Manual

VCI - Virtual CAN Interface

C-API Programmers Manual

Software Version 3

IXXAT

The expert for industrial and automotive communication

IXXAT

Headquarter
IXXAT Automation GmbH
Leibnizstr. 15
D-88250 Weingarten

Tel.: +49 (0)7 51 / 5 61 46-0
Fax: +49 (0)7 51 / 5 61 46-29
Internet: www.ixxat.de
e-Mail: info@ixxat.de

US Sales Office
IXXAT Inc.
120 Bedford Center Road
USA-Bedford, NH 03110

Phone: +1-603-471-0800
Fax: +1-603-471-0880
Internet: www.ixxat.com
e-Mail: sales@ixxat.com

Support

In case of unsolvable problems with this product or other IXXAT products please contact IXXAT in written form by:

Fax: +49 (0)7 51 / 5 61 46-29
e-Mail: support@ixxat.de

For customers from the USA/Canada
Fax: +1-603-471-0880
e-Mail: techsupport@ixxat.com

Copyright

Duplication (copying, printing, microfilm or other forms) and the electronic distribution of this document is only allowed with explicit permission of IXXAT Automation GmbH. IXXAT Automation GmbH reserves the right to change technical data without prior announcement. The general business conditions and the regulations of the license agreement do apply. All rights are reserved.

Document number: 4.02.0250.20012
Version 1.5

1 System overview.......7
2 Device management and device access.......9
2.1 Overview.......9
2.2 List of available devices and interface boards.......10
2.3 Searching for certain devices and interface boards.......11
2.4 Accessing a device or interface board.......12
3 Accessing the bus.......14
3.1 Accessing the CAN bus.......14
3.1.1 Overview.......14
3.1.2 Control unit.......15
3.1.2.1 Controller states.......16
3.1.2.2 Message filters.......18
3.1.3 Message channel.......20
3.1.3.1 Reception of CAN messages.......22
3.1.3.2 Transmission of CAN messages.......23
3.1.3.3 Delayed transmission of CAN messages.......24
3.1.4 Cyclic transmit list.......25
3.2 Accessing the LIN bus.......28
3.2.1 Overview.......28
3.2.2 Control unit.......29
3.2.2.1 Controller states.......30
3.2.2.2 Transmission of LIN messages.......31
3.2.3 Message monitor.......32
3.2.3.1 Reception of LIN messages.......34
4 Interface description.......36
4.1 General functions.......36
4.1.1 vciInitialize.......36
4.1.2 vciFormatError.......36
4.1.3 vciDisplayError.......37
4.1.4 vciGetVersion.......37
4.1.5 vciLuidToChar.......38
4.1.6 vciCharToLuid.......39
4.1.7 vciGuidToChar.......39
4.1.8 vciCharToGuid.......40
4.2 Functions for the device management.......41

4.2.1 Functions for accessing the device list 41
4.2.1.1 vciEnumDeviceOpen 41
4.2.1.2 vciEnumDeviceClose.. 42
4.2.1.3 vciEnumDeviceNext... 42
4.2.1.4 vciEnumDeviceReset.. 43
4.2.1.5 vciEnumDeviceWaitEvent 43
4.2.1.6 vciFindDeviceByHwid... 44
4.2.1.7 vciFindDeviceByClass 45
4.2.1.8 vciSelectDeviceDlg 45
4.2.2 Functions for accessing CAN interface boards 46
4.2.2.1 vciDeviceOpen 46
4.2.2.2 vciDeviceOpenDlg 47
4.2.2.3 vciDeviceClose 47
4.2.2.4 vciDeviceGetInfo 48
4.2.2.5 vciDeviceGetCaps... 48
4.3 Functions for CAN access 49
4.3.1 Control unit.. 49
4.3.1.1 canControlOpen 49
4.3.1.2 canControlClose.. 50
4.3.1.3 canControlGetCaps.. 51
4.3.1.4 canControlGetStatus.. 51
4.3.1.5 canControlDetectBitrate... 52
4.3.1.6 canControlInitialize 54
4.3.1.7 canControlReset... 55
4.3.1.8 canControlStart 56
4.3.1.9 canControlSetAccFilter 57
4.3.1.10 canControlAddFilterIds 58
4.3.1.11 canControlRemFilterIds 59
4.3.2 Message channel 60
4.3.2.1 canChannelOpen 60
4.3.2.2 canChannelClose 61
4.3.2.3 canChannelGetCaps... 61
4.3.2.4 canChannelGetStatus... 62
4.3.2.5 canChannelInitialize 63
4.3.2.6 canChannelActivate 64
4.3.2.7 canChannelPeekMessage 64
4.3.2.8 canChannelPostMessage... 65

Copyright IXXAT Automation GmbH

4.3.2.9 canChannelWaitRxEvent 66
4.3.2.10 canChannelWaitTxEvent 67
4.3.2.11 canChannelReadMessage 68
4.3.2.12 canChannelSendMessage 69
4.3.3 Cyclic transmit list 70
4.3.3.1 canSchedulerOpen 70
4.3.3.2 canSchedulerClose 71
4.3.3.3 canSchedulerGetCaps 71
4.3.3.4 canSchedulerGetStatus 72
4.3.3.5 canSchedulerActivate 72
4.3.3.6 canSchedulerReset 73
4.3.3.7 canSchedulerAddMessage 74
4.3.3.8 canSchedulerRemMessage 74
4.3.3.9 canSchedulerStartMessage 75
4.3.3.10 canSchedulerStopMessage 76
4.4 Functions for LIN access 76
4.4.1 Control unit 76
4.4.1.1 linControlOpen 76
4.4.1.2 linControlClose 77
4.4.1.3 linControlGetCaps 78
4.4.1.4 linControlGetStatus 78
4.4.1.5 linControlInitialize 79
4.4.1.6 linControlReset 80
4.4.1.7 linControlStart 80
4.4.1.8 linControlWriteMessage 81
4.4.2 Message monitor 81
4.4.2.1 linMonitorOpen 82
4.4.2.2 linMonitorClose 83
4.4.2.3 linMonitorGetCaps 83
4.4.2.4 linMonitorGetStatus 84
4.4.2.5 linMonitorInitialize 84
4.4.2.6 linMonitorActivate 85
4.4.2.7 linMonitorPeekMessage 86
4.4.2.8 linMonitorWaitRxEvent 86
4.4.2.9 linMonitorReadMessage 87
5 Types and structures 89
5.1 VCI-specific data types 89

5.1.1 VCIID 89
5.1.2 VCIDEVICEINFO 89
5.1.3 VCIDEVICECAPS 91
5.2 CAN-specific data types 91
5.2.1 CANCAPABILITIES 91
5.2.2 CANLINESTATUS 93
5.2.3 CANCHANSTATUS 94
5.2.4 CANSCHEDULERSTATUS 95
5.2.5 CANMSGINFO 96
5.2.6 CANMSG 99
5.2.7 CANCYCLICTXMSG 100
5.3 LIN-specific data types 101
5.3.1 LINCAPABILITIES 101
5.3.2 LINLINESTATUS 102
5.3.3 LINMONITORSTATUS 102
5.3.4 LINMSGINFO 103
5.3.5 LINMSG 106

1 System overview

The Virtual Card Interface (VCI) is a system extension intended to enable applications uniform access to different CAN interface boards. The following diagram shows the basic structure of the system and its individual components.

