdwDone
);
```

Parameter

Parameter	Dir.	Description
hLinMon	[in]	Handle of the opened LIN monitor
dwTimeout	[in]	Timeout interval, in milliseconds. The function returns if the interval elapses, even if no message is received from the LIN bus. If this parameter is zero, the function tests if a message is available and returns immediately. If this parameter is INFINITE (0xFFFFFFFF), the timeout interval of the function never elapses.
paLinMsg	[out]	Array of buffers where the function stores the retrieved LIN messages. If this parameter is set NULL, the function simply removes the specified number of LIN messages from the receive FIFO.
dwCount	[in]	Size of the array pointed to by *paLinMsg* in count of LIN messages.
pdwDone	[out]	Pointer to a variable where the function stores the number of LIN messages actually read. This parameter is optional and can be NULL.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_RXQUEUE_EMPTY	Currently no LIN message available in the receive FIFO
VCI_E_TIMEOUT	Timeout interval elapses without a CAN message available.
!=VCI_OK	Error, more information about error code provides the function VciFormatError

6 Data Types

6.1 VCI-Specific Data Types

6.1.1 VCIID

Unique VCI object identifier.

```
typedef struct _VCIID
{
  LUID AsLuid;
  T64 AsInt64;
} VCIID, *PVCIID;
```

Member	Dir.	Description
AsLuid	[out]	ID in form of a LUID. Data type LUID is defined in Windows.
AsInt64	[out]	ID as a signed 64 bit integer.

6.1.2 VCIVERSIONINFO

The structure describes the VCI and OS version information.

```
typedef struct _VCIVERSIONINFO
{
  UINT32 VciMajorVersion;
  UINT32 VciMinorVersion;
  UINT32 VciRevNumber;
  UINT32 VciBuildNumber;
  UINT32 OsMajorVersion;
  UINT32 OsMinorVersion;
  UINT32 OsBuildNumber;
  UINT32 OsPlatformId;
} VCIVERSIONINFO, *PVCIVERSIONINFO;
```

Member	Dir.	Description
VciMajorVersion	[out]	Major version number of VCI
VciMinorVersion	[out]	Minor version number of VCI
VciRevNumber	[out]	Revision number of VCI
VciBuildNumber	[out]	Build number of VCI
OsMajorVersion	[out]	Major version number of operating system
OsMinorVersion	[out]	Minor version number of operating system
OsBuildNumber	[out]	Build number of operating system
OsPlatformId	[out]	Platform id of operating system

6.1.3 VCILICINFO

The structure describes the VCI license information.

```
typedef struct _VCILICINFO
{
  GUID DeviceClass;
  UINT32 MaxDevices;
  UINT32 MaxRuntime;
  UINT32 Restrictions;
} VCILICINFO, *PVCILICINFO;
```

Member	Dir.	Description
DeviceClass	[out]	Class ID of the licensed product
MaxDevices	[out]	Maximum number of allowed devices (0=no limit)
MaxRuntime	[out]	Maximum runtime in seconds (0=no limit)
Restrictions	[out]	Additional restrictions: `VCI_LICX_NORESTRICT`: no additional restrictions `VCI_LICX_SINGLEUSE`: single application use only

6.1.4 VCIDRIVERINFO

The structure describes the VCI driver information.

```
typedef struct _VCIDRIVERINFO
{
  VCIID VciObjectId;
  GUID DriverClass;
  UINT16 MajorVersion;
  UINT16 MinorVersion;
} VCIDRIVERINFO, *PVCIDRIVERINFO;
```

Member	Dir.	Description
VciObjectId	[out]	Unique VCI object identifier
DriverClass	[out]	Driver class identifier
MajorVersion	[out]	Major driver version number
MinorVersion	[out]	Minor driver version number

6.1.5　VCIDEVICEINFO

The structure describes the VCI device information.

