

VCI: C-API for CAN-FD

SOFTWARE DESIGN GUIDE

4.02.0250.20023 1.3 en-US ENGLISH

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.

1	User Guide	5
1.1	Related Documents	5
1.2	Document History	5
1.3	Conventions	6
1.4	Glossary	7
2	System Overview	8
2.1	Subcomponents and Functions of the Programming Interface.....	9
2.2	Programming Examples.....	9
3	Device Management and Device Access.....	10
3.1	Listing Available Devices	11
3.2	Searching Individual Devices	12
3.3	Accessing Devices	13
4	Accessing the Bus	14
4.1	Accessing the CAN Bus	14
4.1.1	Message Channels	15
4.1.2	Control Unit	20
4.1.3	Message Filter	27
4.1.4	Cyclic Transmitting List.....	30
4.2	Accessing the LIN Bus	33
4.2.1	Message Monitors.....	33
4.2.2	Control Unit	36

5	Functions	39
5.1	General Functions	39
5.1.1	vciInitialize	39
5.1.2	vciGetVersion	39
5.1.3	vciFormatErrorA	40
5.1.4	vciFormatErrorW	40
5.1.5	vciDisplayErrorA	41
5.1.6	vciDisplayErrorW	41
5.1.7	vciCreateLuid	42
5.1.8	vciLuidToCharA	42
5.1.9	vciLuidToCharW	43
5.1.10	vciCharToLuidA	43
5.1.11	vciCharToLuidW	44
5.1.12	vciGuidToCharA	44
5.1.13	vciGuidToCharW	45
5.1.14	vciCharToGuidA	45
5.1.15	vciCharToGuidW	46
5.2	Functions for the Device Management	47
5.2.1	Functions for Accessing the Device List	47
5.2.2	Functions for Accessing VCI Devices	52
5.3	Functions for CAN Access	55
5.3.1	Control Unit	55
5.3.2	Message Channel	63
5.3.3	Cyclic Transmit List	78
5.4	Functions for LIN Access	84
5.4.1	Control Unit	84
5.4.2	Message Monitor	88

6	Data Types	95
6.1	VCI-Specific Data Types	95
6.1.1	VCID	95
6.1.2	VCVERSIONINFO	95
6.1.3	VCILICINFO	96
6.1.4	VCIDRIVERINFO	96
6.1.5	VCIDEVICEINFO	97
6.1.6	VCIDEVICECAPS	97
6.1.7	VCIDEVRTINFO	98
6.2	CAN-Specific Data Types	99
6.2.1	CANBTP	99
6.2.2	CANCAPABILITIES2	99
6.2.3	CANINITLINE2	102
6.2.4	CANLINESTATUS2	103
6.2.5	CANCHANSTATUS2	104
6.2.6	CANRTINFO	104
6.2.7	CANSCHEDULERSTATUS2	105
6.2.8	CANMSGINFO	105
6.2.9	CANMSG2	108
6.2.10	CANCYCLICTXMSG2	109
6.3	LIN-Specific Data Types	109
6.3.1	LININITLINE	109
6.3.2	LINCAPABILITIES	110
6.3.3	LINLINESTATUS	111
6.3.4	LINMONITORSTATUS	111
6.3.5	LINMSG	112

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
1.0	January 2018	First version
1.1	September 2018	Minor changes, added information about reception time of a CAN message
1.2	May 2019	Layout and terminology changes
1.3	October 2021	Corrections specifying the bit rate

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
VCID	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 (CAN FD) VCINPL2.DLL is described. The programming interface connects the VCI server and the application programs using predefined components, interfaces and functions.

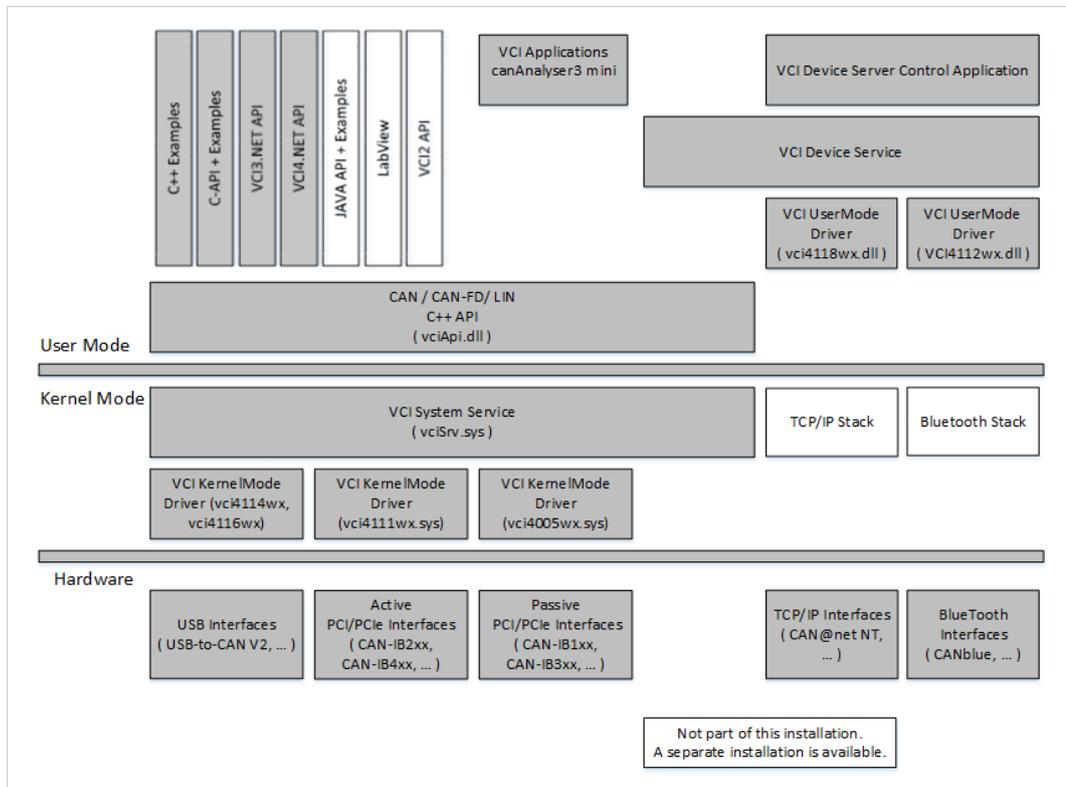


Fig. 1 System structure and components

2.1 Subcomponents and Functions of the Programming Interface

Native VCI programming interfaces (VCINPL2.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	canChannelGetControl	canSchedulerActivate
vciFindDeviceByHwid	canControlInitialize	canChannelInitialize	canSchedulerReset
vciFindDeviceByClass	canControlReset	canChannelGetFilterMode	canSchedulerAddMessage
vciSelectDeviceDlg	canControlStart	canChannelSetFilterMode	canSchedulerRemMessage
vciDeviceOpen	canControlGetFilterMode	canChannelSetAccFilter	canSchedulerStartMessage
vciDeviceOpenDlg	canControlSetFilterMode	canChannelAddFilterIds	canSchedulerStopMessage
vciDeviceClose	canControlSetAccFilter	canChannelRemFilterIds	
vciDeviceGetInfo	canControlAddFilterIds	canChannelActivate	
vciDeviceGetCaps	canControlRemFilterIds	canChannelPeekMessage	
		canChannelPostMessage	
		canChannelWaitRxEvent	
		canChannelWaitTxEvent	
		canChannelReadMessage	
		canChannelSendMessage	
	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\Npl2`.

3 Device Management and Device Access

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

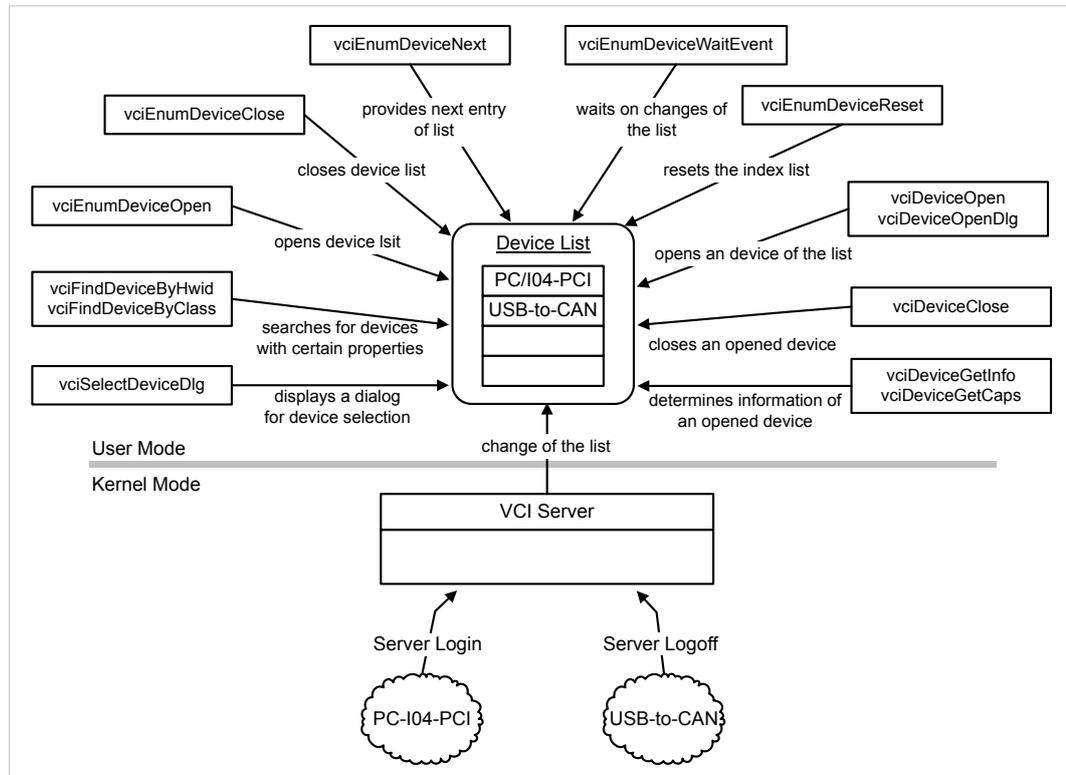


Fig. 2 Device management components

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
<i>VciObjectId</i>	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.
<i>DeviceClass</i>	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.
<i>UniqueHardwareId</i>	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 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` and specify 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.

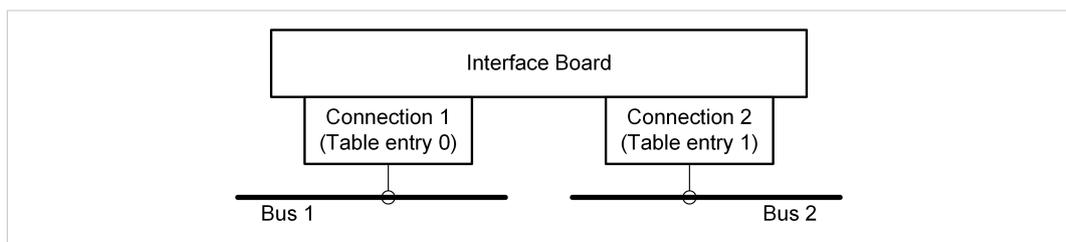


Fig. 3 Interface with two bus connections

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

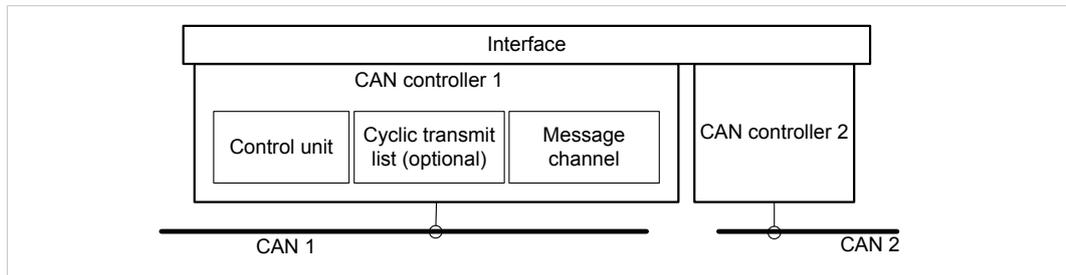


Fig. 4 Components CAN controller and interface IDs

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. 30](#))

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.

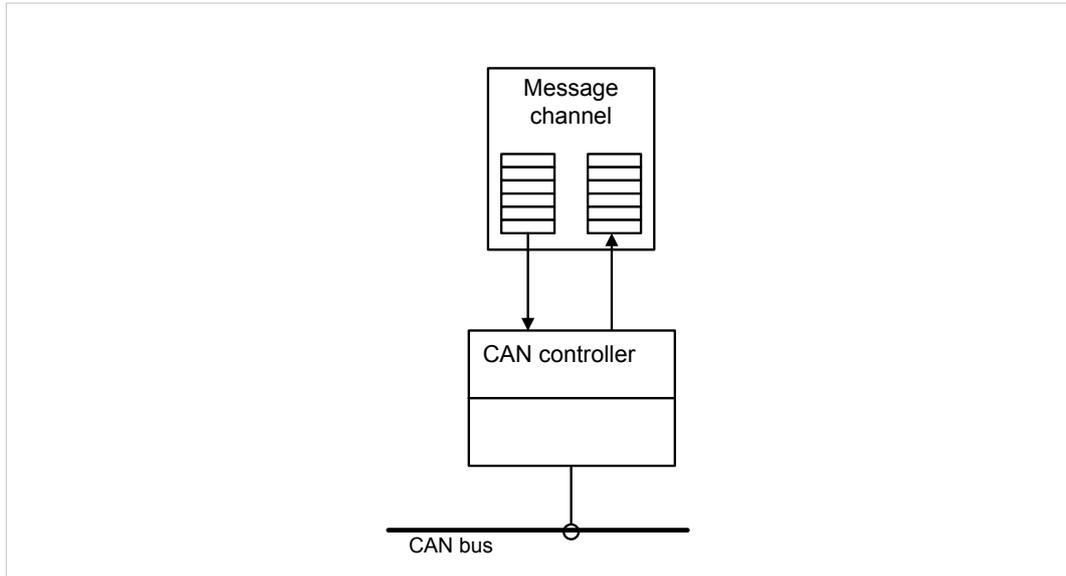


Fig. 5 Exclusive use of a message channel

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.