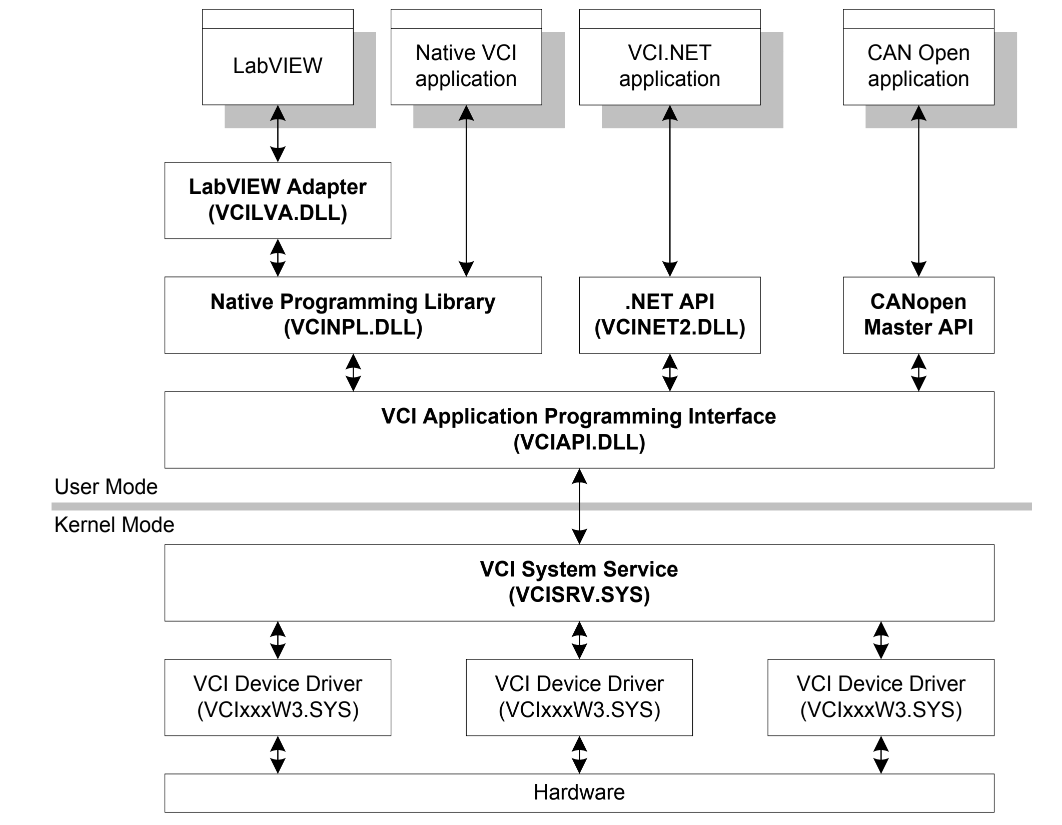


Fig. 1-1: System components

The VCI basically consists of the following components:

- Native VCI programming interface (VCINPL.DLL)
- VCI.NET programming interface (VCINET2.DLL).
- VCI system service API (VCIAPI.DLL)
- VCI system service (VCISRV.SYS)
- One or more VCI device drivers (VCIxxxWy.SYS)

The programming interfaces establish the connection between the VCI system service, or VCI server for short, and the application programs via a set of predefined interfaces and functions. The LabVIEW adapter is used only for adaptation to the data types used by the native programming interface but does not provide any additional functionality itself.

The VCI server running in the kernel of the operating system mainly handles management of the VCI device drivers, controls access to the CAN interface boards and provides mechanisms for the exchange of data between the application level and the operating system level.

The programming interface shown below consists of the following subcomponents and functions.

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

2 Device management and device access

2.1 Overview

The device management of VCI enables listing and primary access to the CAN interface boards logged into the VCI server. The functions shown in the following diagram are available:

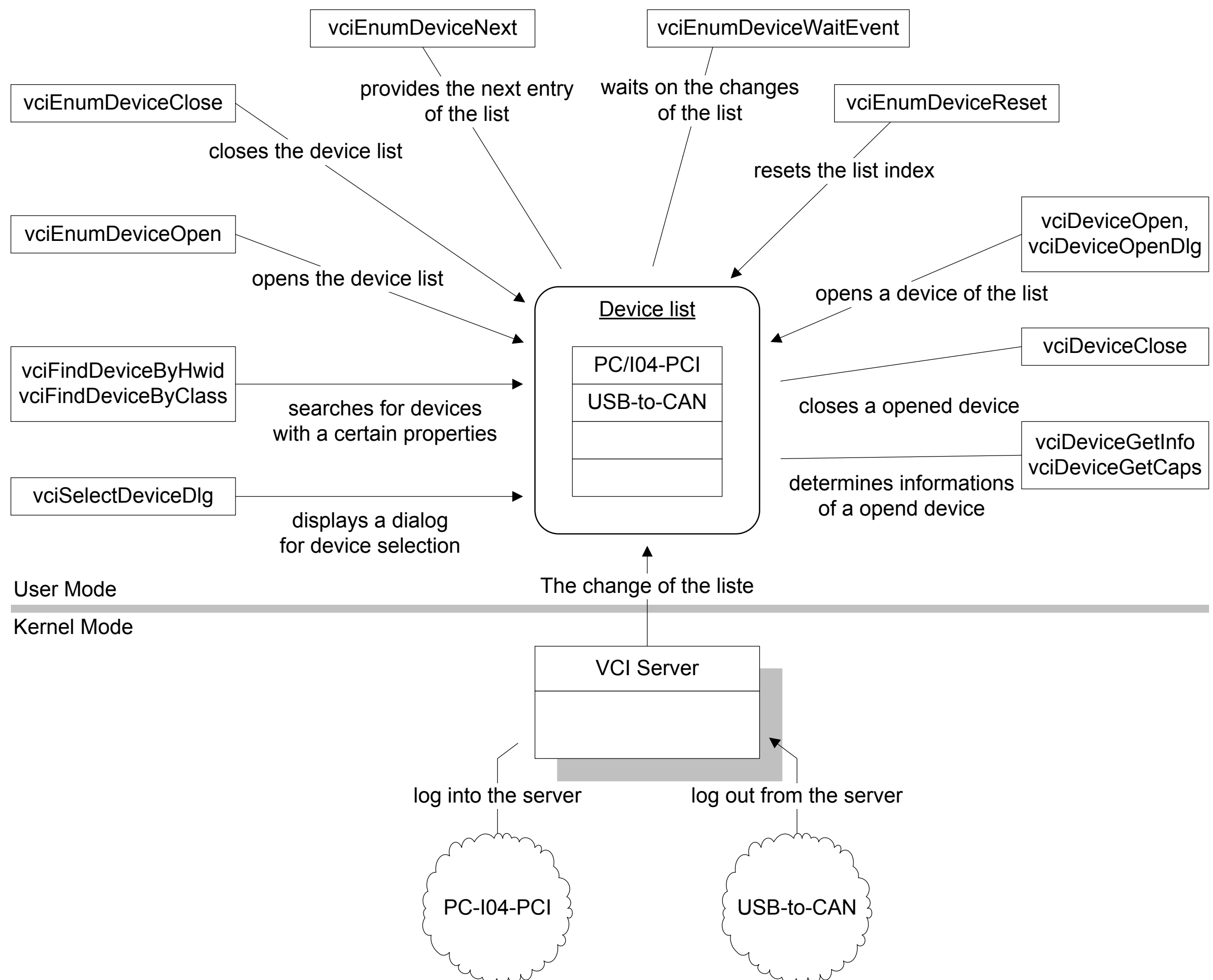


Fig. 2-1: Components and functions of the device management

The VCI server manages all CAN interface boards in a system-wide global list, which is referred to in the following as device list.