```
typedef struct _VCIDEVICEINFO
{
  VCIID VciObjectId;
  GUID DeviceClass;
  UINT8 DriverMajorVersion;
  UINT8 DriverMinorVersion;
  UINT16 DriverBuildVersion;
  UINT8 HardwareBranchVersion;
  UINT8 HardwareMajorVersion;
  UINT8 HardwareMinorVersion;
  UINT8 HardwareBuildVersion;
  GUID_OR_CHARS UniqueHardwareId;
  CHAR Description[128];
  CHAR Manufacturer[126];
  UINT16 DriverReleaseVersion;
} VCIDEVICEINFO, *PVCIDEVICEINFO;
```

Member	Dir.	Description
VciObjectId	[out]	Unique VCI object identifier (VCIID), when a device logs in, it is allocated the system-wide unique ID (VCIID). The ID is required to access the interface.
DeviceClass	[out]	Device class identifier (GUID). All device drivers identify their supported device class by the worldwide unique device class ID. Different devices belong to different device classes, for example the USB-to-CAN belongs to a different device class than PCI04/PCI.
DriverMajorVersion	[out]	Major version number of driver
DriverMinorVersion	[out]	Minor version number of driver
DriverBuildVersion	[out]	Build version number of 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 hardware identifier. Each device has a unique hardware ID. The ID can be used to differentiate between two interfaces or to search for a device with a certain hardware ID.
Description	[out]	Device description as 0-terminated ASCII character string
Manufacturer	[out]	Device manufacturer as 0-terminated ASCII character string
DriverReleaseVersion	[out]	Release version number of driver

6.1.6　VCIDEVICECAPS

The structure describes the capabilities of a VCI device.

```
typedef struct _VCIDEVICECAPS
{
  UINT16 BusCtrlCount;
  UINT16 BusCtrlTypes[32];
} VCIDEVICECAPS, *PVCIDEVICECAPS;
```

Member	Dir.	Description
BusCtrlCount	[out]	Number of supported bus controllers
BusCtrlTypes	[out]	Array of supported bus controllers

6.1.7 VCIDEVRTINFO

The structure describes the run-time status information of a VCI device.

```
typedef struct _VCIDEVRTINFO
{
  UINT32 dwCommId;
  UINT32 dwStatus;
} VCIDEVRTINFO, *PVCIDEVRTINFO;
```

Member	Dir.	Description
dwCommId	[out]	ID of currently running communication layer
dwStatus	[out]	Runtime status flags `VCI_DEVRTI_STAT_LICEXP`: runtime of license expired `VCI_DEVRTI_STAT_DISCON`: device driver disconnected

6.2 CAN-Specific Data Types

6.2.1 CANBTRTABLE

The structure describes the CAN controller bit timing table.

```
typedef struct _CANBTRTABLE
{
  UINT8 bCount;
  UINT8 bIndex;
  UINT8 abBtr0[64];
  UINT8 abBtr1[64];
} CANBTRTABLE, *PCANBTRTABLE;
```

Member	Dir.	Description
bCount	[out]	Number of values within the BTR tables
bIndex	[out]	Index of the selected value within the BTR table
abBtr0	[out]	Test values for bus timing register (BTR) 0
abBtr1	[out]	Test values for bus timing register (BTR) 1

6.2.2 CANCAPABILITIES

The structure describes the CAN controller capabilities.