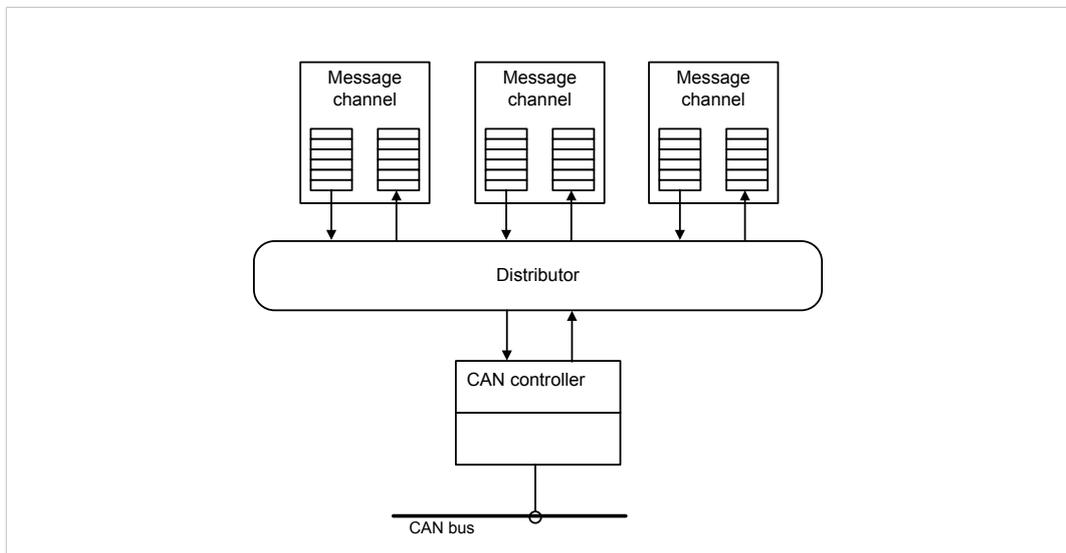


Fig. 6 CAN message distributor: possible configuration with three channels

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.



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` value `TRUE`.
- ▶ To deactivate the channel enter in parameter `fEnable` 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;
    CANMSG2 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 [CANMSG2](#). 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 [CANMSG2](#)).

After an overrun of the counter the timer is reset.

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

- $T_{rx} = dwTime \text{ of message} - dwTime \text{ of } CAN_INFO_START \text{ (start of controller)}$
Field *dwTime* of the message see [CANMSG2](#)
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 [CANCAPABILITIES2](#)

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 CAN 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;
```

```
CANMSG2 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 [CANCAPABILITIES2](#) support the possibility to transmit messages delayed, with a latency between two consecutive messages.

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

- ▶ In field *dwTime* of the structure [CANMSG2](#) 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 [CANCAPABILITIES2](#), the value in *dwTime* must not exceed the value in *dwMaxDtxTicks*.

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

- $\text{Resolution [s]} = \text{dwDtxDivisor} / \text{dwClockFreq}$

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:

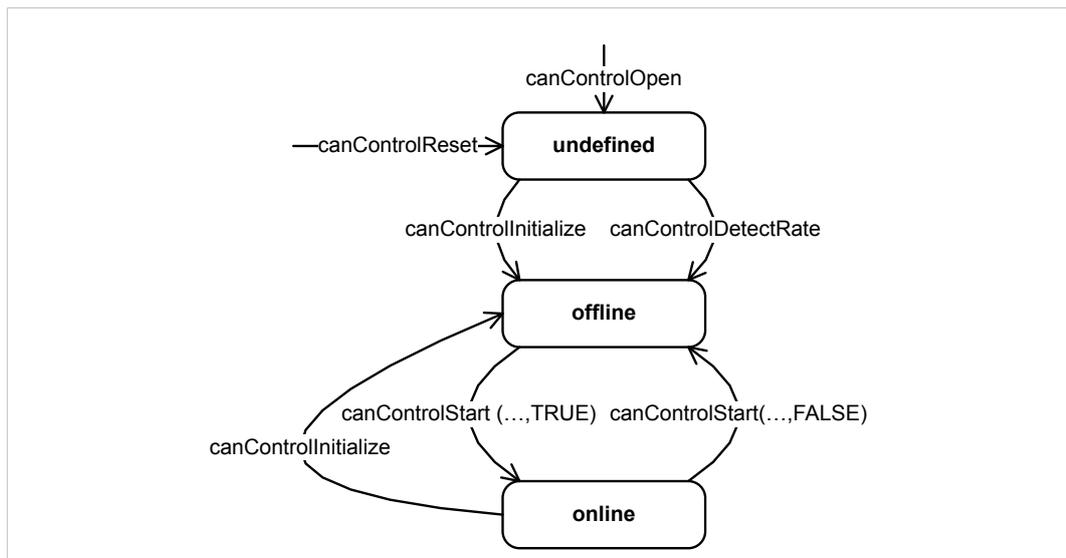


Fig. 7 Controller states

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 `bOpMode`.
- ▶ With `canControlInitialize` set the bit rate and the sampling time in parameters `pBtpSDR` and `pBtpFDR` (see [Specifying the Bit Rate, p. 22](#) for more information).
- ▶ To detect the bit rate 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.

Specifying the Bit Rate

- ▶ In `canControlInitialize` specify the bit rate with the fields `pBtpSDR` and `pBtpFDR`.

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

Time Periods

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

If the bit `CAN_BTMODE_RAW` in `dwMode` is set, all other fields contain controller specific values (see [Mode CAN_BTMODE_RAW](#), p. 25).

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

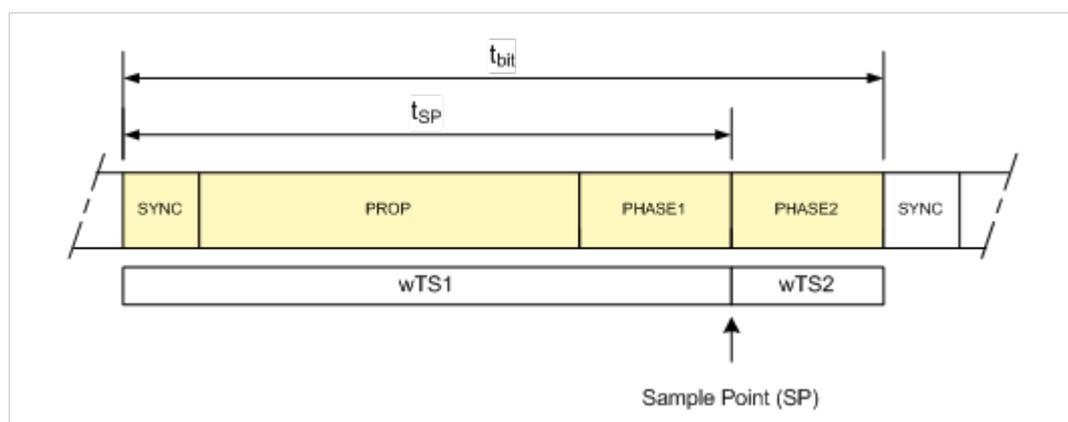


Fig. 8 Segmentation of a bit in different time periods

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

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

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

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

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

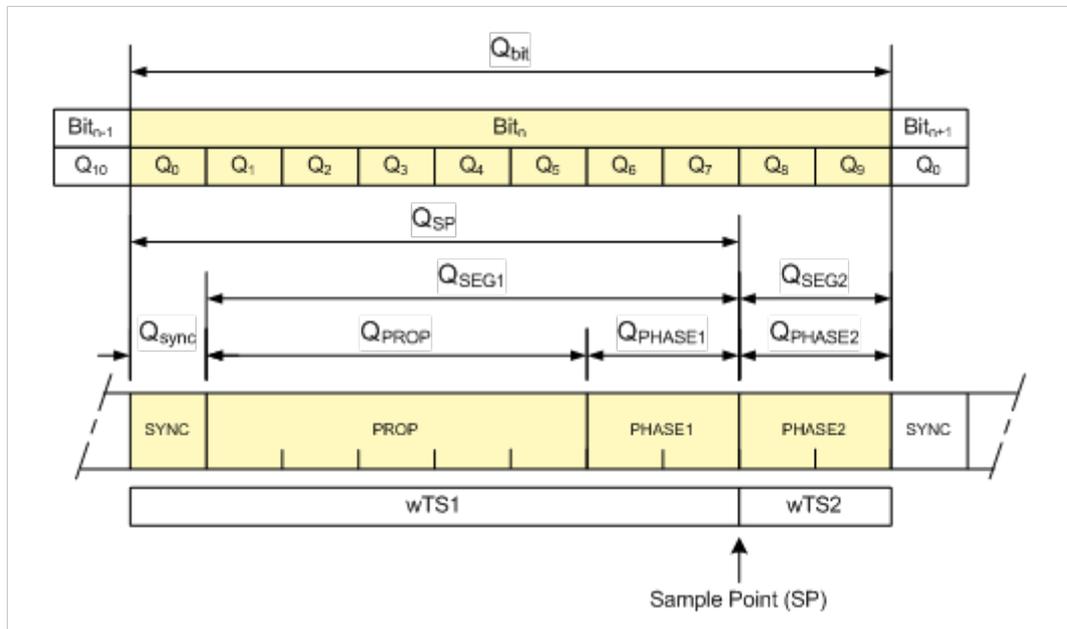


Fig. 9 Segmentation of a bit in time quanta and segments

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

Segments

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

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

Post scoring

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

$$t_{SJW} = t_Q * wSJW$$

Synchronization Jump Width

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

Condition 1

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

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

Condition 2

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

Time Quanta

Observe the following when specifying:

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

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

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

In example [Segmentation of a bit in time quanta and segments, p. 23](#) with $wTS1=8$, $wTS2=2$ and $Q_{bit}=10$ the sampling point is 80 %. The resolution of a time quantum is 1/10 resp. 10 % of a bit time. If the value 1 is specified for $wSJW$ the sampling point of a phase correction is shifted about ± 10 % of a bit time. Higher values than 1 are not allowed for $wSJW$ in this example, because sampling errors could occur.

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

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

The resolution of a time quantum theoretically can be shortened down to $1/131070 \approx 7.63 \cdot 10^{-6}$ resp. 7.63 ppm. As the values for the individual segments have to be converted to the hardware specific register, the limits are higher. Regarding the SJA1000 with 16 MHz clock frequency the maximum possible value for Q_{bit} is 25 (1+16+8) and therefore the minimum possible resolution is 1/25 resp. 4 % of a bit time. With higher bit times the number of time quanta is reduced and is for 1 Mbit only 8, that results in a resolution of 1/8 resp. 12.5 % of a bit time.

- ▶ To get information about the value ranges of the individual segments supported by the hardware call function `canControlGetCaps`.
 - Fields `sSdrRangeMin`, `sSdrRangeMax` resp. `sFdrRangeMin` and `sFdrRangeMax` of structure [CANCAPABILITIES2](#) indicated with calling of the function contain hardware specific minimum and maximum values.

Mode CAN_BTMODE_RAW

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

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

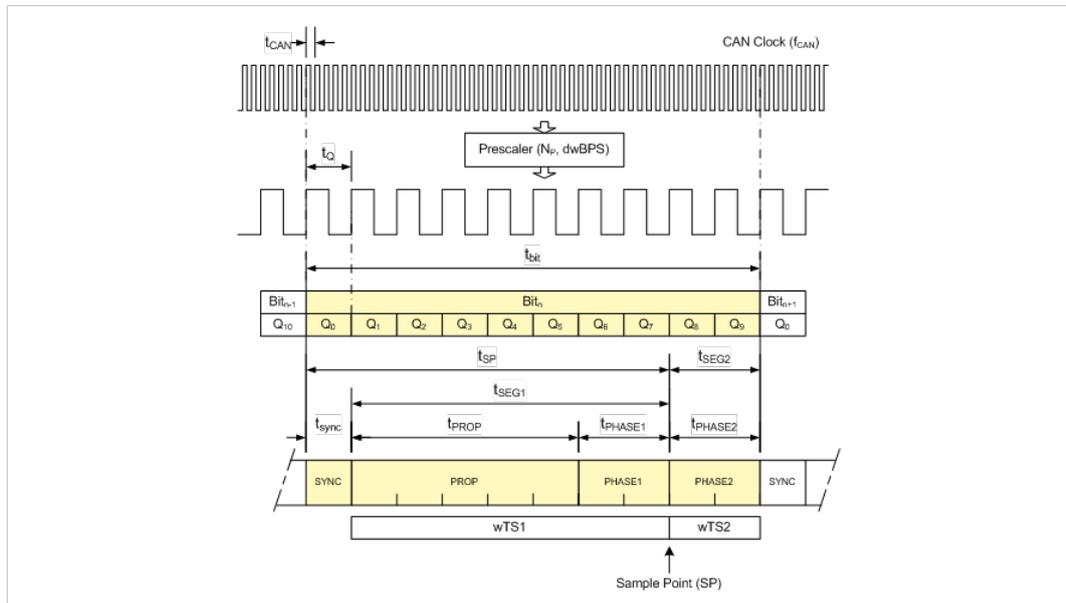


Fig. 10 Clock generator for the bit processor in the CAN controller

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

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

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

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

The bit rate f_{bit} is calculated by:

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

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

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

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

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

If the value has fallen below the minimal amount of time quanta there is no solution for the demanded bit rate. In the other case with the found values for N_P and Q_{bit} the values for w_{TS1} , w_{TS2} and w_{SJW} can be determined in the following way:

- Calculate the time of a time quantum:

$$t_Q = N_P / f_{CAN}$$

- ▶ Determine the amount of time quanta $Q_{S\text{JW}}$ required for the post scoring with [Condition 1](#) and [Condition 2](#).