A CAN interface board logs into the server automatically when the computer is booted or when a connection is established between the computer and the CAN interface board. If a CAN interface board is no longer available, for example because the connection was interrupted, it is automatically removed from the device list.

Copyright IXXAT Automation GmbH

CAN interface boards that can be added or removed during operation, such as USB-to-CAN compact, log in when plugged into the VCI server or log out again when the CAN interface board is unplugged.

CAN interface boards also log in or out when a device driver is enabled or disabled in the device manager by the operating system (see diagram below).

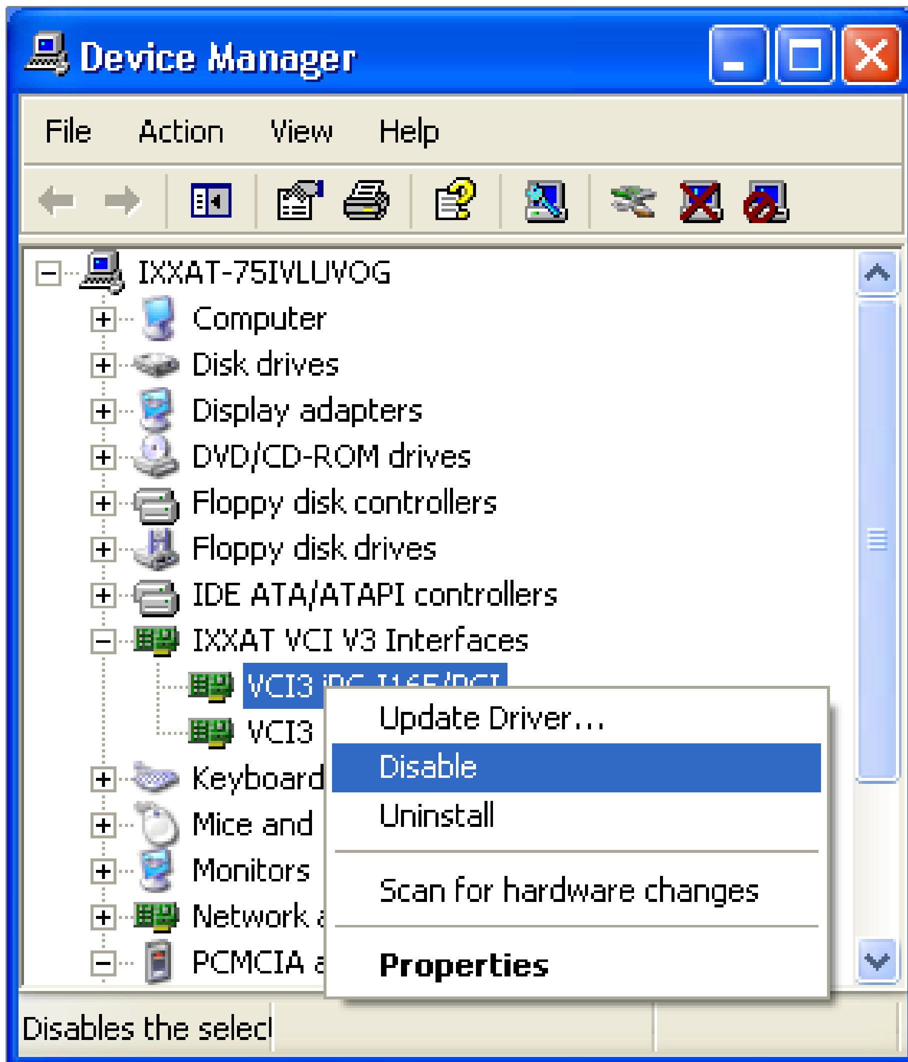


Fig. 2-2: Device manager of the operating system

2.2 List of available devices and interface boards

The VCI server manages a list of all currently available devices and interface boards in a system-wide global device list. This list is accessed by calling the function *vciEnumDeviceOpen*. If run successfully, the function returns a handle to the device list. This handle is required to access information to the individual device or interface board or to monitor changes to the device list.

For each call, the function *vciEnumDeviceNext* provides information on a device or interface board from the list. An internal index is hereby incremented, so that a further call of the function provides the information on the next device or interface board in each case. The memory required for this must provide the application in the form of a structure of type *VCIDEVICEINFO*. In the following, the main information on a device or interface board is summarized:

- *VciObjectId*: unique ID of the device. When a device or interface board logs in, it is allocated a system-wide unique ID (*VCIID*). This ID is required for later accesses to the device.
- *DeviceClass*: each device driver identifies its supported device or interface board class with a globally unique ID (GUID). Different devices or interface boards belong to different device classes. For example, IPC-I165/PCI belongs to a different class than PC-I04/PCI.
- *UniqueHardwareId*: each device has a unique hardware ID. The hardware ID can be used, for example, to differentiate between two PC-I04/PCI cards or to search for a device or interface board with a certain hardware ID.

The device list has been run through completely when the function *vciEnumDeviceNext* returns the value VCI_E_NO_MORE_ITEMS. Return values other than VCI_OK or VCI_E_NO_MORE_ITEMS, on the other hand, indicate an error.

The internal list index can be reset to the beginning with the function *vciEnumDeviceReset*, so that a subsequent call of the function *vciEnumDeviceNext* again provides the information on the first device or interface board in the list.

Applications can monitor changes to the device list via the function *vciEnumDeviceWaitEvent*. If the content of the device list changes, the function returns the value VCI_OK. Return values other than VCI_OK either indicate an error or signal that the waiting time specified for a function call has been exceeded.

The function *vciEnumDeviceClose* closes opened device list and releases the specified handle again. To save system resources, applications should always call this function if no further access to the device list is necessary.

The following section presents some functions that make it easier to search for a certain device or interface board, or make it easier to select a device than the functions described so far.

2.3 Searching for certain devices and interface boards

To search for a device or interface board with certain features, the functions *vciFindDeviceByHwid* and *vciFindDeviceByClass* are available.

The function *vciFindDeviceByHwid* searches for a device or interface board with a certain hardware ID. Each device has a unique hardware ID, which, unlike the device ID (*VCIID*), is also retained when the system is restarted. The hardware ID can therefore be saved, for example, in configuration files and enables automatic configuration after system start.

Similar functionality is also provided by the function *vciFindDeviceByClass*. This expects the device class (GUID) and the instance number of the CAN interface board searched for as parameters. An application can therefore search for the first PC-I04/PCI in the system, for example.

Applications can display a pre-defined dialog window via the function *vciSelectDeviceDlg*. With this dialog, a user can select a device or interface board from the device list. The dialog window can also be used to find the hardware ID or the device class of a device or interface board.

If run successfully, all functions return the device ID (*VCIID*) of the detected or selected device or interface board. This device ID is required for later access to the device or interface board.

2.4 Accessing a device or interface board

A device or interface board is accessed via the functions *vciDeviceOpen* or *vciDeviceOpenDlg*. If run successfully, both functions return a handle to the opened device or interface board.

The function *vciDeviceOpen* expects the device ID (*VCIID*) of the device to be opened as the input device. This ID can be determined as described in section 2.2 or 2.3.

In contrast, the function *vciDeviceOpenDlg* displays a dialog window with the current device list and provides the user with a visual possibility to select the device or interface board to be opened.

With the function *vciDeviceGetInfo*, information on the opened device or interface board can be requested. The memory required for this is provided by the application in the form of a structure of type *VCIDEVICEINFO*. The function provides the same information as the function *vciEnumDeviceNext* described in section 2.2. A more detailed description of the structure is given in section 5.1.2.

Information on the technical equipment of a device or interface board is provided by the function *vciDeviceGetCaps*. In addition to the handle of the device or interface board, the function requires the address of a structure of type *VCIDEVICECAPS*. If run successfully, the function returns the required information in this structure.