```
typedef struct _CANCAPABILITIES
{
  UINT16 wCtrlType;
  UINT16 wBusCoupling;
  UINT32 dwFeatures;
  UINT32 dwClockFreq;
  UINT32 dwTscDivisor;
  UINT32 dwCmsDivisor;
  UINT32 dwCmsMaxTicks;
  UINT32 dwDtxDivisor;
  UINT32 dwDtxMaxTicks;
} CANCAPABILITIES, *PCANCAPABILITIES;
```

Member	Dir.	Description
wCtrlType	[out]	Type of CAN controller (see list of controller types in *cantype.h* `CAN_CTRL_`)
wBusCoupling	[out]	Type of bus coupling `CAN_BUSC_LOWSPEED`: low-speed coupling of CAN controller `CAN_BUSC_HIGHSPEED`: high-speed coupling of CAN controller
dwFeatures	[out]	Supported features, value is one of the following constants or a combination: `CAN_FEATURE_STDOREXT`: 11 or 29 bit messages (exclusive, not simultaneously) `CAN_FEATURE_STDANDEXT`: 11 and 29 bit messages (simultaneously) `CAN_FEATURE_RMTFRAME`: reception of remote frames `CAN_FEATURE_ERRFRAME`: reception of error frames `CAN_FEATURE_BUSLOAD`: calculation of the bus load `CAN_FEATURE_IDFILTER`: exact filtering of messages `CAN_FEATURE_LISTONLY`: listen only mode `CAN_FEATURE_SCHEDULER`: cyclic transmit list (message scheduler) `CAN_FEATURE_GENERRFRM`: generation of error frames `CAN_FEATURE_DELAYEDTX`: delayed transmission of messages
dwClockFreq	[out]	Clock frequency of the primary counter in Hz
dwTscDivisor	[out]	Divisor for the message time stamp counter. The time stamp counter provides the time stamp for CAN messages. The frequency of the time stamp counter is calculated from the frequency of the primary timer divided by the value specified here.

Member	Dir.	Description
dwCmsDivisor	[out]	Divisor for the cyclic message scheduler. The frequency of the timer is calculated from the frequency of the primary timer divided by the value specified here. If no cyclic transmit list is available, the value is 0.
dwCmsMaxTicks	[out]	Maximum tick count value of the cyclic message scheduler. If no cyclic transmit list is available, the value is 0.
dwDtxDivisor	[out]	Divisor for the delayed message transmitter. The frequency of the timer is calculated from the frequency of the primary timer divided by the value specified here. If delayed transmission is not supported, the value is 0.
dwDtxMaxTicks	[out]	Maximum delay time of the delayed message transmitter in timer ticks. If delayed transmission is not supported, the value is 0.

6.2.3 CANINITLINE

The structure describes the CAN controller initialization parameters.

```
typedef struct _CANINITLINE
{
   UINT8 bOpMode;
   UINT8 bReserved;
   UINT8 bBtReg0;
   UINT8 bBtReg1;
} CANINITLINE, *PCANINITLINE;
```

Member	Dir.	Description
bOpMode	[out]	CAN operating mode, contains the value that is defined by function *canControlInitialize*. Value is a logical combination of one or more CAN_OPMODE_ constants (see *canControlInitialize*).
bReserved	[out]	Reserved set to 0
bBtReg0	[out]	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. For more information see data sheet of SJA 1000.
bBtReg1	[out]	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. For more information see data sheet of SJA 1000.

6.2.4 CANLINESTATUS

The structure describes the CAN controller status.

```
typedef struct _CANLINESTATUS
{
   UINT8 bOpMode;
   UINT8 bBtReg0;
   UINT8 bBtReg1;
   UINT8 bBusLoad;
   UINT8 dwStatus;
} CANLINESTATUS, *PCANLINESTATUS;
```

Member	Dir.	Description
bOpMode	[out]	Current operating mode of controller. CAN operating mode, contains the value that is defined by function *canControlInitialize* Value is a logical combination of one or more CAN_OPMODE_ constants (see *canControlInitialize*).
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. For more 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. For more information see data sheet of SJA 1000.

Member	Dir.	Description
bBusLoad	[out]	Bus load in the second before the call of the function in percentage (0 to 100). Value shows a state. To monitor the bus load over a time span use appropriate analysis tools. Value is exclusively valid if calculation of bus load is supported by the controller (see *CANCAPABILITIES*).
dwStatus	[out]	Current status of CAN controller `CAN_STATUS_TXPEND`: transmitting a message to the bus `CAN_STATUS_OVRRUN`: data overrun in receive buffer occurred (reset CAN controller) `CAN_STATUS_ERRLIM`: overrun of error counter `CAN_STATUS_BUSOFF`: bus off status `CAN_STATUS_ININIT`: stopped status, init mode active `CAN_STATUS_BUSCERR`: bus coupling error

6.2.5 CANCHANSTATUS

The structure describes the CAN message channel status.