The value is dependent on the oscillation tolerance ΔF . The oscillation tolerance of Ixxat CAN interfaces is normally smaller than 0.1 % but in this case the greatest oscillation tolerance of all nodes existing in the network must be considered.

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

$$Q_{\text{PROP}} = \text{round_up}(t_{\text{PROP}} / t_{\text{Q}})$$

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

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

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

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

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

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

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

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

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.

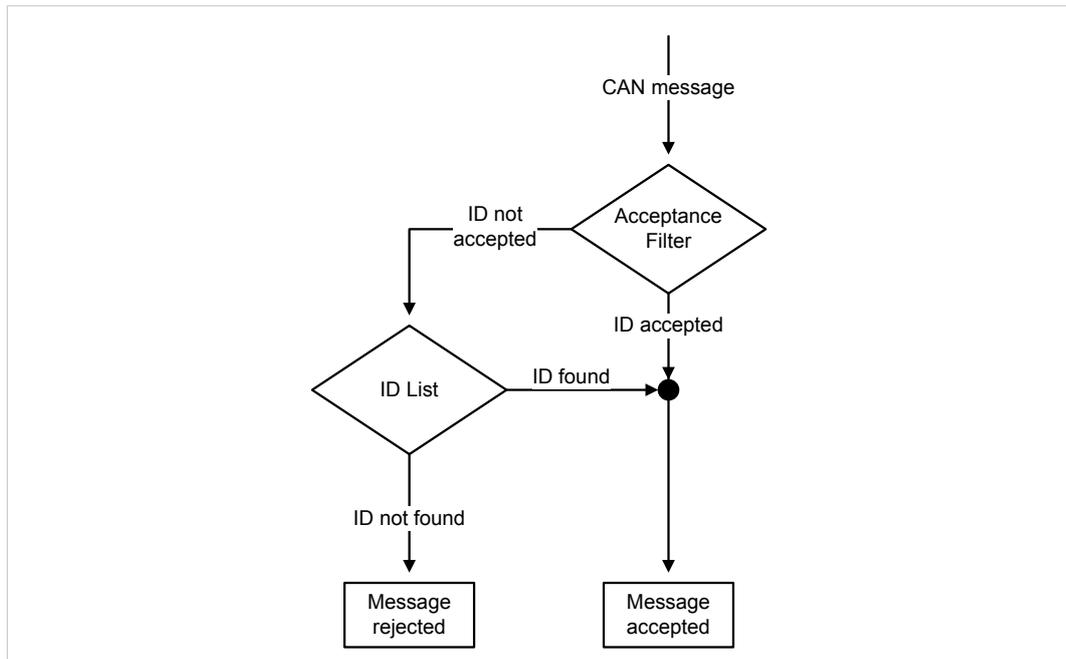


Fig. 11 Filtering mechanism

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:

<i>dwCode</i>	001 0000 0000 0
<i>dwMask</i>	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 0xFFFF (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 *CANCYCLICTXMSG2* 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 *CANCYCLICTXMSG2*.
- ▶ Make sure that the specified value is higher than 0 but less than or equal the value in field *dwCmsMaxTicks* of the structure *CANCAPABILITIES2*.
- ▶ 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:

$$t_z [s] = (dwCmsDivisor / dwClockFreq)$$

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).

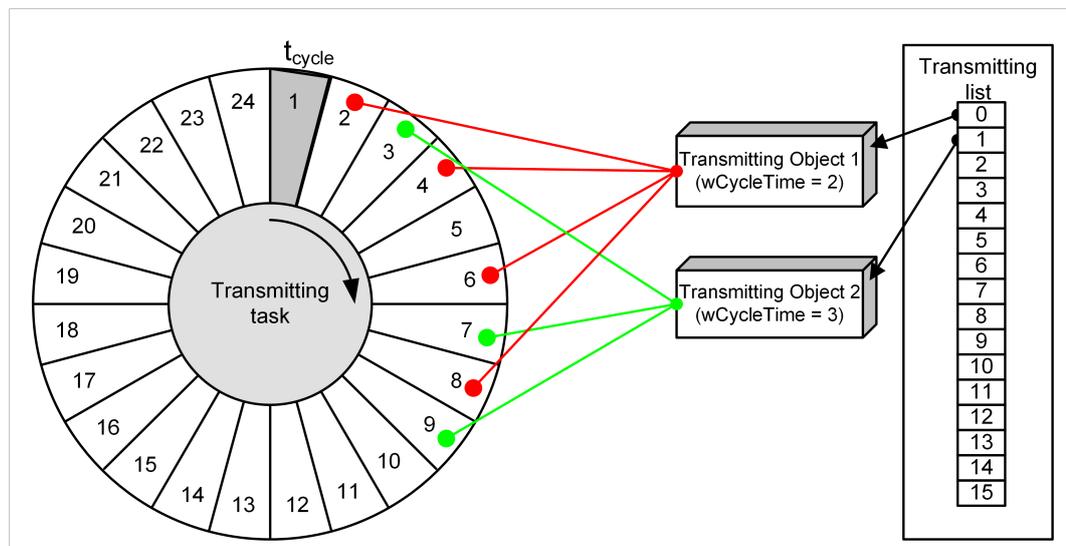


Fig. 12 Transmitting task of the cyclic transmitting list with 24 time frames

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 *blncrMode* of structure *CANCYCLICTXMSG2* determines if certain parts of a message are automatically incremented after transmitting or if they remain unmodified.

If in *blncrMode* *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 *CANCYCLICTXMSG2* determines the starting position of the data value.

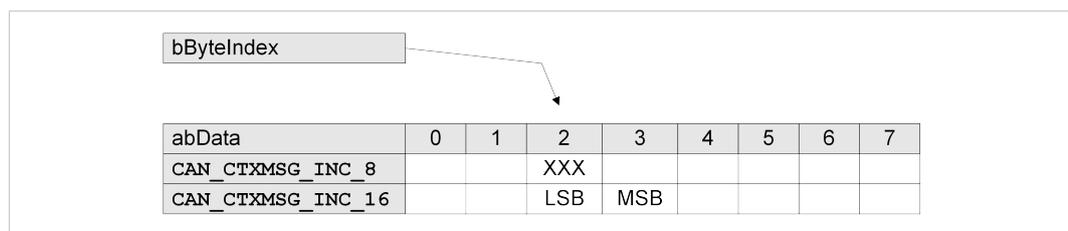


Fig. 13 Auto increment of data fields

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 `CANSCHEDULERSTATUS2` 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

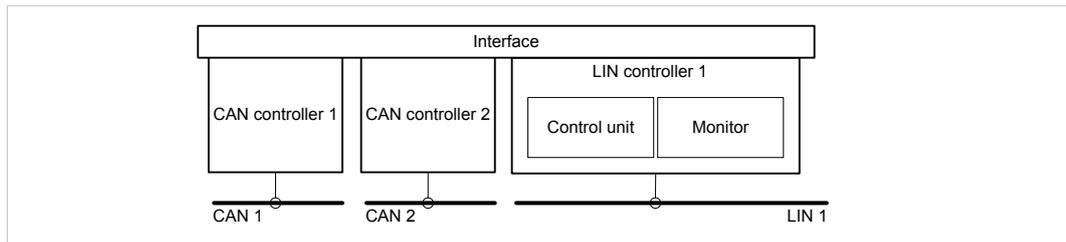


Fig. 14 Components LIN controller

Each LIN connections consists of the following components:

- control unit (see [Control Unit, p. 36](#))
- one or more message monitors (see [Message Monitors, p. 33](#))

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.

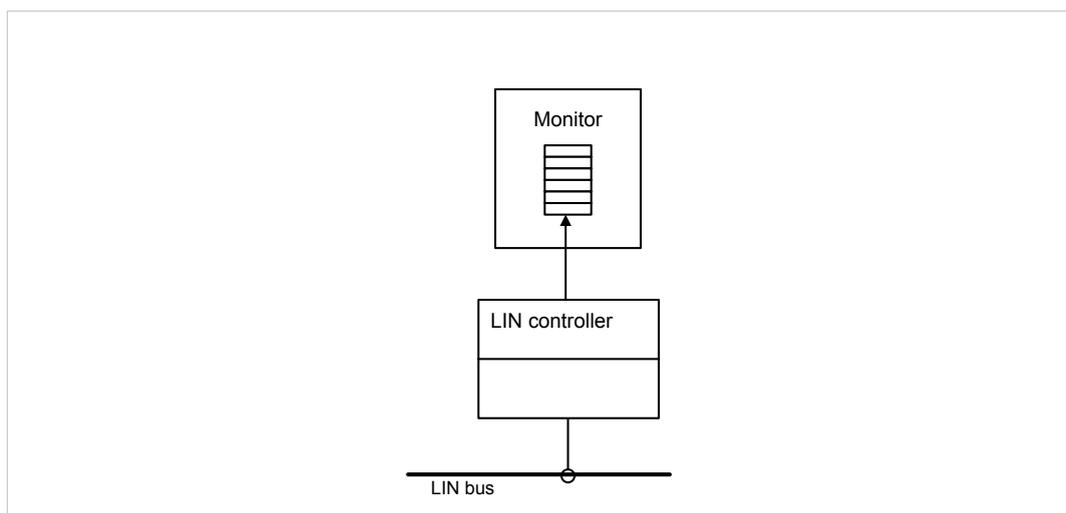


Fig. 15 Exclusive usage

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.

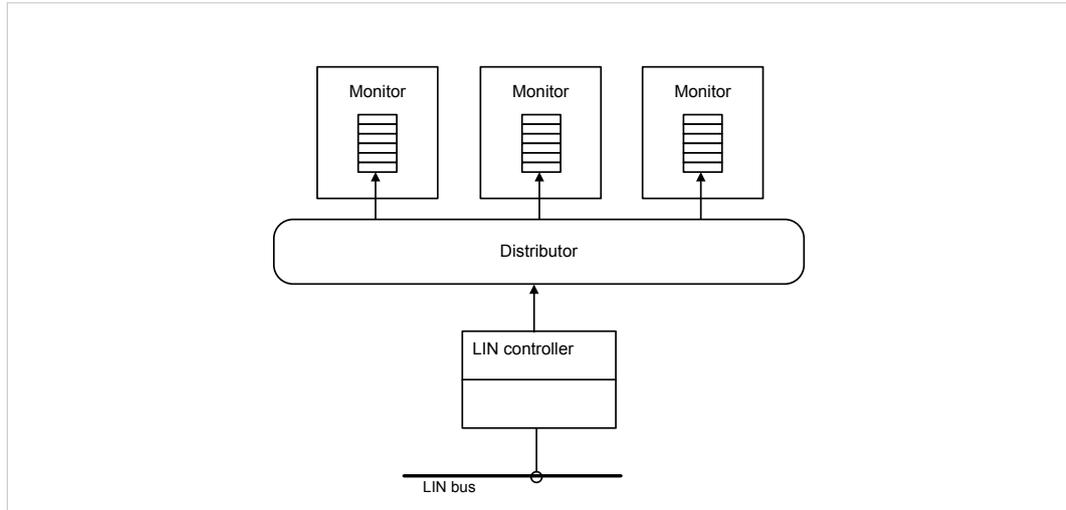


Fig. 16 Non-exclusive usage (with distributor)

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 more information about LIN controllers see chapter [Control Unit, p. 36](#).

- ▶ 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. 34](#)).

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.

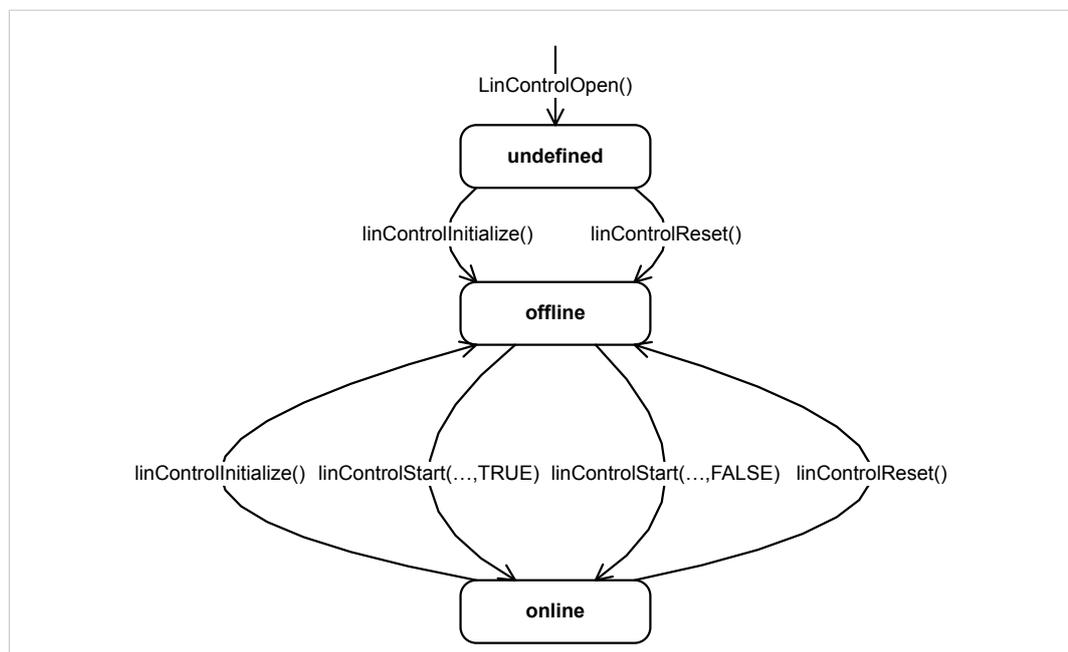


Fig. 17 LIN controller states

Initializing the Controller

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

- ▶ To leave the non-initialized 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 that are 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.