The information provided by *vciDeviceGetCaps* shows how many bus connections there are on one device or interface board. The diagram below shows a CAN interface board with two bus connections.

Copyright IXXAT Automation GmbH

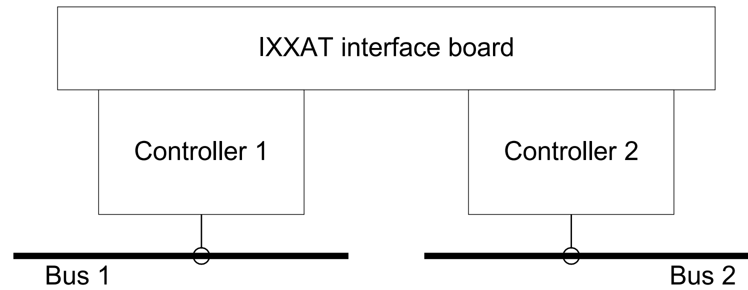


Fig. 2-3: IXXAT interface board with two bus connections.

The structure *VCIDEVICECAPS* contains a table with up to 32 entries, which describe the individual bus connection or controller. The table entry 0 describes bus connection type 1, table entry 1 describes that of bus connection 2 and so on. Further information on the structure *VCIDEVICECAPS* is given in section 5.1.3.

The function *vciDeviceClose* closes an opened device or interface board and releases its handle again. To save system resources, applications should always call this function if no further access to the device or interface board is necessary.

In addition to the above-mentioned functions, the device or interface board handle is also required for access to the bus connections. More information on this is given in section 3.

Copyright IXXAT Automation GmbH

3 Accessing the bus

3.1 Accessing the CAN bus

3.1.1 Overview

In the following, a typical CAN interface board with two connections for CAN 1 and CAN 2 is shown.

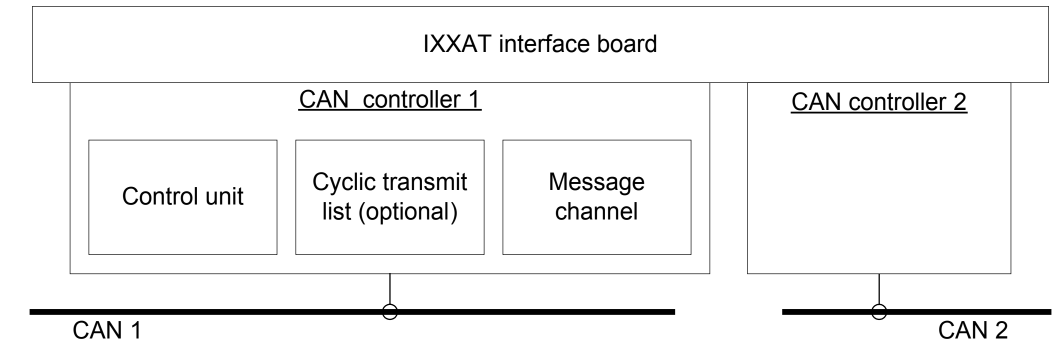


Fig. 3-1: CAN interface board with two CAN connections.

In addition to the CAN connections, a CAN interface board can also have other types of bus connections. However, these are not relevant in the following.

As shown in Fig. 3-1 for connection 1, each CAN connection consists of up to three different components. A control unit, one or more message channels and where appropriate a cyclic transmit list. The control unit and the message channels are always available. The cyclic transmit list is only normally found with CAN interface boards that have their own microprocessor.

The control unit of a CAN connection is accessed with the function *canControlOpen*. The function *canChannelOpen* opens a message channel and the function *canSchedulerOpen* access to the cyclic transmit list of the connection.

All three functions expect in the first parameter the handle of the CAN interface board and in the second parameter the number of the CAN connection. For connection 1, the number 0 is allocated, for connection 2 number 1 and so on.

To save system resources, the handle of the CAN interface board can be released again after opening a component. For further accesses to the connection, only the handle of the 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 board 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 board, 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.

If run successfully, all three functions return a handle to the opened component. If an error or an access conflict occurs, the functions return a corresponding error code.

If an opened component is no longer required, it can be closed again by calling one of the functions *canControlClose*, *canChannelClose* or *canSchedulerClose*.

The following sub-sections describe in more detail the possibilities offered by a CAN connection and how the CAN connection is to be used.

3.1.2 Control unit

The control unit provides functions for the configuration of the CAN controller, its transmission features and functions for the configuration of CAN message filters and to request the current controller state.

The control unit can be opened by several applications simultaneously in order to determine the status and the features of the CAN controller. But the CAN controller can be initialized only by one application at the same time. This prevents situations in which, for example, one application starts the CAN controller and another application stops it.

The control unit is opened by calling the function *canControlOpen*.

With the function *canControlClose*, an opened control unit is closed again and therefore available for other applications. A program should therefore release the control unit if it is no longer needed.

The application, that calls the function *canControlOpen* first, gets the exclusive control over the CAN controller. Before another application can get the exclusive control, all applications have to close the parallel opened control unit with the function *canControlClose*.

Copyright IXXAT Automation GmbH

3.1.2.1 Controller states

The following diagram shows the various states of a controller.

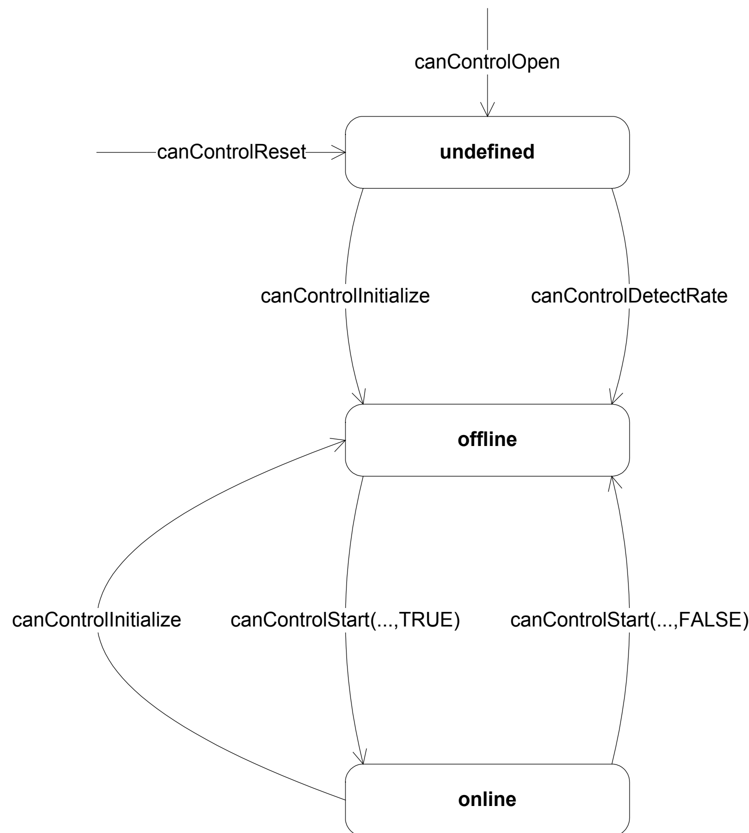


Fig. 3-2: Controller states

After opening the control unit, the controller is normally in a non-initialized state. This state is exited by calling one of the functions *canControlInitialize* or *canControlDetectBitrate*. The CAN controller is then in “offline” state.

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

With *canControlInitialize*, the operating mode and bitrate of the CAN controller are set. The values for the bus timing register in the parameters *bBtr0* and *bBtr1* correspond to the values of the registers BTR0 and BTR1 of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz.