```
typedef struct _CANCHANSTATUS
{
  CANLINESTATUS sLineStatus;
  BOOL32 fActivated;
  BOOL32 fRxOverrun;
  UINT8 bRxFifoLoad;
  UINT8 bTxFifoLoad;
} CANCHANSTATUS, *PCANCHANSTATUS;
```

Member	Dir.	Description
sLineStatus	[out]	Current status of CAN line (for more information see *CANLINESTATUS*)
fActivated	[out]	`TRUE` if the channel is activated, `FALSE` if channel is deactivated
fRxOverrun	[out]	`TRUE` if receive FIFO overrun occurred
bRxFifoLoad	[out]	Receive FIFO load in percent (0..100)
bTxFifoLoad	[out]	Transmit FIFO load in percent (0..100)

6.2.6 CANSCHEDULERSTATUS

The structure describes the current status of the cyclic transmitting list.

```
typedef struct _CANSCHEDULERSTATUS
{
  UINT8 bTaskStat;
  UINT8 abMsgStat[16];
} CANSCHEDULERSTATUS, *PCANSCHEDULERSTATUS;
```

Member	Dir.	Description
bTaskStat	[out]	Current status of transmitting task `CAN_TXTSK_STAT_STOPPED`: stopped or deactivated `CAN_TXTSK_STAT_RUNNING`: running
abMsgStat	[out]	Table with status of all 16 transmitting objects, possible values for the table entries: `CAN_CTXMSG_STAT_EMPTY`: entry not allocated to a transmitting task, or entry is currently not used. `CAN_CTXMSG_STAT_BUSY`: transmit message is being processed `CAN_CTXMSG_STAT_DONE`: processing of transmit message is completed

6.2.7   CANMSGINFO

The structure describes the CAN message information in a 32-bit value. The value can be addressed either byte-wise or via individual bit fields.

```
typedef struct _CANMSGINFO
{
  UINT8 bType;
  UINT8 bFlags2;
  UINT8 bFlags;
  UINT8 bAccept;
} CANMSGINFO, *PCANMSGINFO;
```

Member	Dir.	Description
bType	[in/out]	Type of message `CAN_MSGTYPE_DATA`: data message, see description of the message structure in *CANMSG* `CAN_MSGTYPE_INFO`: information message (only receive messages) `CAN_MSGTYPE_ERROR`: error frame (only receive messages) `CAN_MSGTYPE_STATUS`: status frame (only receive messages) `CAN_MSGTYPE_WAKEUP`: wakeup frame (only receive messages) `CAN_MSGTYPE_TIMEOVR`: timer overrun (only receive messages) `CAN_MSGTYPE_TIMERST`: timer reset (only receive messages)
bFlags2	[out]	Extended message flags `CAN_MSGFLAGS2_SSM`: [bit 0] single shot mode `CAN_MSGFLAGS2_HPM`: [bit 1] high priority message `CAN_MSGFLAGS2_EDL`: [bit 2] extended data length `CAN_MSGFLAGS2_FDR`: [bit 3] fast data bit rate `CAN_MSGFLAGS2_ESI`: [bit 4] error state indicator `CAN_MSGFLAGS2_RES`: [bit 5..7] reserved bits
bFlags	[out]	Standard message flags `CAN_MSGFLAGS_DLC`: [bit 0] data length code `CAN_MSGFLAGS_OVR`: [bit 4] data overrun flag `CAN_MSGFLAGS_SRR`: [bit 5] self reception request `CAN_MSGFLAGS_RTR`: [bit 6] remote transmission request `CAN_MSGFLAGS_EXT`: [bit 7] frame format (0 = 11 bit, 1= 29 bit
bAccept	[out]	Shows in receive messages which filter has accepted the message (see CAN_ACCEPT_ constants)

6.2.8   CANMSG

The structure describes the CAN message structure.