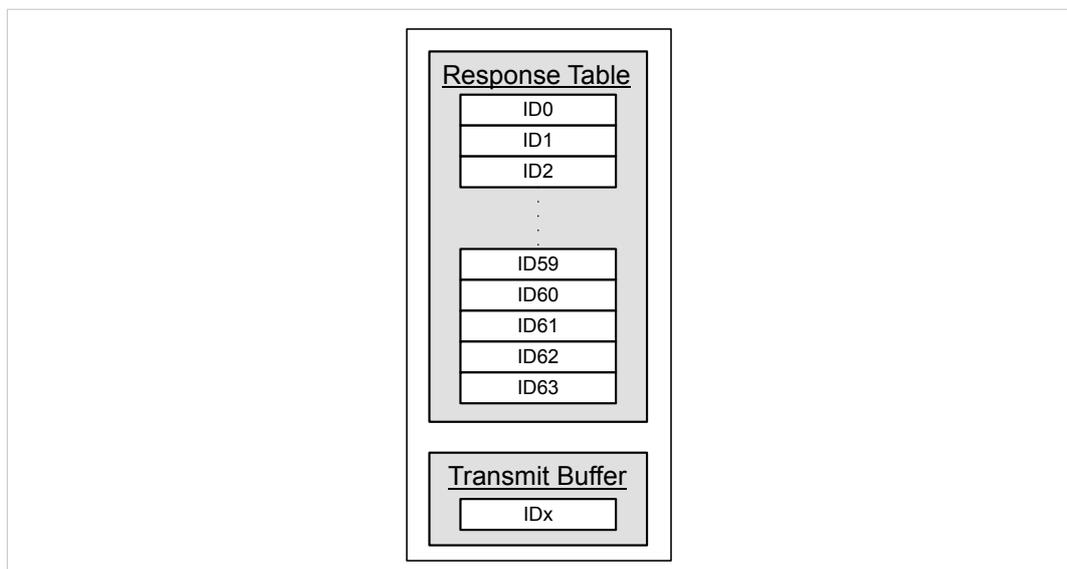


Fig. 18 Internal structure of a control unit

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 *hLinCtrl* 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 *hLinCtrl* 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 more 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 more information see [LINMONITORSTATUS](#)).

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 VCINPL2 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,
    PUINT32 pdwRevNumber,
    PUINT32 pdwBuildNumber
);
```

Parameter

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

5.1.3 vciFormatErrorA

Formats VCI error code as text.

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

Parameter

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

5.1.4 vciFormatErrorW

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

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

Parameter

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

Displays VCI error code in message box.

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

Parameter

Parameter	Dir.	Description
<i>hwndParent</i>	[in]	Handle of the higher order window. If value ZERO is specified here, the message window has no higher order window.
<i>pszCaption</i>	[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.
<i>hrError</i>	[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

5.1.6 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
<i>hwndParent</i>	[in]	Handle of the higher order window. If value ZERO is specified here, the message window has no higher order window.
<i>pszCaption</i>	[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.
<i>hrError</i>	[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 vciCreateLuid

Creates a locally unique VCI ID.

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

Parameter

Parameter	Dir.	Description
<i>pVciid</i>	[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

5.1.8 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
<i>rVciid</i>	[in]	Reference to the locally unique VCI ID to be converted into a character string
<i>pszLuid</i>	[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.
<i>cbSize</i>	[in]	Size of the buffer specified in <i>pszLuid</i> in bytes

Return Value

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

5.1.9 vciLuidToCharW

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

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

Parameter

Parameter	Dir.	Description
<i>rVciid</i>	[in]	Reference to the locally unique VCI ID to be converted into a character string
<i>pszLuid</i>	[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.
<i>cbSize</i>	[in]	Size of the buffer specified in <i>pszLuid</i> in bytes

Return Value

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

5.1.10 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
<i>pszLuid</i>	[in]	Pointer to the 0-terminated character string to be converted
<i>pVciid</i>	[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 <i>pszLuid</i> or <i>pVciid</i> points to an invalid buffer.
VCI_E_FAIL	Character string specified in <i>pszLuid</i> can not be converted into a valid ID.

5.1.11 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
<i>pwszLuid</i>	[in]	Pointer to the 0-terminated wide character string to be converted
<i>pVciid</i>	[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 <i>pszLuid</i> or <i>pVciid</i> points to an invalid buffer.
VCI_E_FAIL	Character string specified in <i>pszLuid</i> can not be converted into a valid ID.

5.1.12 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
<i>rGuid</i>	[in]	Reference to the globally unique ID that is to be converted into a character string
<i>pszGuid</i>	[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.
<i>cbSize</i>	[in]	Size of the in <i>pszGuid</i> specified buffer in Bytes.

Return Value

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

5.1.13 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
<i>rGuid</i>	[in]	Reference to the globally unique ID that is to be converted into a character string.
<i>pwszGuid</i>	[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.
<i>cbSize</i>	[in]	Size of the in <i>pwszGuid</i> specified buffer in Bytes

Return Value

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

5.1.14 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
<i>pszGuid</i>	[in]	Pointer to the 0-terminated character string to be converted
<i>pGuid</i>	[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 <i>pszGuid</i> or <i>pGuid</i> points to an invalid buffer.
VCI_E_FAIL	Character string specified in <i>pszGuid</i> can not be converted into a valid ID.

5.1.15 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
<i>pwszGuid</i>	[in]	Pointer to the 0-terminated character string to be converted
<i>pGuid</i>	[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 <i>pwszGuid</i> or <i>pGuid</i> points to an invalid buffer.
VCI_E_FAIL	Character string specified in <i>pszGuid</i> 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
<i>hEnum</i>	[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
<i>hEnum</i>	[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
<i>hEnum</i>	[in]	Handle to the opened device list
<i>pInfo</i>	[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	List 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
<i>hEnum</i>	[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
<i>hEnum</i>	[in]	Handle of the opened device list
<i>dwTimeout</i>	[in]	Specifies the timeout interval, in milliseconds. The function returns if the interval elapses, even if the device list has not changed. If <i>dwTimeout</i> is zero, the function tests the state of the device list and returns immediately. If <i>dwTimeout</i> is INFINITE (0xFFFFFFFF), the timeout interval of the function 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 the <i>dwTimeout</i> parameter 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
<i>rHwid</i>	[in]	Reference to the unique hardware ID of the adapter to search for
<i>pVciid</i>	[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,
    PVICEID pVciid
);
```

Parameter

Parameter	Dir.	Description
<i>rClass</i>	[in]	Reference to the device class of the adapter to search for
<i>dwInst</i>	[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.
<i>pVciid</i>	[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,
    PVICEID pVciid
);
```

Parameter

Parameter	Dir.	Description
<i>hwndParent</i>	[in]	Handle of the higher order window. If value ZERO is specified here, the dialog window has no higher order window.
<i>pVciid</i>	[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
<i>rVciid</i>	[in]	Device ID of the adapter to be opened
<i>phDevice</i>	[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
<i>hwndParent</i>	[in]	Handle of the higher order window. If value ZERO is specified here, the dialog window has no higher order window.
<i>phDevice</i>	[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
<i>hDevice</i>	[in]	Handle of the adapter to be closed. The specified handle must come from a call of one of the functions vciDeviceOpen 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
<i>hDevice</i>	[in]	Handle of the opened adapter
<i>pInfo</i>	[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,
    PVCIDVICECAPS  pCaps
);
```

Parameter

Parameter	Dir.	Description
<i>hDevice</i>	[in]	Handle of the opened adapter
<i>pCaps</i>	[out]	Address of a structure of type VCIDVICECAPS . 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
<i>hDevice</i>	[in]	Handle of the fieldbus adapter
<i>dwCanNo</i>	[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.
<i>phCanCtl</i>	[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
<i>hCanCtl</i>	[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,
    PCANCAPABILITIES2 pCanCaps
);
```

Parameter

Parameter	Dir.	Description
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>pCanCaps</i>	[out]	Pointer to a structure of type CANCAPABILITIES2 . 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,
    PCANLINESTATUS2 pStatus
);
```

Parameter

Parameter	Dir.	Description
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>pStatus</i>	[out]	Pointer to a structure of type CANLINESTATUS2 . 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,
    UINT8 bOpMode,
    UINT8 bExMode,
    UINT16 wTimeout,
    UINT32 dwCount,
    PCANBTP paBtpSDR,
    PCANBTP paBtpFDR,
    PUINT32 pdwIndex
);
```

Parameter

Parameter	Dir.	Description
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>bOpMode</i>	[in]	Operating mode of the CAN controller
<i>bExMode</i>	[in]	Extended operating mode of the CAN controller
<i>wTimeout</i>	[in]	Maximum waiting time in milliseconds between two messages on the bus.
<i>dwCount</i>	[in]	Number of elements in the bit timing tables <i>paBtpSDR</i> and <i>paBtpFDR</i>
<i>paBtpSDR</i>	[in]	Pointer to a table with the arbitration bit rates. The table must contain at least <i>dwCount</i> elements.
<i>paBtpFDR</i>	[in]	Pointer to a table with the fast bit rates. The table must contain at least <i>dwCount</i> elements.
<i>pdwIndex</i>	[out]	Pointer to a variable of type UINT32. 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 <i>wTimeout</i>

Remark

More information on the bus timing values in the tables *paBtpSDR* and *paBtpFDR* is given in 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 *pdwIndex* 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 bOpMode,
    UINT8 bExMode,
    UINT8 bSFMode,
    UINT8 bEFMode,
    UINT32 dwSFIds,
    UINT32 dwEFIds,
    PCANBTP pBtpSDR,
    PCANBTP pBtpFDR
);
```

Parameter

Parameter	Dir.	Description
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>bOpMode</i>	[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
<i>bExMode</i>	[in]	Extended operating mode of the CAN controller CAN_EXMODE_DISABLED: no extended operation CAN_EXMODE_EXTDATA: extended data length CAN_EXMODE_FASTDATA: fast data bit rate CAN_EXMODE_NONISO: non ISO conform frames
<i>bSFMode</i>	[in]	Operating mode of 11-bit ID filter
<i>bEFMode</i>	[in]	Operating mode of 29-bit ID filter
<i>dwSFIds</i>	[in]	Size of 11-bit ID filter
<i>dwEFIds</i>	[in]	Size of 29-bit ID filter
<i>pBtpSDR</i>	[in]	Pointer to the timing parameters used for standard (nominal) bit rate (see Remarks)
<i>pBtpFDR</i>	[in]	Pointer to the timing parameters used for fast data bit rate. This parameter can be NULL if <i>bExMode</i> is not set to CAN_EXMODE_FASTDATA.

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 *pBtpSDR* and *pBtpFDR* 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
<i>hCanCtl</i>	[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
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>fStart</i>	[in]	Value <code>TRUE</code> starts and value <code>FALSE</code> stops the CAN controller.

Return Value

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

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.

canControlGetFilterMode

Retrieves the current operating mode for the specified CAN message filter.

```
HRESULT EXTERN_C canControlGetFilterMode (
    HANDLE hCanCtl,
    BOOL   fExtend,
    PUINT8 pbMode
);
```

Parameter

Parameter	Dir.	Description
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>fExtend</i>	[in]	Filter selection. If this parameter is set to <code>TRUE</code> , the function selects the 29-bit filter. If this parameter is set to <code>FALSE</code> , the function selects the 11-bit filter.
<i>pbMode</i>	[out]	Pointer to a buffer where the function stores the current operating mode of the specified filter (see canControlSetFilterMode).

Return Value

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

canControlSetFilterMode

Sets the operating mode for the specified CAN message filter. The function clears the current filter settings, if the mode changes. The function can be called only if the controller is in init mode.

```
HRESULT EXTERN_C canControlSetFilterMode (
    HANDLE hCanCtl,
    BOOL   fExtend,
    UINT8  bMode,
    PUINT8 pbPrev
);
```

Parameter

Parameter	Dir.	Description
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>fExtend</i>	[in]	Filter selection. If this parameter is set to <code>TRUE</code> , the function selects the 29-bit filter. If this parameter is set to <code>FALSE</code> , the function selects the 11-bit filter.
<i>bMode</i>	[in]	Operating mode (<code>CAN_FILTER_STD</code> for 11-bit standard filter, <code>CAN_FILTER_EXT</code> for 29-bit extended filter)
<i>pbPrev</i>	[out]	Optional pointer to a buffer where the function stores the previous operating mode of the specified filter. This parameter can be <code>NULL</code> .

Return Value

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

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

Return Value

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

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
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>fExtend</i>	[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.
<i>dwCode</i>	[in]	Bit sample of the identifier(s) to be identified including RTR-bit
<i>dwMask</i>	[in]	Bit sample of the relevant bits in <i>dwCode</i> . If a bit has the value 0 in <i>dwMask</i> , the corresponding bit in <i>dwCode</i> 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
<i>hCanCtl</i>	[in]	Handle of the opened CAN controller
<i>fExtend</i>	[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.
<i>dwCode</i>	[in]	Bit sample of the identifier(s) to be removed including RTR-bit
<i>dwMask</i>	[in]	Bit sample of the relevant bits in <i>dwCode</i> . If a bit has the value 0 in <i>dwMask</i> , the corresponding bit in <i>dwCode</i> 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
<i>hDevice</i>	[in]	Handle of the fieldbus adapter
<i>dwCanNo</i>	[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.
<i>fExclusive</i>	[in]	Defines whether the connection is used exclusively for the channel to be opened. If value <code>TRUE</code> is specified here, the CAN connection is used exclusively for the new message channel. With value <code>FALSE</code> , more than one message channel can be opened for the CAN connection.
<i>phCanChn</i>	[out]	Pointer to a variable of type <code>HANDLE</code> . 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 <code>ZERO</code> .

Return Value

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

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* instead of 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 [canChannelClose](#).

canChannelClose

Closes an opened message channel.

```
HRESULT_EXTERN_C canChannelClose (
    HANDLE hCanChn
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[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,
    PCANCAPABILITIES2 pCanCaps
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>pCanCaps</i>	[out]	Pointer to a structure of type CANCAPABILITIES2 . 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,
    PCANCHANSTATUS2 pStatus
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>pStatus</i>	[out]	Pointer to a structure of type CANCHANSTATUS2 . 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

canChannelGetControl

Tries to acquire exclusive access to the CAN controller.

```
HRESULT EXTERN_C canChannelGetControl (
    HANDLE  hCanChn,
    PHANDLE phCanCtl
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>phCanCtl</i>	[out]	Points to a variable where the function stores the handle to the CAN controller. This variable is set to NULL if the CAN controller is currently in use.