More detailed information on setting the bitrate is given in the data sheet for SJA 1000 in section 6.5. A summary of the bus timing values with all CiA- or CANopen-compliant bitrates is given in the following table.

Bitrate (kbit)	Pre-defined constants for BTR0, BTR1	BTR0	BTR1
10	CAN_BT0_10KB, CAN_BT1_10KB	0x31	0x1C
20	CAN_BT0_20KB, CAN_BT1_20KB	0x18	0x1C
50	CAN_BT0_50KB, CAN_BT1_50KB	0x09	0x1C
100	CAN_BT0_100KB, CAN_BT1_100KB	0x04	0x1C
125	CAN_BT0_125KB, CAN_BT1_125KB	0x03	0x1C
250	CAN_BT0_250KB, CAN_BT1_250KB	0x01	0x1C
500	CAN_BT0_500KB, CAN_BT1_500KB	0x00	0x1C
800	CAN_BT0_800KB, CAN_BT1_800KB	0x00	0x16
1000	CAN_BT0_1000KB, CAN_BT1_1000KB	0x00	0x14

If the CAN connection is connected to a running system whose bitrate is unknown, the current bitrate can be determined with the function *canControlDetectBitrate*. The bus timing values determined by the function can then be applied into the function *canControlInitialize*.

The CAN controller is started or stopped by calling the function *canControlStart*. When the function is successfully called with the value TRUE in the parameter *fStart*, the controller is in "online" state. In this state the CAN controller is actively connected to the bus. Incoming CAN messages are forwarded to all active message channels, or transmit messages are output to the bus by the message channels.

Calling the function *canControlStart* with the value FALSE in the parameter *fStart* switches the CAN controller to "offline" mode. The message transfer is interrupted and the controller disabled.

The function *canControlReset* switches the CAN controller to "not initialized" status. In addition, the function resets the controller hardware and the set message filters. Please note that resetting the controller hardware leads to false message telegrams on the bus if a transmit process is aborted in the middle of transmission when a function is called.

Copyright IXXAT Automation GmbH

3.1.2.2 Message filters

Each control unit has a two-stage message filter. Messages received are filtered only according to their ID (CAN-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 has been transmitted on the bus. In this case the message filter is bypassed.

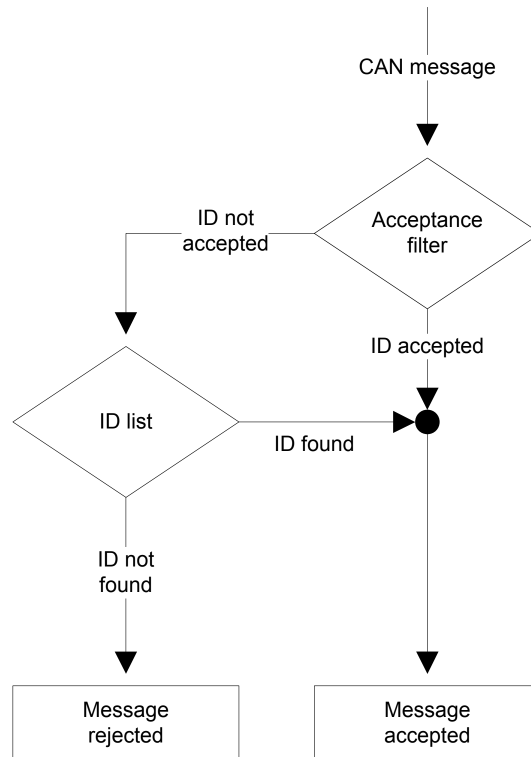


Fig. 3-3: Filter mechanism

The first filter stage, consisting of an acceptance filter, compares the ID of a received message with a binary bit sample. If the ID correlates with the set bit sample, the message is accepted, otherwise it is passed to the second filter stage. The second filter stage consists of a list with registered IDs. If the ID corresponds to the message of an ID in the list, the message is also accepted, otherwise it is rejected.

The CAN controller has separate filters for 11-bit and 29-bit Ids, which are independent of each other. When the controller is reset or initialized, the filters are set so that all messages are let through.

The filter settings can be changed with the functions *canControlSetAccFilter*, *canControlAddFilterIds* and *canControlRemFilterIds*. The functions expect two bit samples as input values in the parameters *dwCode* and *dwMask* that define the ID or the group of IDs that the filter lets through. However, a call of the functions is only successful when the controller is in "offline" state.

The bit samples in the parameters *dwCode* and *dwMask* define which IDs the filter lets through. The value of *dwCode* defines the bit sample of the ID, whereas *dwMask* defines which bits in *dwCode* are used for the comparison. If a bit in *dwMask* has the value 0, the corresponding bit in *dwCode* is not used for the comparison. If on the other hand it has the value 1, it is relevant for the comparison.

With the 11-bit filter, only the lower 12 bits are relevant. With the 29-bit filter, bits 0 to 29 are used. The other bits must be set to 0 before calling the functions.

The following tables show the relationship between the bits in the parameters *dwCode* and *dwMask* and the bits of the message ID (CAN ID):

- Meaning of the bits for the 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

- Meaning of the bits for the 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 11 to 1 or 29 to 1 correspond to the ID bits 10 to 0 or 28 to 0. Bit 0 always corresponds to the Remote Transmission Requests bit (RTR) of a message.

The following example shows the parameter values for *dwCode* and *dwMask*, in order to accept only the messages in the range 100h to 103h, with RTR-bit not set (= 0):

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

Copyright IXXAT Automation GmbH

As the example shows, only individual IDs or groups of IDs can be activated with the simple acceptance filter. However, if the required IDs do not correspond to a certain bit sample, the acceptance filter quickly reaches its limits. This is where the second filter stage with the ID list comes into play. Each list can contain up to 2048 IDs, or 4096 entries.

Individual IDs or groups of IDs can be added to the list via the function *canControlAddFilterIds* and removed again with the function *canControlRemFilterIds*. The parameters *dwCode* and *dwMask* have the same format as with the acceptance filter.

If the function *canControlAddFilterIds* is called, for example, with the values from the previous example, the function adds the IDs 100h to 103h to the list. If only one single ID is to be added when the function is called, the required ID (including RTR-bit) is entered in *dwCode* and *dwMask* set to FFFh or 3FFFFFFFh.

The acceptance filter can be completely blocked by calling the function *canControlSetAccFilter* if the value CAN_ACC_CODE_NONE is entered for *dwCode* and the value CAN_ACC_MASK_NONE for *dwMask*. Further filtering thereafter is only carried out based on the ID list. Calling the function with the values CAN_ACC_CODE_ALL and CAN_ACC_MASK_ALL, on the other hand, opens the acceptance filter completely. The ID list therefore has no effect in this case.

3.1.3 Message channel

A message channel is generated or opened by calling the function *canChannelOpen*. The parameter *fExclusive* defines whether the connection is to be used exclusively. If the value TRUE is entered here, no further message channels can be opened after the function has been successfully run. If the CAN connection is not used exclusively, any number of message channels can theoretically be created.

In the case of exclusive use of the connection, the message channel is directly connected to the CAN controller. The following diagram shows this configuration.