```
typedef struct _CANMSG
{
  UINT32 dwTime;
  UINT32 dwMsgId;
  CANMSGINFO uMsgInfo;
  UINT8 abData[8];
} CANMSG, *PCANMSG;
```

Member	Dir.	Description
dwTime	[out]	In receiving messages this field contains the starting point of the message in ticks. For more information see *Reception Time of a Message, p. 18*. In transmitting messages this fields determines with how many ticks delay the message is transmitted after the message sent before.
dwMsgId	[out]	CAN ID of the message in Intel format (aligned right) without RTR bit.
uMsgInfo	[out]	Bit field with information about the message type. For detailed description of bit field see *CANMSGINFO*.
abData	[out]	Array for up to 8 data bytes. Number of valid data bytes is defined by field uMsgInfo.Bits.dlc.

> ❗ Note that, when using interfaces with FPGA, error frames get the same time stamp (*field dwTime*) as the last received CAN message.

6.2.9   CANCYCLICTXMSG

The structure describes the cyclic transmit message.

```
typedef struct _CANCYCLICTXMSG
{
  UINT16 wCycleTime;
  UINT8 bIncrMode;
  UINT8 bByteIndex;
  UINT32 dwMsgId;
  CANMSGINFO uMsgInfo;
  UINT8 abData[8];
} CANCYCLICTXMSG, *PCANCYCLICTXMSG;
```

Member	Dir.	Description
wCycleTime	[out]	Cycle time of the transmit message in number of ticks. The cycle time can be calculated in the fields dwClockFreq and dwCmsDivisor of structure *CANCAPABILITIES* with the following formula. T cycle [s] = (dwCmsDivisor / dwClockFreq) * wCycleTime The maximum value for the field is limited to the value in field dwCmsMaxTicks of structure *CANCAPABILITIES*.
bIncrMode	[out]	Determines if a part of the cyclic transmitting list is automatically incremented after each transmitting: `CAN_CTXMSG_INC_NO`: no increment `CAN_CTXMSG_INC_ID`: Increments the field *dwMsgId* of the message after every transmit process by 1. If the field reaches the value 2048 (11-bit ID) or 536.870.912 (29-bit ID), there is an automatic overrun to 0. `CAN_CTXMSG_INC_8`: Increments an 8-bit value in the data field *abData* of the message. The data field to be incremented is defined in the field *bByteIndex*. If the maximum value 255 is exceeded, there is an automatic overrun to 0. `CAN_CTXMSG_INC_16`: Increments a 16-bit value in the data field *abData* of the message. The least significant byte of the 16-bit value to be incremented is defined in the field *bByteIndex*. The most significant byte is in *abData[bByteIndex+1]*. If the maximum value 65535 is exceeded, there is an automatic overrun to 0.

Member	Dir.	Description
bByteIndex	[out]	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	[out]	CAN ID of the message in Intel format (aligned right) without RTR bit
uMsgInfo	[out]	Bit field with information about the message type. For description of bit field see *CANMSGINFO*.
abData	[out]	Message data

6.3 LIN-Specific Data Types

6.3.1 LININITLINE

The structure contains the controller initialization parameters.

```
typedef struct _LININITLINE
{
  UINT8 bOpMode;
  UINT8 bReserved;
  UINT16 wBitrate;
} LININITLINE, *PLININITLINE;
```

Member	Dir.	Description
bOpMode	[out]	Operating mode of controller `LIN_OPMODE_SLAVE`: Slave mode `LIN_OPMODE_MASTER`: Master mode `LIN_OPMODE_ERRORS`: reception of error frames enabled
bReserved	[out]	Reserved. Value must be initialized with 0.
wBitrate	[out]	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` (see predefined bit rates in *lintype.h* `LIN_BITRATE_`).

6.3.2 LINCAPABILITIES

The structure describes the LIN controller capabilities.