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,
    UINT32 dwFilterSize,
    UINT8 bFilterMode
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>wRxFifoSize</i>	[in]	Size of the receive buffer in number of CAN messages
<i>wRxThreshold</i>	[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.
<i>wTxFifoSize</i>	[in]	Size of the transmit buffer in number of CAN messages
<i>wTxThreshold</i>	[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.
<i>dwFilterSize</i>	[in]	Number of CAN message IDs supported by the extended ID filter. The standard ID filter support always all of the possible 2048 IDs. If this parameter is set to 0 CAN message filtering is disabled.
<i>bFilterMode</i>	[in]	Initial mode of the CAN message filter. This can be one of the following values: CAN_FILTER_LOCK: lock filter CAN_FILTER_PASS: bypass filter CAN_FILTER_INCL: inclusive filtering CAN_FILTER_EXCL: exclusive filtering The filter mode can be combined with CAN_FILTER_SRRA to pass all self reception messages sent from other channels which are also connected to the same CAN controller as this channel.

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.

canChannelGetFilterMode

Retrieves the current operating mode for the specified CAN message filter.

```
HRESULT EXTERN_C canChannelGetFilterMode (
    HANDLE hCanChn,
    BOOL fExtend,
    PUINT8 pbMode
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>fExtend</i>	[in]	Filter selection. If this parameter is set to <code>TRUE</code> , the function selects the 29-bit filter. If this parameter is set to <code>FALSE</code> , the function selects the 11-bit filter.
<i>pbMode</i>	[out]	Pointer to a buffer where the function stores the current operating mode of the specified filter (see canChannelSetFilterMode).

Return Value

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

canChannelSetFilterMode

Sets the operating mode for the specified CAN message filter. The function clears the current filter settings if the mode changes. The function can only be called if the channel is currently disabled.

```
HRESULT EXTERN_C canChannelSetFilterMode (
    HANDLE hCanChn,
    BOOL fExtend,
    UINT8 bMode,
    PUINT8 pbPrev
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>fExtend</i>	[in]	Filter selection. If this parameter is set to <code>TRUE</code> , the function selects the 29-bit filter. If this parameter is set to <code>FALSE</code> , the function selects the 11-bit filter.
<i>bMode</i>	[in]	Operating mode
<i>pbPrev</i>	[out]	Optional pointer to a buffer where the function stores the previous operating mode of the specified filter. This parameter can be <code>NULL</code> .

Return Value

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

canChannelSetAccFilter

Sets the 11-bit or 29-bit acceptance filter of the message channel.

```
HRESULT EXTERN_C canChannelSetAccFilter (
    HANDLE hCanChn,
    BOOL   fExtend,
    UINT32 dwCode,
    UINT32 dwMask
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>fExtend</i>	[in]	Filter selection. If this parameter is set to <code>TRUE</code> , the function sets the 29-bit filter acceptance filter. If this parameter is set to <code>FALSE</code> , the function sets the 11-bit acceptance filter.
<i>dwCode</i>	[in]	Acceptance code inclusive RTR bit
<i>dwMask</i>	[in]	Acceptance mask that specifies the relevant bits within <i>dwCode</i> . Relevant bits are specified by a 1 in the corresponding bit position, non relevant bits are 0.

Return Value

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

canChannelAddFilterIds

Registers the specified CAN message identifier or group of identifiers at the specified filter list.

```
HRESULT EXTERN_C canChannelAddFilterIds (
    HANDLE hCanChn,
    BOOL   fExtend,
    UINT32 dwCode,
    UINT32 dwMask
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>fExtend</i>	[in]	Filter selection. If this parameter is set to <code>TRUE</code> , the function adds the IDs to the 29-bit filter list. If this parameter is set to <code>FALSE</code> , the function adds the IDs to the 11-bit filter list.
<i>dwCode</i>	[in]	Message identifier (inclusive RTR) to add to the filter list
<i>dwMask</i>	[in]	Mask that specifies the relevant bits within <i>dwCode</i> . Relevant bits are specified by a 1 in the corresponding bit position, non relevant bits are 0.

Return Value

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

Remark

Depending on the current setting of the filter mode (`CAN_FILTER_INCL` or `CAN_FILTER_EXCL`) IDs registered within the filter list are either accepted or rejected. The function can only be called if the channel is currently disabled.

canChannelRemFilterIds

Removes the specified CAN message identifier or group of identifiers from the specified filter list. The function can only be called if the channel is currently disabled.

```
HRESULT EXTERN_C canChannelRemFilterIds (
    HANDLE hCanChn,
    BOOL fExtend,
    UINT32 dwCode,
    UINT32 dwMask
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>fExtend</i>	[in]	Filter selection. If this parameter is set to <code>TRUE</code> , the function removes the IDs from the 29-bit filter list. If this parameter is set to <code>FALSE</code> , the function removes the IDs from the 11-bit filter list.
<i>dwCode</i>	[in]	Message identifier (inclusive RTR) to remove from the filter list
<i>dwMask</i>	[in]	Mask that specifies the relevant bits within <i>dwCode</i> . Relevant bits are specified by a 1 in the corresponding bit position, non relevant bits are 0.

Return Value

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

canChannelActivate

Activates or deactivates a message channel.

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

Parameter

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

Return Value

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

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 [canControlStart](#) 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,
    PCANMSG2 pCanMsg
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>pCanMsg</i>	[out]	Pointer to a CANMSG2 structure where the function stores the retrieved CAN message. If this parameter is set to <code>NULL</code> , the function simply removes the next CAN message from the receive FIFO.

Return Value

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

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
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>paCanMsg</i>	[out]	Array of buffers where the function stores the retrieved CAN messages. If this parameter is set NULL, the function simply removes the specified number of CAN messages from the receive FIFO.
<i>dwCount</i>	[in]	Number of messages available in buffer
<i>pdwDone</i>	[out]	Pointer to a variable where the function stores the number of CAN 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 CAN message available
!=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,
    PCANMSG2 pCanMsg
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>pCanMsg</i>	[in]	Pointer to an initialized structure of type CANMSG2 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,
    PCANMSG2 paCanMsg,
    UINT32 dwCount,
    PUINT32 pdwDone
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>paCanMsg</i>	[in]	Pointer to array of transmit messages
<i>dwCount</i>	[in]	Number of valid messages in buffer
<i>pdwDone</i>	[out]	Pointer to a variable where the function stores the number of CAN messages written. This parameter 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
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>dwTimeout</i>	[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 parameter *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
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>dwTimeout</i>	[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 parameter is zero, the function tests if a message can be written and returns immediately. If this parameter is INFINITE (0xFFFFFFFF), the timeout interval of the function never elapses.

Return Value

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

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 parameter *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,
    PCANMSG2  pCanMsg
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>dwTimeout</i>	[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.
<i>pCanMsg</i>	[out]	Pointer to a CANMSG2 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
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,
    PCANMSG2  paCanMsg,
    UINT32    dwCount,
    PUINT32   pdwDone
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>dwTimeout</i>	[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.
<i>paCanMsg</i>	[out]	Pointer to message buffer
<i>dwCount</i>	[in]	Number of slots in message buffer
<i>pdwDone</i>	[out]	Number of received CAN messages

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_RXQUEUE_EMPTY	Currently no CAN message available
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,
    PCANMSG2  pCanMsg
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>dwTimeout</i>	[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.
<i>pCanMsg</i>	[in]	Pointer to an initialized structure of type CANMSG2 with the CAN message to be transmitted.

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_TXQUEUE_FULL	<i>dwTimeout</i> 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 <code>VciFormatError</code>

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,
    PCANMSG2  paCanMsg,
    UINT32    dwCount,
    PUINT32   pdwDone
);
```

Parameter

Parameter	Dir.	Description
<i>hCanChn</i>	[in]	Handle of the opened message channel
<i>dwTimeout</i>	[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.
<i>paCanMsg</i>	[in]	Pointer to array of CAN message to transmit
<i>dwCount</i>	[in]	Number of valid entries in message array
<i>pdwDone</i>	[out]	Number of sent CAN messages

Return Value

Return value	Description
VCI_OK	Function succeeded
VCI_E_TXQUEUE_FULL	<i>dwTimeout</i> 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
<i>hDevice</i>	[in]	Handle of the fieldbus adapter
<i>dwCanNo</i>	[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.
<i>phCanShd</i>	[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* instead of the handle of the adapter.

canSchedulerClose

Closes an opened cyclic transmit list.

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

Parameter

Parameter	Dir.	Description
<i>hCanShd</i>	[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,
    PCANCAPABILITIES2 pCanCaps
);
```

Parameter

Parameter	Dir.	Description
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>pCanCaps</i>	[out]	Pointer to a structure of type CANCAPABILITIES2 . 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,
    PCANSCHEDULERSTATUS2 pStatus
);
```

Parameter

Parameter	Dir.	Description
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>pStatus</i>	[out]	Pointer to a structure of type CANSCHEDULERSTATUS2 . 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 [CANSCHEDULERSTATUS2.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.

canSchedulerGetControl

Tries to acquire exclusive access to the CAN controller.

```
HRESULT EXTERN_C canSchedulerGetControl (
    HANDLE hCanShd,
    PHANDLE phCanCtl
);
```

Parameter

Parameter	Dir.	Description
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>phCanCtl</i>	[out]	Points to a variable where the function stores the handle to the CAN controller. This variable is set to NULL if the CAN controller is currently in use.

Return Value

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

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
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>fEnable</i>	[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
<i>hCanShd</i>	[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,
    PCANCYCLICTXMSG2 pMessage,
    PUINT32 pdwIndex
);
```

Parameter

Parameter	Dir.	Description
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>pMessage</i>	[in]	Pointer to an initialized structure of type CANCYCLICTXMSG2 with the transmit object
<i>pdwIndex</i>	[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 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
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>dwIndex</i>	[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
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>dwIndex</i>	[in]	List index of the transmit object to be started. The list index must come from a previous call of the function canSchedulerAddMessage .
<i>wRepeat</i>	[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
<i>hCanShd</i>	[in]	Handle of the opened transmit list
<i>dwIndex</i>	[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
<i>hDevice</i>	[in]	Handle of the fieldbus adapter
<i>dwLinNo</i>	[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.
<i>phLinCtl</i>	[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
<i>hLinCtl</i>	[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
<i>hLinCtl</i>	[in]	Handle of the opened LIN controller
<i>pLinCaps</i>	[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,
    PLINLINESTATUS2 pStatus
);
```

Parameter

Parameter	Dir.	Description
<i>hLinCtl</i>	[in]	Handle of the opened LIN controller
<i>pStatus</i>	[out]	Pointer to a structure of type LINLINESTATUS2 . 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
<i>hLinCtl</i>	[in]	Handle of the opened LIN controller
<i>bMode</i>	[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
<i>wBitrate</i>	[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
<i>hLinCtl</i>	[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
<i>hLinCtl</i>	[in]	Handle of the opened LIN controller
<i>fStart</i>	[in]	Value <code>TRUE</code> starts and value <code>FALSE</code> stops the LIN controller.

Return Value

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

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
<i>hLinCtl</i>	[in]	Handle of the opened LIN controller
<i>fSend</i>	[in]	Determines if a message is directly transmitted to the bus or if it is assigned to the response table of the controller. With <code>TRUE</code> the message is transmitted directly, with <code>FALSE</code> the message is assigned to the response table.
<i>pLinMsg</i>	[in]	Pointer to initialized structure of type <code>LINMSG</code> 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 <code>VciFormatError</code>

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
<i>hDevice</i>	[in]	Handle of the device where the LIN controller is located
<i>dwLinNo</i>	[in]	Number of the LIN controller to open, value 0 selects the first LIN connection, value 1 the second LIN connection, etc (see Remarks)
<i>fExclusive</i>	[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.
<i>phLinMon</i>	[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
<i>hLinMon</i>	[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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>pLinCaps</i>	[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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>pStatus</i>	[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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>wFifoSize</i>	[in]	Size of receive FIFO in number of LIN messages
<i>wThreshold</i>	[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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>fEnable</i>	[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 also [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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>pLinMsg</i>	[out]	Pointer to a <i>PLINMSG</i> 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
VCI_E_TIMEOUT	Timeout interval elapses without a CAN message available.
!=VCI_OK	Error, more information about error code provides the function <i>VciFormatError</i>

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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>paLinMsg</i>	[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.
<i>dwCount</i>	[in]	Number of available slots in LIN message buffer
<i>pdwDone</i>	[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 CAN message available
VCI_E_TIMEOUT	Timeout interval elapses without a CAN message available.
!=VCI_OK	Error, more information about error code provides the function <i>VciFormatError</i>

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

Return Value

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

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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>dwTimeout</i>	[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.
<i>pLinMsg</i>	[out]	Pointer to a <i>LINMSG</i> 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 CAN message available
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
<i>hLinMon</i>	[in]	Handle of the opened LIN monitor
<i>dwTimeout</i>	[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.
<i>paLinMsg</i>	[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.
<i>dwCount</i>	[in]	Size of the array pointed to by <i>paLinMsg</i> in count of LIN messages
<i>pdwDone</i>	[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 CAN message available
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
<i>AsLuid</i>	[out]	ID in form of a LUID. Data type LUID is defined in Windows.
<i>AsInt64</i>	[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
<i>VciMajorVersion</i>	[out]	Major version number of VCI
<i>VciMinorVersion</i>	[out]	Minor version number of VCI
<i>VciRevNumber</i>	[out]	Revision number of VCI
<i>VciBuildNumber</i>	[out]	Build number of VCI
<i>OsMajorVersion</i>	[out]	Major version number of operating system
<i>OsMinorVersion</i>	[out]	Minor version number of operating system
<i>OsBuildNumber</i>	[out]	Build number of operating system
<i>OsPlatformId</i>	[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
<i>DeviceClass</i>	[out]	Class ID of the licensed product
<i>MaxDevices</i>	[out]	Maximum number of allowed devices (0=no limit)
<i>MaxRuntime</i>	[out]	Maximum runtime in seconds (0=no limit)
<i>Restrictions</i>	[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
<i>VciObjectId</i>	[out]	Unique VCI device identifier. The VCI assigns a system wide unique id to every running VCI device.
<i>DriverClass</i>	[out]	Driver class identifier
<i>MajorVersion</i>	[out]	Major driver version number
<i>MinorVersion</i>	[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
<i>VciObjectId</i>	[out]	Unique VCI object identifier
<i>DeviceClass</i>	[out]	Device class identifier
<i>DriverMajorVersion</i>	[out]	Major version number of driver
<i>DriverMinorVersion</i>	[out]	Minor version number of driver
<i>DriverBuildVersion</i>	[out]	Build version number of driver
<i>HardwareBranchVersion</i>	[out]	Branch version number of hardware
<i>HardwareMajorVersion</i>	[out]	Major version number of hardware
<i>HardwareMinorVersion</i>	[out]	Minor version number of hardware
<i>HardwareBuildVersion</i>	[out]	Build version number of hardware
<i>UniqueHardwareId</i>	[out]	Unique hardware identifier
<i>Description</i>	[out]	Device description
<i>Manufacturer</i>	[out]	Device manufacturer
<i>DriverReleaseVersion</i>	[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
<i>BusCtrlCount</i>	[out]	Number of supported bus controllers
<i>BusCtrlTypes</i>	[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
<i>dwCommId</i>	[out]	ID of currently running communication layer
<i>dwStatus</i>	[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 CANBTP