Copyright IXXAT Automation GmbH

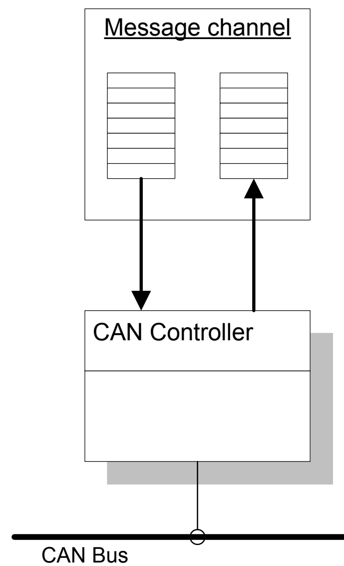


Fig. 3-4: Exclusive use of a CAN connection

In the case of non-exclusive use of the connection (*fExclusive* = FALSE), a distributor is connected between the controller and the message channels. The distributor forwards incoming messages from the CAN controller to all message channels and transfers the transmit messages of the channels to the controller. The messages are distributed in such a way that no channel receives preferential treatment. The following diagram shows a configuration with three channels on one CAN connection.

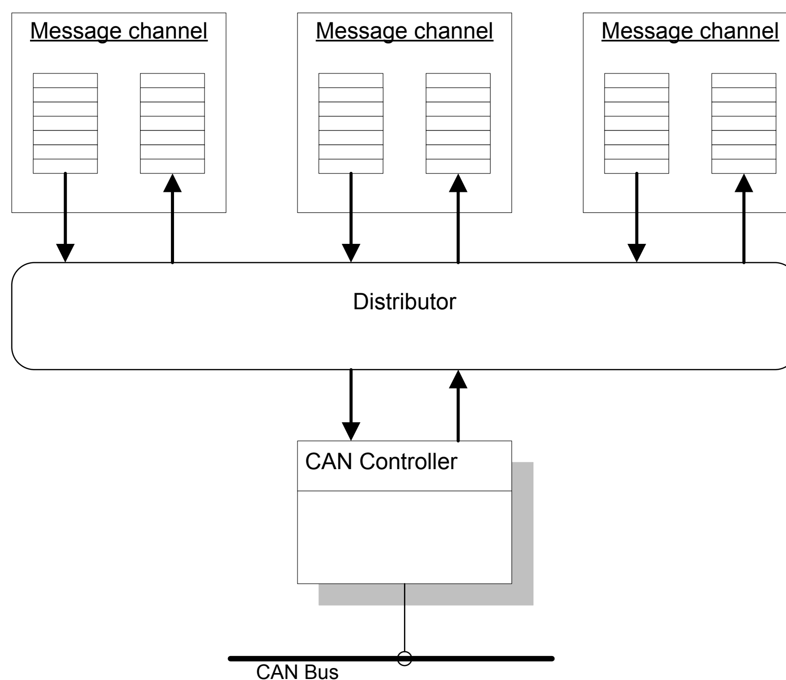


Fig. 3-5: CAN message distributor

Copyright IXXAT Automation GmbH

After a message channel has been opened, it must first be initialized. This is done with the function *canChannelInitialize*. The function expects as input parameters the size of the receive and transmit buffers in number of CAN messages and the threshold values for the receive or transmit event.

The memory reserved for the receive and transmit buffers comes from a limited system memory pool. The individual buffers of a message channel should therefore not contain more than approx. 2000 messages.

The receive event of a channel is triggered when the receive buffer contains at least the number of messages specified in *wRxThreshold*. The transmit event is triggered when the transmit buffer has at least sufficient space for the number of messages specified in *wTxThreshold*.

If the channel has been created, it can be enabled or disabled with the function *canChannelActivate*. The channel is enabled when the function is called with the value TRUE in the parameter *fEnable*. The channel is disabled by calling the function with the value FALSE in the parameter *fEnable*. A channel is disabled after opening as the default setting.

Messages are only received by the CAN controller or sent to it if the channel is active. In addition, the CAN controller must be started, otherwise no message transfer takes place between the CAN bus and the channel. The message filter of the CAN controller also has an influence on the received messages. More information on this is given in section 3.1.2.

A message channel is closed with the function *canChannelClose*. The function should always be called when a channel is no longer needed.

3.1.3.1 Reception of CAN messages

The simplest way of reading received messages from the receive buffer is to call the function *canChannelReadMessage*. If there are no messages available in the receive buffer, the function waits until either a new message is received or the waiting time specified in the parameter *dwMsTimeout* has expired. Every message has a associated message type (data, info, error, status, timer-overrun). (see §5.2.5 CANMSGINFO)

The function *canChannelPeekMessage* also reads the next message from the receive buffer. In contrast to *canChannelReadMessage*, however, the function does not wait for a new message but returns immediately to the calling program with a corresponding error code if no message is available in the receive buffer.

It is possible to wait for a new receive message or the occurrence of the receive event with the function *canChannelWaitRxEvent*. The receive event is triggered when the receive buffer contains at least the number of messages specified in *wRxThreshold* when *canChannelInitialize* is called.

The following code fragment shows a possible use of the functions *canChannelWaitRxEvent* and *canChannelPeekMessage*.

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

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

  return 0;
}
```

Note that the thread procedure only ends when the function *canChannelWaitRxEvent* returns an error code not equal to VCI_OK. However, when correctly called, all message channel-specific functions only return an error code not equal to VCI_OK when a serious problem has occurred.

A program-controlled "normal" abort of the thread procedure from the previous example does not therefore appear to be possible. However, it is possible to close the handle of the message channel from another thread, where all currently outstanding function calls or all new calls end with an error code not equal to VCI_OK. However, the disadvantage of this is that any transmit threads running simultaneously are also affected.

3.1.3.2 Transmission of CAN messages

The easiest way of transmitting CAN messages to the bus is to call the function *canChannelSendMessage*. The function waits until sufficient space is available in the message channel and then writes the message in a free entry in the transmit buffer. If it was possible to enter the message in the transmit buffer in the time specified in *dwMsTimeout*, the function returns the value VCI_OK. If the specified time has elapsed without the message being written in the transmit buffer, the function returns the value VCI_E_TIMEOUT.

The function *canChannelPostMessage* also writes a CAN message in the transmit buffer. Unlike *canChannelSendMessage*, however, the function does not wait until sufficient space is available in the transmit buffer but returns to the calling program with an error code if no free entry is available.

It is possible to wait for the occurrence of transmit events with the function *canChannelWaitTxEvent*. The transmit event is triggered when the transmit buffer has at least enough space for the number of messages specified in *wTxThreshold* when *canChannelInitialize* is called.

The following code fragment shows a possible use of the functions *canChannelWaitTxEvent* and *canChannelPostMessage*.

```
HRESULT hResult;
HANDLE  hChannel;
CANMSG  sCanMsg;
.
.
.
hResult = canChannelPostMessage(hChannel, &sCanMsg);

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

3.1.3.3 Delayed transmission of CAN messages

Connections in which the bit CAN_FEATURE_DELAYEDTX is set in the field *dwFeatures* of the structure *CANCAPABILITIES* support the delayed transmission of CAN messages.

Delayed transmission of messages can, for example, prevent a device connected to the CAN bus from receiving too much data in too short a time, which can lead to data loss with “slow” devices.

To transmit a CAN message with a delay, the minimum time in ticks is entered in the field *dwTime* of the structure *CANMSG* that must elapse before the message is forwarded to the CAN controller. The value 0 does not trigger delayed transmission, the maximum possible delay time is given in the field *dwDtxMaxTicks* of the structure *CANCAPABILITIES*. The resolution of a tick in seconds is calculated from the values in the fields *dwClockFreq* and *dwDtxDivisor* of the structure *CANCAPABILITIES* according to the following formula:

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

The specified delay time only represents a minimum, as it cannot be guaranteed that the message can be transmitted to the bus after the time elapses. It must also be noted that when using several message channels on one CAN connection the specified time values cannot generally be observed, as the distributor processes all channels simultaneously. Applications that require an exact time sequence must therefore use the CAN connection exclusively.