```
typedef struct _LINCAPABILITIES
{
  UINT32 dwFeatures;
  UINT32 dwClockFreq;
  UINT32 dwTscDivisor;
} LINCAPABILITIES, *PLINCAPABILITIES;
```

Member	Dir.	Description
dwFeatures	[out]	Supported features, combination of one or more of the following constants: `LIN_FEATURE_MASTER`: Master mode `LIN_FEATURE_AUTORATE`: automatic bit rate detection `LIN_FEATURE_ERRFRAME`: reception of error messages `LIN_FEATURE_BUSLOAD`: calculation of bus load `LIN_FEATURE_SLEEP`: sleep messages (Master only) `LIN_FEATURE_WAKEUP`: 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 timestamp for LIN messages. Frequency of time stamp counter is calculated by the frequency of the primary timer divided by the value specified here.

6.3.3 LINLINESTATUS

The structure describes the controller status information.

```
typedef struct _LINLINESTATUS
{
  UINT8 bOpMode;
  UINT8 bBusLoad;
  UINT16 wBitrate;
  UINT32 dwStatus;
} LINLINESTATUS, *PLINLINESTATUS;
```

Member	Dir.	Description
bOpMode	[out]	Current operating mode of controller. Value is a logical combination of one or more of the following constants: LIN_OPMODE_SLAVE: Slave mode LIN_OPMODE_MASTER: Master mode LIN_OPMODE_ERRORS: reception of error frames enabled
bBusLoad	[out]	Bus load in the second before the call of the function in percentage (0 to 100). Value shows a state. To monitor the bus load over a time span use appropriate analysis tools. Value is exclusively valid if calculation of bus load is supported by the controller (see *LINCAPABILITIES*).
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_OVERRUN: data overrun in receive buffer occurred LIN_STATUS_ININIT: stopped status, init mode active

6.3.4 LINMONITORSTATUS

The structure describes the message monitor status information.

```
typedef struct _LINMONITORSTATUS
{
  LINLINESTATUS sLineStatus;
  BOOL32 fActivated;
  BOOL32 fRxOverrun;
  UINT8 bRxFifoLoad;
} LINMONITORSTATUS, *PLINMONITORSTATUS;
```

Member	Dir.	Description
sLineStatus	[out]	Current status of LIN controller. For more 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.

6.3.5   LINMSG

The structure describes the LIN message structure.

```
typedef struct _LINMSG
{
  UINT32 dwTime;
  LINMSGINFO uMsgInfo;
  UINT8 abData[8];
} LINMSG, *PLINMSG;
```

Member	Dir.	Description
dwTime	[out]	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 dwClockFreq and dwTscDivisor of structure *LINCAPABILITIES* with the following formula: Resolution[s] = dwTscDivisor / dwClockFreq
uMsgInfo	[out]	Bit field with information about the message. For detailed description of bit field see LINMSGINFO.
abData	[out]	Array for up to 8 data bytes. Number of valid data bytes is determined by the field uMsgInfo.Bits.dlen.

6.3.6   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:

Fields	Dir.	Description
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:

Bit field	Dir.	Description		
Bytes.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.	
		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:	
			Constant	**Meaning**
			LIN_INFO_START	Controller is started. Field *dwTime* contains the relative starting point (normally 0).
			LIN_INFO_STOP	Controller is stopped. Field *dwTime* contains value 0.
			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:	
			Constant	**Meaning**
			LIN_ERROR_BIT	Bit error
			LIN_ERROR_CHKSUM	Check sum error
			LIN_ERROR_PARITY	Parity error of identifier
			LIN_ERROR_SLNORE	Slave does not answer.
			LIN_ERROR_SYNC	Invalid synchronization field.
			LIN_ERROR_NOBUS	No bus activity.
			LIN_ERROR_OTHER	Another, not specified error
			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 undefined.	
		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.	

Bit field	Dir.	Description	
		`LIN_MSGTYPE_ SLEEP`	*Go-to-Sleep* message. The fields *dwTime*, *bPid* and *bDlen* have no significance. 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	[in/out]	Reserved for further extensions. This field is 0.	

This page intentionally left blank