The structure describes the bit timing parameter.

```
typedef struct _CANBTP
{
    UINT32 dwMode;
    UINT32 dwBPS;
    UINT16 wTS1;
    UINT16 wTS2;
    UINT16 wSJW;
    UINT16 wTDO;
} CANBTP, *PCANBTP;
```

Member	Dir.	Description
<i>dwMode</i>	[out]	Operating mode. This bit field determines how the following fields are interpreted. For the operating mode a logical combination of one or more of the following constants can be specified: CAN_BTMODE_RAW: Native mode. The fields <i>dwBPS</i> , <i>wTS1</i> , <i>wTS2</i> , <i>wSJW</i> and <i>wTDO</i> contain hardware specific values for the corresponding registers of the controller. The values of these fields must be inside the limits which are determined by the fields <i>sSdrRangeMin</i> resp. <i>sFdrRangeMin</i> and <i>sSdrRangeMax</i> resp. <i>sFdrRangeMax</i> of structure CANCAPABILITIES2 . CAN_BTMODE_TSM: Activating triple sampling mode
<i>dwBPS</i>	[out]	Transmitting rate in bits per second. If in field <i>dwMode</i> the bit CAN_BTMODE_RAW is set, the hardware specific value for the baud rate prescaler register is expected here. If not, the bit rate in bits per second is expected.
<i>wTS1</i>	[out]	Length of time segment 1. If in field <i>dwMode</i> the bit CAN_BTMODE_RAW is set, the hardware specific number of time quanta for the time segment 1 is expected here. If not, the value defines the length of this time segment in relation to the total number of time quanta per bit.
<i>wTS2</i>	[out]	Length of time segment 2. If in field <i>dwMode</i> the bit CAN_BTMODE_RAW is set, the hardware specific number of time quanta for the time segment 2 is expected here. If not, the value defines the length of this time segment in relation to the total number of time quanta per bit.
<i>wSJW</i>	[out]	Jump width for re-synchronization. If in field <i>dwMode</i> the bit CAN_BTMODE_RAW is set, the hardware specific number of time quanta for the re-synchronization is expected here. If not, the value defines the length of the jumping width in relation to the total number of time quanta per bit.
<i>wTDO</i>	[out]	Offset to the transceiver delay (or Secondary Sample Point SSP) that is automatically determined by the controller. Value is only relevant with fast data bit rate. If in field <i>dwMode</i> the bit CAN_BTMODE_RAW is set, the hardware specific number of CAN clock cycles is expected here. If not, the value defines the Secondary Sample Point (SSP) in relation to the total number of time quanta per bit (example: if <i>wTS1</i> + <i>wTS2</i> =100 and <i>wTDO</i> =65 the SSP is 65% of a bit time). Value 0 deactivates the SSP. If value 0xFFF is specified, the SSP offset is calculated internally based on the other parameters (simplified SSP positioning). For more information about the formula see CiA specification 601-3 Part 3, chapter System Design Recommendation.

6.2.2 CANCAPABILITIES2

The structure describes the CAN controller capabilities.

```
typedef struct _CANCAPABILITIES2
{
    UINT16 wCtrlType;
    UINT16 wBusCoupling;
    UINT32 dwFeatures;
    UINT32 dwCanClkFreq;
    CANBTP sSdrRangeMin;
    CANBTP sSdrRangeMax;
    CANBTP sFdrRangeMin;
```

```

CANBTP sFdrRangeMax;
UINT32 dwTscClkFreq;
UINT32 dwTscDivisor;
UINT32 dwCmsClkFreq;
UINT32 dwCmsDivisor;
UINT32 dwCmsMaxTicks;
UINT32 dwDtxClkFreq;
UINT32 dwDtxDivisor;
UINT32 dwDtxMaxTicks;
} CANCAPABILITIES2, *PCANCAPABILITIES2;

```

Member	Dir.	Description
<i>wCtrlType</i>	[out]	Type of CAN controller. The value of this field is corresponding to a CAN_TYPE_ constant defined in <i>cantype.h</i> .
<i>wBusCoupling</i>	[out]	Type of bus coupling. For the bus coupling the following values are defined: CAN_BUSC_UNDEFINED: undefined CAN_BUSC_LOWSPEED: CAN controller has a low speed coupling. CAN_BUSC_HIGHSPEED: CAN controller has a high speed coupling.
<i>dwFeatures</i>	[out]	Supported features. Value is a logical combination of one or more of the following constants: CAN_FEATURE_STDOEXT: CAN controller supports 11 or 29 bit messages, exclusive, but not both formats simultaneously. CAN_FEATURE_STDANEXT: CAN controller supports 11 and 29 bit messages simultaneously. CAN_FEATURE_RMTFRAME: CAN controller supports remote transmission request (RTR) messages. CAN_FEATURE_ERRFRAME: CAN controller supports returns error frames. CAN_FEATURE_BUSLOAD: CAN controller supports bus load calculation. CAN_FEATURE_IDFILTER: CAN controller supports allows exact message filtering. CAN_FEATURE_LISTONLY: CAN controller supports listen only mode. CAN_FEATURE_SCHEDULER: cyclic transmitting list provided CAN_FEATURE_GENERRFRM: CAN controller supports error frame generation. CAN_FEATURE_DELAYEDTX: CAN controller supports delayed transmitting of messages. CAN_FEATURE_SINGLESHOT: CAN controller supports Single shot mode. If a message is of type <i>Single Shot</i> the controller does not try to transmit again if the message is not transmitted with the first attempt. CAN_FEATURE_HIGHPRIOR: CAN controller supports transmitting high priority messages. Messages with high priority are assigned to a transmitting buffer by the controller, the transmitting buffer is prior to messages in the normal transmitting buffer. Messages of high priority are transmitted with priority to the bus. CAN_FEATURE_AUTOBAUD: CAN controller supports automatic bit rate detection. CAN_FEATURE_EXTDATA: CAN controller provides messages with extended data field, if this bit is not set at a CAN FD controller, it supports maximally 8 byte in the data field. CAN_FEATURE_FASTDATA: CAN controller supports transmission with fast data bit rate. CAN_FEATURE_ISOFRAME: CAN controller supports ISO conform frame (exclusively CAN FD) CAN_FEATURE_NONISOFRAME: CAN controller supports non ISO conform frame (different CRC computation, exclusively CAN FD) CAN_FEATURE_64BITTSC: 64 bit time stamp counter
<i>dwCanClkFreq</i>	[out]	Frequency in hertz of the primary clock generator. The bit rate generator defines together with the values in the structure <i>CANBTP</i> the bit transmission rate for the standard resp. for the nominal arbitration bit rate and the high data bit rate.
<i>sSdrRangeMin</i>	[out]	Minimum bit timing values for standard resp. the nominal arbitration bit rate
<i>sSdrRangeMax</i>	[out]	Maximum bit timing values for standard Minimum bit timing values for standard resp. the nominal arbitration bit rate bit rate
<i>sFdrRangeMin</i>	[out]	Minimum bit timing values for fast data bit rate. All fields of the structure contain the value 0 if the controller do not support a high data bit rate. See CAN_FEATURE_FASTDATA.

Member	Dir.	Description
<i>sFdrRangeMax</i>	[out]	Maximum bit timing values for fast data bit rate. All fields of the structure contain the value 0 if the controller do not support a high data bit rate. See CAN_FEATURE_FASTDATA.
<i>dwTscClkFreq</i>	[out]	Frequency in Hertz of clock generator which is used to create the time stamps of CAN messages (Time Stamp Counter).
<i>dwTscDivisor</i>	[out]	Divisor for the message time stamp counter. Resolution of the time stamps of CAN messages is calculated by the values specified here divided by the frequency of the primary clock generator.
<i>dwCmsClkFreq</i>	[out]	Frequency in Hertz of the clock generator of the cyclic transmitting list (Cyclic Message Timer). If no cyclic transmitting list is available the field contains value 0.
<i>dwCmsDivisor</i>	[out]	Divisor for the clock generator of the cyclic transmitting list. Frequency of cyclic transmitting list is calculated by the frequency of the cyclic message timer divided by the value specified here. If no cyclic transmitting list is available the field contains value 0.
<i>dwCmsMaxTicks</i>	[out]	Maximum cyclic time of the cyclic transmitting list in timer ticks. If no cyclic transmitting list is available the field contains value 0.
<i>dwDtxClkFreq</i>	[out]	Frequency in Hertz of clock generator, that is used for delayed transmission of CAN messages (Delay Timer). If delayed transmission is not supported the field contains value 0.
<i>dwDtxDivisor</i>	[out]	Divisor for the clock generator for delayed transmission of messages. The resolution of the timer for delayed transmission of messages is calculated by the values specified here divided by the frequency of the delay timer. If delayed transmission is not supported the field contains value 0.
<i>dwDtxMaxTicks</i>	[out]	Maximum delay time in number of timer ticks. If delayed transmission is not supported the field contains value 0.

6.2.3 CANINITLINE2

The structure is used to initialize the extended CAN control unit.

```
typedef struct _CANINITLINE2
{
    UINT8 bOpMode;
    UINT8 bExMode;
    UINT8 bSFMode;
    UINT8 bEFMode;
    UINT32 dwSFIds;
    UINT32 dwEFIds;
    CANBTP sBtpSdr;
    CANBTP sBtpFdr;
} CANINITLINE2, *PCANINITLINE2;
```

Member	Dir.	Description
<i>bOpMode</i>	[out]	Operating mode of controller. For the operating mode a logical combination of one or more of the following constants can be specified: CAN_OPMODE_STANDARD: controller accepts messages with 11 bit identifier. CAN_OPMODE_EXTENDED: controller accepts messages with 29 bit identifier. CAN_OPMODE_LISTONLY: controller is used in <i>Listen Only</i> mode (TX passive). CAN_OPMODE_ERRFRAME: controller supports error frames. CAN_OPMODE_LOWSPEED: controller uses low speed bus coupling. CAN_OPMODE_AUTOBAUD: if supported by the controller the controller performs an automatic detection of the bit rate during the initialization. Controller must be connected with running system. If this bit is set the bit timing parameters specified in the fields <i>sBtpSdr</i> and <i>sBtpFdr</i> are ignored.
<i>bExMode</i>	[out]	Extended operating mode. If supported by the controller, a logical combination of one or more of the following constants can be specified: CAN_EXMODE_DISABLED: no extended operating mode is activated. The value also must be specified with all other controllers that do not support CAN FD operating mode. For more information see description of field <i>dwFeatures</i> of structure CANCAPABILITIES2 . CAN_EXMODE_EXTDATA: allows messages with extended data length up to 64 bytes. CAN_EXMODE_FASTDATA: allows fast data bit rate (exclusively available with CAN FD controller with the feature CAN_FEATURE_NONISOFRM) CAN_EXMODE_NONISO:: supports non ISO conform frames.
<i>bSFMode</i>	[out]	Default value for the operating mode of 11 bit filter
<i>bEFMode</i>	[out]	Default value for the operating mode of 29 bit filter
<i>dwSFIds</i>	[out]	Number of CAN IDs supported by the 11 bit filter. With value 0 no filter is specified and all messages with 11 bit ID are allowed to pass. The operating mode specified in <i>bSFMode</i> is not considered.
<i>dwEFIds</i>	[out]	Number of CAN IDs supported by the 29 bit filter. With value 0 no filter is specified and all messages with 29 bit ID are allowed to pass. The operating mode specified in <i>bEFMode</i> is not considered.
<i>sBtpSdr</i>	[out]	Bit timing parameter for default or nominal bit rate resp. for bit rate during the arbitration phase. For more information see description of data type CANBTP .
<i>sBtpFdr</i>	[out]	Bit timing parameter for fast data bit rate. Field is exclusively relevant if the controller supports the fast data transmission and if constant CAN_EXMODE_FASTDATA in field <i>bExMode</i> is specified. For more information see description of data type CANBTP .