3.1.4 Cyclic transmit list

With the optionally available cyclic transmit list, up to 16 messages per CAN connection can be transmitted cyclically, i.e. recurrently at certain time intervals. Here it is possible that after each transmit process a certain part of a CAN message is automatically incremented.

As with the control unit, access to the cyclic transmit list is also restricted to one single application. It cannot therefore be used by more than one program simultaneously.

The transmit list is opened by calling the function *canSchedulerOpen*. If the function returns a corresponding "access denied" error code, the transmit list is already being used by another program. An opened transmit list is closed again and released for other applications with the function *canSchedulerClose*.

A message object is added to the list with the function *canSchedulerAddMessage*. In addition to the handle of the list, the function expects a pointer to a structure of type *CANCYCLICTXMSG* that specifies the transmit object that is to be added to the list. If run successfully, the function returns the list index of the added transmit object.

The cycle time of a transmit object is specified in number of ticks in the field *wCycleTime* of the structure *CANCYCLICTXMSG*. The value in this field must be greater than 0 and must not exceed the value in the field *dwCmsMaxTicks* of the structure *CANCAPABILITIES*.

The duration of a tick, or the cycle time t_z of the transmit list can be calculated with the fields *dwClockFreq* and *dwCmsDivisor* of the structure *CANCAPABILITIES* according to the following formula.

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

The transmit task of the cyclic transmit list divides the time available to it into individual sections, so-called time slots. The duration of a time slot corresponds to the duration of a tick or the cycle time. The number is shown in the field *dwCmsMaxTicks* of the structure *CANCAPABILITIES*.

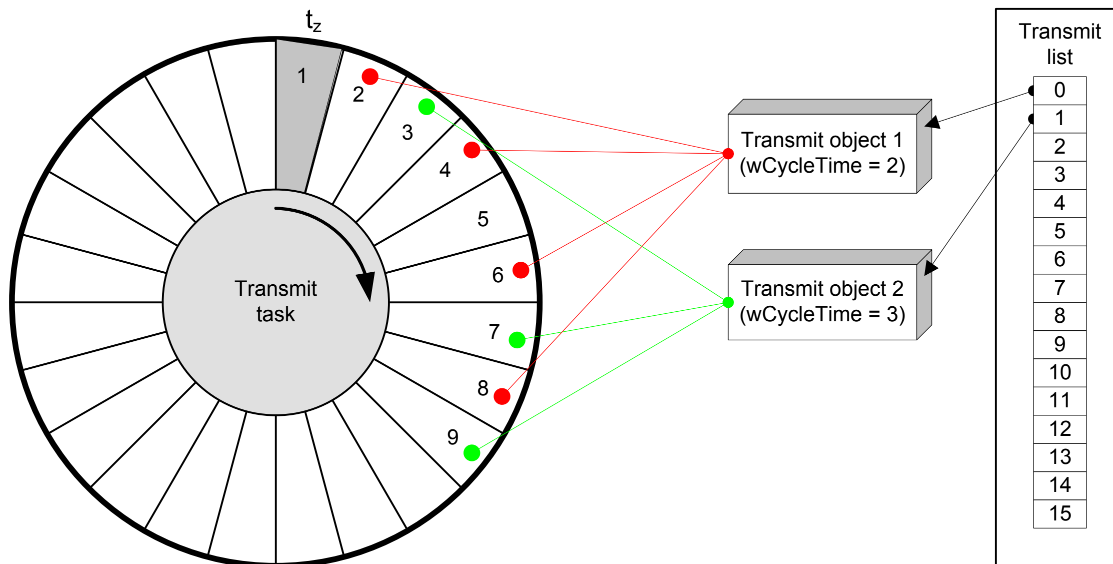


Fig. 3-6: Transmit task of the cyclic transmit list

The transmit task can always send only one message per tick. A time slot therefore contains only one transmit object. If the first transmit object is created with a cycle time of 1, all time slots are occupied and no further objects can be created. The more transmit objects are created, the longer their cycle time must be. The rule for this is: the sum of all 1/*wCycleTime* must be less than one. If, for example, a message is to be transmitted every 2 ticks and another message every 3 ticks, then 1/2 + 1/3= 5/6 = 0.833 and thus a permissible value.

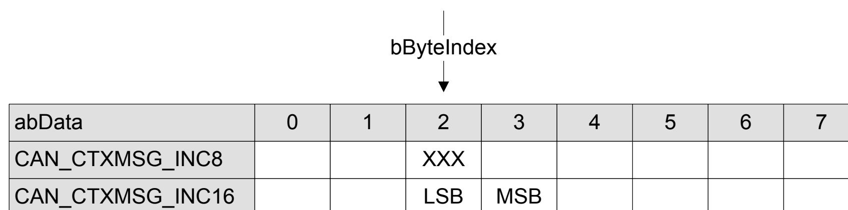
The example in Fig. 3-6 shows two transmit objects with the cycle times 2 and 3. When creating transmit object 1, the time slots 2, 4, 6, 8 etc. are occupied. When subsequently creating the second object (cycle time = 3), collisions occur in the time slots 6, 12, 18 etc., as these time slots are already occupied by object 1.

Such collisions are resolved by the transmit task in that it uses the next free time slot in each case. Object 2 from the previous example thus occupies the time slots 3, 7, 9, 13, 19 etc. The cycle time of the second object is thus not always observed exactly, which leads to an inaccuracy in the example of ± 1 tick.

The time accuracy with which the individual objects are transmitted also depends on the general bus load, as the time of transmission becomes increasingly inaccurate as the bus load increases. Generally, accuracy decreases with increasing bus load, shorter cycle times and increasing number of transmit objects.

The field *bIncrMode* of the structure *CANCYCLICTXMSG* defines whether a part of the message is automatically incremented after each transmit process. If the value CAN_CTXMSG_INC_NO is specified here, the contents remain unchanged. With the value CAN_CTXMSG_INC_ID, the field *dwMsgId* of the message is automatically incremented by 1 after each transmit process. If the field *dwMsgId* reaches the value 2048 (11-bit ID) or 536.870.912 (29-bit ID), there is an automatic overrun to 0.

With the value CAN_CTXMSG_INC_8 or CAN_CTXMSG_INC_16 in the field *bIncrMode*, an individual 8- or 16-bit value is incremented in the data field *abData[]* of the message. The field *bByteIndex* of the structure *CANCYCLICTXMSG* defines the index of the data field. With 16-bit values, the least significant byte (LSB) is in the data field *abData[bByteIndex]* and the most significant byte (MSB) in the field *abData[bByteIndex+1]*. If the value 255 (8-bit) or 65535 (16-bit) is reached, there is an overrun to 0.



abData	0	1	2	3	4	5	6	7
CAN_CTXMSG_INC8			XXX					
CAN_CTXMSG_INC16			LSB	MSB				

Fig. 3-7: Auto-increment of data fields

A transmit object can be removed from the list again with the function *canSchedulerRemMessage*. In addition to the handle of the transmit list, the function expects the list index of the object to be removed provided by the function *canSchedulerAddMessage*.

A newly created transmit object is first in idle state and is not transmitted by the transmit task until it is started by calling the function *canSchedulerStartMessage*. The transmit process for an object can be stopped with the function *canSchedulerStopMessage*.

The current status of the transmit task and of all created transmit objects is provided by the function *canSchedulerGetStatus*. The memory required for this is provided by the application in the form of a structure of type *CANSCHEDULERSTATUS* . After the function has been successfully run, the status of the transmit list and transmit objects is given in the fields *bTaskStat* and *abMsgStat*.