6.2.4 CANLINESTATUS2

The structure describes the CAN controller status.

```
typedef struct _CANLINESTATUS2
{
    UINT8 bOpMode;
    UINT8 bExMode;
    UINT8 bBusLoad;
    UINT8 bReserved;
    CANBTP sBtpSdr;
    CANBTP sBtpFdr;
    UINT32 dwStatus;
} CANLINESTATUS2, *PCANLINESTATUS2;
```

Member	Dir.	Description
<i>bOpMode</i>	[out]	Current operating mode of controller. Value is a logical combination of one or more CAN_OPMODE_ (see CANINITLINE2)
<i>bExMode</i>	[out]	Current extended operating mode of controller. Value is a logical combination of one or more CAN_EXMODE_ (see CANINITLINE2).
<i>bBusLoad</i>	[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 CANCAPABILITIES2).
<i>bReserved</i>	[out]	Reserved, set to 0
<i>sBtpSdr</i>	[out]	Current bit timing parameter for nominal bit rate resp. for bit rate during the arbitration phase.
<i>sBtpFdr</i>	[out]	Current bit timing parameter for fast data bit rate
<i>dwStatus</i>	[out]	Current status of CAN controller. Value is a logical combination of one or more of the following constants: CAN_STATUS_TXPEND: CAN controller is currently transmitting a message to the bus (transmission pending). CAN_STATUS_OVRRUN: data overflow in the receiving buffer of the CAN controller had happened. CAN_STATUS_ERRLIM: overflow of an error counter of the CAN controller has happened. CAN_STATUS_BUSOFF: CAN controller has shifted to state <i>BUS-OFF</i> . CAN_STATUS_ININIT: CAN controller is in stopped state. CAN_STATUS_BUSCERR: Faulty bus coupling, only relevant with CAN interfaces with CAN low-speed transceiver and activated low-speed CAN bus. The output pin ERR of the CAN low-speed transceiver is low active. If the output pin ERR of the CAN low-speed transceiver is set to 0, the flag DCAN_STATUS_BUSCERR in the CAN controller status is set to 1. If the output pin ERR of the CAN low-speed transceiver is set to 1, the flag DCAN_STATUS_BUSCERR in the CAN controller status is set to 0. This means, if an error occurs on the CAN bus line of the CAN low-speed transceiver, the flag DCAN_STATUS_BUSCERR in the CAN controller status is set to 1.

6.2.5 CANCHANSTATUS2

The structure describes the CAN message channel status.

```
typedef struct _CANCHANSTATUS2
{
    CANLINESTATUS2 sLineStatus;
    BOOL8 fActivated;
    BOOL8 fRxOverrun;
    UINT8 bRxFifoLoad;
    UINT8 bTxFifoLoad;
} CANCHANSTATUS2, *PCANCHANSTATUS2;
```

Member	Dir.	Description
<i>sLineStatus</i>	[out]	Current status of CAN controller (see CAN_STATUS_ in CANLINESTATUS2)
<i>fActivated</i>	[out]	Shows if message channel is active (TRUE) or inactive (FALSE).
<i>fRxOverrun</i>	[out]	Signalizes an overflow in the receiving buffer with the value TRUE.
<i>bRxFifoLoad</i>	[out]	Receive FIFO load in percent (0..100)
<i>bTxFifoLoad</i>	[out]	Transmit FIFO load in percent (0..100)

6.2.6 CANRTINFO

The structure describes the CAN run-time status information.

```
typedef struct _CANRTINFO
{
    UINT32 dwNumChannels;
    UINT32 dwActChannels;
    UINT32 dwLockStatus;
    UINT16 wRxFifoLoad;
    UINT16 wTxFifoLoad;
} CANRTINFO, *PCANRTINFO;
```

Member	Dir.	Description
<i>dwNumChannels</i>	[out]	Total number of open channels
<i>dwActChannels</i>	[out]	Number of active channels
<i>dwLockStatus</i>	[out]	Lock status of various interfaces CAN_RTI_LOCKSTAT_CTL: ICanControl locked CAN_RTI_LOCKSTAT_SHD: ICanScheduler locked CAN_RTI_LOCKSTAT_CHN: exclusive channel lock
<i>wRxFifoLoad</i>	[out]	Device receive FIFO load in percent (0..100)
<i>wTxFifoLoad</i>	[out]	Device transmit FIFO load in percent (0..100)

6.2.7 CANSCHEDULERSTATUS2

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

```
typedef struct _CANSCHEDULERSTATUS2
{
    CANLINESTATUS2 sLineStatus;
    UINT8 bTaskStat;
    UINT8 abMsgStat[16];
} CANSCHEDULERSTATUS2, *PCANSCHEDULERSTATUS2;
```

Member	Dir.	Description
<i>sLineStatus</i>	[out]	Current state of CAN line controller (see <code>CAN_STATUS_</code> in CANLINESTATUS2)
<i>bTaskStat</i>	[out]	Current status of cyclic transmitting task <code>CAN_CTXTSK_STAT_STOPPED</code> : cyclic transmit task stopped <code>CAN_CTXTSK_STAT_RUNNING</code> : cyclic transmit task running
<i>abMsgStat</i>	[out]	Table with status of all 16 transmitting objects. Each table entry can take one of the following values: <code>CAN_CTXTSK_STAT_EMPTY</code> : entry is not assigned to a transmitting object resp. the entry is currently not used. <code>CAN_CTXTSK_STAT_BUSY</code> : processing of message in progress <code>CAN_CTXTSK_STAT_DONE</code> : processing of message completed

6.2.8 CANMSGINFO

The data type combines different information about a CAN message in a union. The individual values can either be addresses via byte fields or via bus bit fields.

```
typedef struct _CANMSGINFO
{
    struct {
        UINT8 bType;
        union {
            UINT8 bReserved;
            UINT8 bFlags2;
        };
        UINT8 bFlags;
        UINT8 bAccept;
    } Bytes;
    struct {
        UINT32 type : 8;
        UINT32 ssm : 1;
        UINT32 hpm : 1;
        UINT32 edl : 1;
        UINT32 fdr : 1;
        UINT32 esi : 1;
        UINT32 res : 3;
        UINT32 dlc : 4;
        UINT32 ovr : 1;
        UINT32 srr : 1;
        UINT32 rtr : 1;
        UINT32 ext : 1;
        UINT32 afc : 8;
    } Bits;
} CANMSGINFO, *PCANMSGINFO;
```

The information are accessed byte by byte via the following byte fields:

Member	Dir.	Description
<i>bType</i>	[out]	Message type (see bit field <i>type</i>)
<i>bReserved</i>	[out]	Reserved
<i>bFlags2</i>	[out]	Extended message flags CAN_MSGFLAGS2_SSM: [bit 0] single shot mode (see bit field <i>ssm</i>) CAN_MSGFLAGS2_HPM: [bit 1] high priority message (see bit field <i>hpm</i>) CAN_MSGFLAGS2_EDL: [bit 2] extended data length (see bit field <i>edl</i>) CAN_MSGFLAGS2_FDR: [bit 3] fast data bit rate (see bit field <i>fdl</i>) CAN_MSGFLAGS2_ESI: [bit 4] error state indicator (see bit field <i>esi</i>) CAN_MSGFLAGS2_RES: [bit 5..7] reserved bits (see bit field <i>res</i>)
<i>bFlags</i>	[out]	Standard message flags CAN_MSGFLAGS_DLC: [bit 0] data length code (see bit field <i>dlc</i>) CAN_MSGFLAGS_OVR: [bit 4] data overrun flag (see bit field <i>ovr</i>) CAN_MSGFLAGS_SRR: [bit 5] self reception request (see bit field <i>srr</i>) CAN_MSGFLAGS_RTR: [bit 6] remote transmission request (see bit field <i>rtr</i>) CAN_MSGFLAGS_EXT: [bit 7] frame format (0 = 11 bit, 1= 29 bit, (see bit field <i>ext</i>)
<i>bAccept</i>	[out]	Shows in receive messages which filter accepted the message (see bit field <i>afc</i>)

The information are accessed bit by bit via the following bit fields:

<i>type</i>	[out]	Message type, for transmit messages exclusively the message type CAN_MSGTYPE_DATA is valid.												
		<table border="1"> <tbody> <tr> <td>CAN_MSGTYPE_DATA</td> <td> <p>Standard data message</p> <p>Fields in receive messages (<i>CANMSG2</i>): <i>dwMsgId</i> contains the ID of the message, <i>dwTime</i> the receiving time in ticks, <i>abData</i> contains depending on the length (see <i>bits.dlc</i>) the data bytes of the message.</p> <p>Fields in transmit messages (<i>CANMSG2</i>): <i>dwMsgId</i> contains the message ID, <i>abData</i> the data bytes to be transmitted, <i>dwTime</i> is 0 or in delayed messages the desired delay time in ticks to the message transmitted before. See Transmitting Messages Delayed, p. 19.</p> </td> </tr> <tr> <td>CAN_MSGTYPE_INFO:</td> <td> <p>Information message</p> <p>Generated by certain events resp. state changes of the control unit and registered in the receiving buffers of all active message channels. Field <i>dwMsgId</i> of the message (<i>CANMSG2</i>) contains the value CAN_MSGID_INFO. Field <i>abData[0]</i> contains one of the following values:</p> <p>CAN_INFO_START: controller is started, field <i>dwTime</i> contains the starting point.</p> <p>CAN_INFO_STOP controller is stopped, field <i>dwTime</i> contains value 0.</p> <p>CAN_INFO_RESET controller is reset, field <i>dwTime</i> contains value 0.</p> </td> </tr> <tr> <td>CAN_MSGTYPE_ERROR:</td> <td> <p>Error frame</p> <p>Generated if a bus error occurs and registered in the receiving buffers off all active message channels, provided that the flag CAN_OPMODE_ERRFRAME is set during the initialization of the CAN controller. Field <i>dwMsgId</i> of the message (<i>CANMSG2</i>) contains the value CAN_MSGID_ERROR, field <i>dwTime</i> the time of the event and field <i>abData[0]</i> contains one of the following values: CAN_ERROR_STUFF (stuff error), CAN_ERROR_FORM (format error), CAN_ERROR_ACK (acknowledgement error), CAN_ERROR_BIT (bit error), CAN_ERROR_FDB (fast data bit error), CAN_ERROR_CRC (CRC error), CAN_ERROR_OTHER (unspecified), CAN_ERROR_DLC (data length error).</p> </td> </tr> <tr> <td>CAN_MSGTYPE_STATUS</td> <td> <p>Status frame</p> <p>Generated by state changes of the CAN controller and registered in the receiving buffers of all active message channels. Field <i>dwMsgId</i> (<i>CANMSG2</i>) contains the value CAN_MSGID_STATUS, field <i>dwTime</i> the time of the event and field <i>abData[0]</i> contains the low byte of the current CAN state. The content of the other data fields is undefined.</p> </td> </tr> <tr> <td>CAN_MSGTYPE_WAKEUP</td> <td>Not used</td> </tr> <tr> <td>CAN_MSGTYPE_TIMEOVR</td> <td>Timer overrun</td> </tr> </tbody> </table>	CAN_MSGTYPE_DATA	<p>Standard data message</p> <p>Fields in receive messages (<i>CANMSG2</i>): <i>dwMsgId</i> contains the ID of the message, <i>dwTime</i> the receiving time in ticks, <i>abData</i> contains depending on the length (see <i>bits.dlc</i>) the data bytes of the message.</p> <p>Fields in transmit messages (<i>CANMSG2</i>): <i>dwMsgId</i> contains the message ID, <i>abData</i> the data bytes to be transmitted, <i>dwTime</i> is 0 or in delayed messages the desired delay time in ticks to the message transmitted before. See Transmitting Messages Delayed, p. 19.</p>	CAN_MSGTYPE_INFO:	<p>Information message</p> <p>Generated by certain events resp. state changes of the control unit and registered in the receiving buffers of all active message channels. Field <i>dwMsgId</i> of the message (<i>CANMSG2</i>) contains the value CAN_MSGID_INFO. Field <i>abData[0]</i> contains one of the following values:</p> <p>CAN_INFO_START: controller is started, field <i>dwTime</i> contains the starting point.</p> <p>CAN_INFO_STOP controller is stopped, field <i>dwTime</i> contains value 0.</p> <p>CAN_INFO_RESET controller is reset, field <i>dwTime</i> contains value 0.</p>	CAN_MSGTYPE_ERROR:	<p>Error frame</p> <p>Generated if a bus error occurs and registered in the receiving buffers off all active message channels, provided that the flag CAN_OPMODE_ERRFRAME is set during the initialization of the CAN controller. Field <i>dwMsgId</i> of the message (<i>CANMSG2</i>) contains the value CAN_MSGID_ERROR, field <i>dwTime</i> the time of the event and field <i>abData[0]</i> contains one of the following values: CAN_ERROR_STUFF (stuff error), CAN_ERROR_FORM (format error), CAN_ERROR_ACK (acknowledgement error), CAN_ERROR_BIT (bit error), CAN_ERROR_FDB (fast data bit error), CAN_ERROR_CRC (CRC error), CAN_ERROR_OTHER (unspecified), CAN_ERROR_DLC (data length error).</p>	CAN_MSGTYPE_STATUS	<p>Status frame</p> <p>Generated by state changes of the CAN controller and registered in the receiving buffers of all active message channels. Field <i>dwMsgId</i> (<i>CANMSG2</i>) contains the value CAN_MSGID_STATUS, field <i>dwTime</i> the time of the event and field <i>abData[0]</i> contains the low byte of the current CAN state. The content of the other data fields is undefined.</p>	CAN_MSGTYPE_WAKEUP	Not used	CAN_MSGTYPE_TIMEOVR	Timer overrun
CAN_MSGTYPE_DATA	<p>Standard data message</p> <p>Fields in receive messages (<i>CANMSG2</i>): <i>dwMsgId</i> contains the ID of the message, <i>dwTime</i> the receiving time in ticks, <i>abData</i> contains depending on the length (see <i>bits.dlc</i>) the data bytes of the message.</p> <p>Fields in transmit messages (<i>CANMSG2</i>): <i>dwMsgId</i> contains the message ID, <i>abData</i> the data bytes to be transmitted, <i>dwTime</i> is 0 or in delayed messages the desired delay time in ticks to the message transmitted before. See Transmitting Messages Delayed, p. 19.</p>													
CAN_MSGTYPE_INFO:	<p>Information message</p> <p>Generated by certain events resp. state changes of the control unit and registered in the receiving buffers of all active message channels. Field <i>dwMsgId</i> of the message (<i>CANMSG2</i>) contains the value CAN_MSGID_INFO. Field <i>abData[0]</i> contains one of the following values:</p> <p>CAN_INFO_START: controller is started, field <i>dwTime</i> contains the starting point.</p> <p>CAN_INFO_STOP controller is stopped, field <i>dwTime</i> contains value 0.</p> <p>CAN_INFO_RESET controller is reset, field <i>dwTime</i> contains value 0.</p>													
CAN_MSGTYPE_ERROR:	<p>Error frame</p> <p>Generated if a bus error occurs and registered in the receiving buffers off all active message channels, provided that the flag CAN_OPMODE_ERRFRAME is set during the initialization of the CAN controller. Field <i>dwMsgId</i> of the message (<i>CANMSG2</i>) contains the value CAN_MSGID_ERROR, field <i>dwTime</i> the time of the event and field <i>abData[0]</i> contains one of the following values: CAN_ERROR_STUFF (stuff error), CAN_ERROR_FORM (format error), CAN_ERROR_ACK (acknowledgement error), CAN_ERROR_BIT (bit error), CAN_ERROR_FDB (fast data bit error), CAN_ERROR_CRC (CRC error), CAN_ERROR_OTHER (unspecified), CAN_ERROR_DLC (data length error).</p>													
CAN_MSGTYPE_STATUS	<p>Status frame</p> <p>Generated by state changes of the CAN controller and registered in the receiving buffers of all active message channels. Field <i>dwMsgId</i> (<i>CANMSG2</i>) contains the value CAN_MSGID_STATUS, field <i>dwTime</i> the time of the event and field <i>abData[0]</i> contains the low byte of the current CAN state. The content of the other data fields is undefined.</p>													
CAN_MSGTYPE_WAKEUP	Not used													
CAN_MSGTYPE_TIMEOVR	Timer overrun													

			Generated by the time stamp counter with every overflow and registered in the receiving buffer of all active message channels. Field <i>dwTime</i> of the message contains the time of the event and field <i>dwMsgId</i> the number of occurred overflows (normally 1). The content of the data fields <i>abData</i> is undefined.																		
		CAN_MSGTYPE_TIMERST	Not used																		
<i>ssm</i>	[out]		Single shot message. If this bit is set in transmitting messages the controller tries to transmit the message only once. If the message loses its arbitration during the first transmitting attempt, the controller rejects the message and no further automatic transmitting attempt follows. If this bit is 0 no transmitting is attempted until the message has been transmitted over the bus. For receive messages this bit has no significance.																		
<i>hpm</i>	[out]		High priority message. Transmit messages with high priority are assigned to a transmitting buffer by the controller, the transmitting buffer is prior to messages in the normal transmitting buffer. Messages of high priority are transmitted with priority to the bus. For receive messages this bit has no significance.																		
<i>edl</i>	[out]		Message with extended data length. For more information see description of data length field <i>Bits.dlc</i> . The bit is exclusively valid with extended controller operating mode CAN_EXMODE_EXTDATA.																		
<i>fdr</i>	[out]		This bit can be set in transmit messages to transfer the data bytes and bits from the DLC field with high bit rate on the bus. If this bit is set the RTR bit is ignored. See description of <i>bits.rtr</i> . The bit is exclusively valid with extended controller operating mode CAN_EXMODE_FASTDATA.																		
<i>esi</i>	[out]		Error state indicator. Nodes that are <i>error active</i> transmit this bit dominant (0), nodes that are <i>error passive</i> recessive (1). This bit is exclusively considered in receive messages. In transmit messages it is has no significance and must be set to 0.																		
<i>res</i>	[out]		Reserved for further extensions. Due to compatibility reasons set field always to 0.																		
<i>dlc</i>	[out]		Data length code. The value defines the number of valid data bytes in field <i>abData</i> of a message. The following assignment applies: <table border="1"> <thead> <tr> <th><i>dlc</i></th> <th>Number of data bytes</th> </tr> </thead> <tbody> <tr> <td>0...8</td> <td>0...8</td> </tr> <tr> <td>9</td> <td>12</td> </tr> <tr> <td>10</td> <td>16</td> </tr> <tr> <td>11</td> <td>20</td> </tr> <tr> <td>12</td> <td>24</td> </tr> <tr> <td>13</td> <td>32</td> </tr> <tr> <td>14</td> <td>48</td> </tr> <tr> <td>15</td> <td>64</td> </tr> </tbody> </table> <p>A value higher than 8 is exclusively allowed in messages with extended data field (see CANMSG2). To transmit a message with more than 8 byte the CAN must be used in the operating mode CAN_EXMODE_EXTDATA and additionally the bit <i>edl</i> of the message to be transmitted must be set to 1.</p>	<i>dlc</i>	Number of data bytes	0...8	0...8	9	12	10	16	11	20	12	24	13	32	14	48	15	64
<i>dlc</i>	Number of data bytes																				
0...8	0...8																				
9	12																				
10	16																				
11	20																				
12	24																				
13	32																				
14	48																				
15	64																				
<i>ovr</i>	[out]		Data overrun. The bit is set to 1 in receive messages if an overflow of the receiving FIFO took place.																		
<i>srr</i>	[out]		Self reception request. If the bit is set in transmit messages the message is assigned to the receiving FIFO as soon as it is transmitted to the bus. In receive messages a set bit indicates that it is a self reception message. This bit must not be mistaken as substitute remote request (SRR) bit of CAN FD.																		
<i>rtr</i>	[out]		Remote transmission request. This bit is set in transmit messages to scan other bus participants specifically for certain messages. Observe that the bit is ignored if one of the bits <i>edl</i> or <i>fdr</i> is also set. RTR messages are not possible with CAN FD.																		
<i>ext</i>	[out]		Extended frame format (0=standard, 1=extended)																		
<i>afc</i>	[out]		Acceptance filter code, shows in receive messages which filter accepted the message. The following values are defined: <table border="1"> <tbody> <tr> <td>CAN_ACCEPT_REJECT</td> <td>Message not accepted</td> </tr> <tr> <td>CAN_ACCEPT_ALWAYS</td> <td>Message always accepted</td> </tr> <tr> <td>CAN_ACCEPT_FILTER_1</td> <td>Message accepted by the acceptance filter. The filter must be used in operating mode CAN_FILTER_INCL.</td> </tr> <tr> <td>CAN_ACCEPT_FILTER_2</td> <td>Message accepted by the filter list. Exclusively messages of type CAN_MSGTYPE_DATA contain this value.</td> </tr> <tr> <td>CAN_ACCEPT_PASSEXCL</td> <td>Used in the filter operating mode CAN_FILTER_EXCL if a message of type CAN_MSGTYPE_DATA has been accepted.</td> </tr> </tbody> </table>	CAN_ACCEPT_REJECT	Message not accepted	CAN_ACCEPT_ALWAYS	Message always accepted	CAN_ACCEPT_FILTER_1	Message accepted by the acceptance filter. The filter must be used in operating mode CAN_FILTER_INCL.	CAN_ACCEPT_FILTER_2	Message accepted by the filter list. Exclusively messages of type CAN_MSGTYPE_DATA contain this value.	CAN_ACCEPT_PASSEXCL	Used in the filter operating mode CAN_FILTER_EXCL if a message of type CAN_MSGTYPE_DATA has been accepted.								
CAN_ACCEPT_REJECT	Message not accepted																				
CAN_ACCEPT_ALWAYS	Message always accepted																				
CAN_ACCEPT_FILTER_1	Message accepted by the acceptance filter. The filter must be used in operating mode CAN_FILTER_INCL.																				
CAN_ACCEPT_FILTER_2	Message accepted by the filter list. Exclusively messages of type CAN_MSGTYPE_DATA contain this value.																				
CAN_ACCEPT_PASSEXCL	Used in the filter operating mode CAN_FILTER_EXCL if a message of type CAN_MSGTYPE_DATA has been accepted.																				

6.2.9 CANMSG2

The structure describes the extended CAN message structure.

```
typedef struct _CANMSG2
{
    UINT32 dwTime;
    UINT32 _rsvd_;
    UINT32 dwMsgId;
    CANMSGINFO uMsgInfo;
    UINT8 abData[64];
} CANMSG2, *PCANMSG2;
```

Member	Dir.	Description
<i>dwTime</i>	[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 delayed transmitting messages this field determines with how many ticks delay the message is transmitted after the message sent before.
<i>_rsvd_</i>	[out]	Reserved (set to 0)
<i>dwMsgId</i>	[out]	CAN ID of the message in Intel format (aligned right) without RTR bit.
<i>uMsgInfo</i>	[out]	Bit field with information about the message type. For detailed description of bit field see CANMSGINFO .
<i>abData</i>	[out]	Array for up to 64 data bytes. Number of valid data bytes is defined by field <i>uMsgInfo.Bits.dlc</i> .



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

6.2.10 CANCYCLICTXMSG2

The structure describes the extended cyclic transmit message.

```
typedef struct _CANCYCLICTXMSG2
{
    UINT16 wCycleTime;
    UINT8 bIncrMode;
    UINT8 bByteIndex;
    UINT32 dwMsgId;
    CANMSGINFO uMsgInfo;
    UINT8 abData[64];
} CANCYCLICTXMSG2, *PCANCYCLICTXMSG2;
```

Member	Dir.	Description
<i>wCycleTime</i>	[out]	Cycle time of the message in number ticks. The cycle time can be calculated in the fields <i>dwClockFreq</i> and <i>dwCmsDivisor</i> of structure CANCAPABILITIES2 with the following formula. $T \text{ cycle [s]} = (\text{dwCmsDivisor} / \text{dwClockFreq}) * \text{wCycleTime}$ The maximum value for the field is limited to the value in field <i>dwCmsMaxTicks</i> of structure CANCAPABILITIES2 .
<i>bIncrMode</i>	[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 CAN identifier (field <i>dwMsgId</i>). If the field reaches the value 2048 (11 bit ID) resp. 536.870.912 (29 bit ID) an overflow automatically takes place. CAN_CTXMSG_INC_8: Increment 8 bit data field. The data byte to be incremented is determined via the parameter <i>bByteIndex</i> . If the maximum value 255 is exceeded an overflow to 0 takes place. CAN_CTXMSG_INC_16: Increment 16 bit data field. The low byte of the 16 bit value to be incremented is determined via the field <i>bByteIndex</i> . The high byte is in the data field on position <i>bByteIndex+1</i> . If the maximum value 65535 is exceeded an overflow to 0 takes place.
<i>bByteIndex</i>	[out]	Determines the byte resp. the low byte (LSB) of the 16 bit value in data field <i>abData</i> , that is automatically incremented after each transmission. The value range of the field is limited by the data length specified in the field <i>uMsgInfo.Bits.dlc</i> of structure CANMSGINFO and it is limited to the range 0 to (<i>dlc-1</i>) in case of 8 bit increment and 0 to (<i>dlc-2</i>) in case of 16 bit increment.
<i>dwMsgId</i>	[out]	CAN ID of the message in Intel format (aligned right) without RTR bit.
<i>uMsgInfo</i>	[out]	Bit field with information about the message type. For description of bit field see CANMSGINFO .
<i>abData</i>	[out]	Array for up to 64 data bytes. Number of valid data bytes is defined by field <i>uMsgInfo.Bits.dlc</i> .

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
<i>bOpMode</i>	[in]	Operating mode of LIN controller. One or more of the following constants can be specified: LIN_OPMODE_SLAVE: Slave mode (default) LIN_OPMODE_MASTER: Master mode (if supported see LINCAPABILITIES). LIN_OPMODE_ERRORS: Reception of error frames enabled
<i>bReserved</i>	[in]	Reserved. Value must be initialized with 0.
<i>wBitrate</i>	[in]	Transmitting rate in bits per second. The specified value must be in between the limits that are determined by the constants LIN_BITRATE_MIN and LIN_BITRATE_MAX. If the controller is used as slave and supports an automatic bit detection the bit rate can be determined automatically by setting the value LIN_BITRATE_AUTO.

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
<i>dwFeatures</i>	[out]	Supported features. Value is a logical combination of one or more of the following constants: LIN_FEATURE_MASTER: LIN controller supports Master mode. LIN_FEATURE_AUTORATE: LIN controller supports automatic bit rate detection. LIN_FEATURE_ERRFRAME: LIN controller supports reception of error frames. LIN_FEATURE_BUSLOAD: LIN controller supports bus load calculation. LIN_FEATURE_SLEEP: LIN controller supports sleep message (master only). LIN_FEATURE_WAKEUP: LIN controller supports wakeup message.
<i>dwClockFreq</i>	[out]	Frequency in hertz of the primary timer
<i>dwTscDivisor</i>	[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
<i>bOpMode</i>	[out]	Current operating mode of controller (see <i>LIN_OPMODE_</i> in <i>LININITLINE</i>)
<i>bBusLoad</i>	[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 <i>LINCAPABILITIES</i>).
<i>wBitrate</i>	[out]	Currently specified transmission rate in bits per second
<i>dwStatus</i>	[out]	Current status of LIN controller. Value is a logical combination of one or more of the following constants: <i>LIN_STATUS_TXPEND</i> : controller is currently transmitting a message to the bus. <i>LIN_STATUS_OVRRUN</i> : data overflow occurred in the receiving buffer of the controller: <i>LIN_STATUS_ININIT</i> : controller is in stopped state. <i>LIN_STATUS_ERRLIM</i> : overflow of an error counter of the controller occurred. <i>LIN_STATUS_BUSOFF</i> : controller has shifted to state <i>BUS-OFF</i> .

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
<i>sLineStatus</i>	[out]	Current status of LIN controller. For more information see description of data structure <i>LINLINESTATUS</i> .
<i>fActivated</i>	[out]	Shows if message monitor is active (TRUE) or inactive (FALSE).
<i>fRxOverrun</i>	[out]	Signalizes an overflow in the receiving buffer with the value TRUE.
<i>bRxFifoLoad</i>	[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
<i>dwTime</i>	[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 <i>dwClockFreq</i> and <i>dwTscDivisor</i> of structure LINCAPABILITIES with the following formula: Resolution [s] = <i>dwTscDivisor</i> / <i>dwClockFreq</i>
<i>uMsgInfo</i>	[out]	Bit field with information about the message. For detailed description of bit field see LINMSGINFO .
<i>abData</i>	[out]	Array for up to 8 data bytes. Number of valid data bytes is determined by the field <i>uMsgInfo.Bits.dlen</i> .

This page intentionally left blank