To determine the status of an individual transmit object, the list index provided by the function *canSchedulerAddMessage* is used as an index in the table *abMsgStat*, i.e. *abMsgStat[Index]* contains the status of the object with the specified index.

Normally, the transmit task is disabled after opening the transmit list. The transmit task does not transmit any messages in the disabled state, even if the list contains created and started transmit objects.

The transmit task of a transmit list can be enabled or disabled by calling the function *canSchedulerActivate*.

The function can be used to start all transmit objects simultaneously by first starting all transmit objects with *canSchedulerStartMessage* and only then enabling the transmit task. It is also possible to stop all transmit objects simultaneously, by simply disabling the transmit task.

The transmit list can be reset with the function *canSchedulerReset*. The function stops the transmit task and removes all registered transmit objects from the specified cyclic transmit list.

3.2 Accessing the LIN bus

3.2.1 Overview

The following diagram shows an example of an IXXAT interface board with one LIN and two CAN connections.

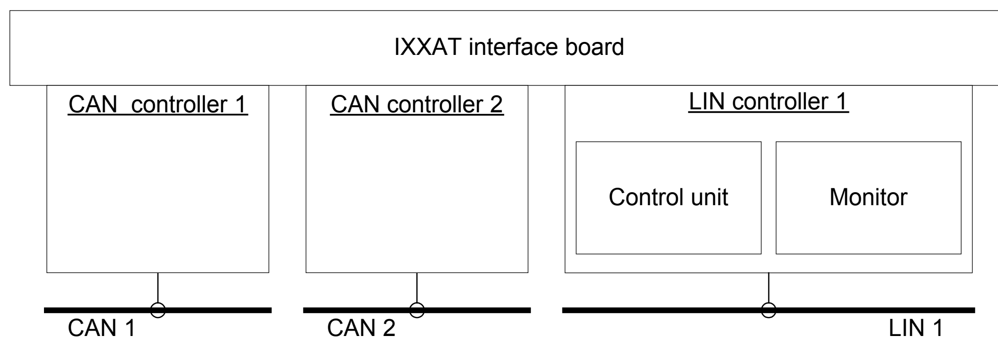


Fig. 3-8: IXXAT interface board with one LIN and 2 CAN connections.

As shown in Fig. 3-8 for the LIN connection 1, this consists of a control unit and one or more message monitors. The CAN connections available in addition to the LIN connection do not play a role in the following description.

The control unit of the LIN connection is accessed with the function *linControlOpen*. The function *linMonitorOpen* opens a message monitor.

Both functions expect the handle of the device or of the interface board in the first parameter and in the second parameter the number of the LIN connection. For connection 1 the number 0 is defined, for connection 2 the number 1 and so on.

To save system resources, after opening a component the handle of the device or of the interface board can be released again. For further accesses to the connection, only the handle of the control unit or of the message monitor is required.

The functions *linControlOpen* and *linMonitorOpen* can be called in such a way that a dialogue window is presented to the user where he can select the device or the interface board or the LIN connection. This is done by entering the value 0xFFFFFFFF for the connection number. In this case, the functions expect the handle of the higher order window (parent) or the value NULL, if no higher order window is available, in the first parameter and not the handle of the device or the interface board.

If run successfully, all three functions return a handle to the opened component. If an error or an access conflict occurs, the functions return a corresponding error code.

If an opened component is no longer required, it can be closed again by calling one of the functions *linControlClose* and *linMonitorClose*.

The possibilities offered by a LIN connection, or how the LIN connection is to be used, is described in more detail in the following sub-sections.

3.2.2 Control unit

The control unit provides functions for configuring the LIN controller and its transmission properties as well as functions for requesting the current controller status.

The component is designed in such a way that it can always only be opened by one application. Simultaneous opening of the component by different programs is not possible. This prevents situations where, for example, an application wants to start the LIN controller but another wants to stop it.

The control unit is opened by calling the function *linControlOpen*. If the function returns an error code “Access denied”, the LIN controller is already being used by another program.

This generally only represents a problem if an application cannot be run without direct control by the LIN controller. In all other cases, however, the application should be able to continue working normally.

With the function *linControlClose*, an opened control unit is closed again and is therefore available for other applications. A program should thus only release a control unit when it is no longer required.

Copyright IXXAT Automation GmbH

3.2.2.1 Controller states

The following diagram shows the various states of a controller.

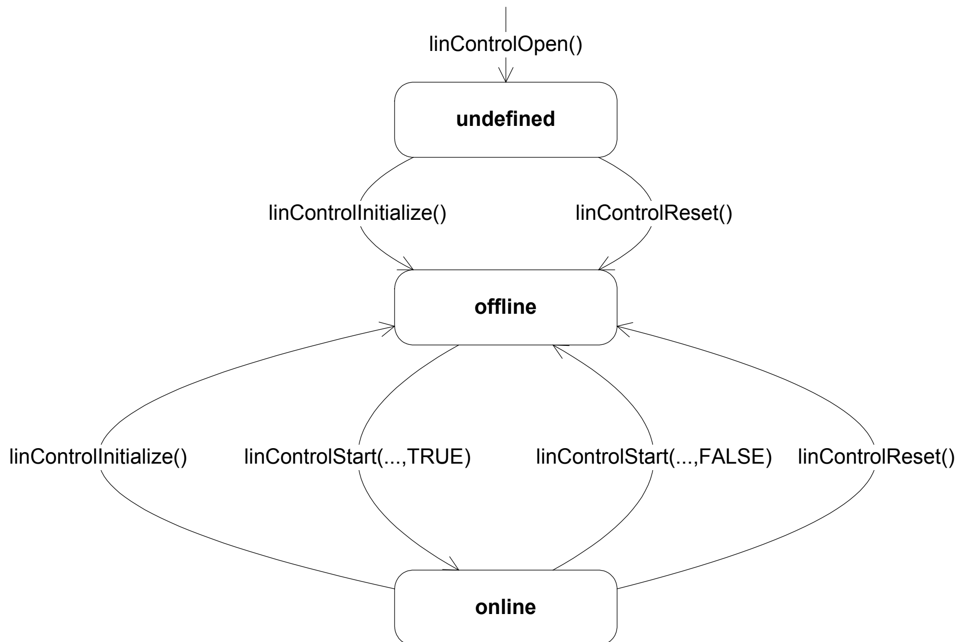


Fig. 3-9: Controller states

After opening the control unit, the controller is normally in an uninitialized state. This state is exited by calling the function *linControlInitialize*. Then the LIN controller is in "offline" state.

With *linControlInitialize*, the operating mode and the bitrate of the LIN controller are set . The bitrate in bits per second is defined in the parameter *wBitrate*. Valid values for the bitrate are between 1000 and 20000, or between LIN_BITRATE_MIN and LIN_BITRATE_MAX. If the connection supports automatic bitrate detection, this can be activated with LIN_BITRATE_AUTO in the field *wBitrate*. The following table shows some recommended bitrates:

Slow (Bit/Sec)	Medium (Bit/Sec)	Fast (Bit/Sec)
2400	9600	19200

The LIN controller is started or stopped by calling the function *linControlStart*. After successfully calling the function with the value TRUE in the parameter *fStart*, the controller is in "online" state. In this state the LIN controller is actively connected to the bus. Incoming LIN messages are forwarded to all active message monitors.

Calling the function *linControlStart* with the value FALSE in the parameter *fStart* switches the LIN controller "offline". Message transport is interrupted and the controller deactivated.

The function *linControlReset* also switches the LIN controller to the "offline" state. In addition, the function resets the controller hardware. Note that faulty telegrams may be produced on the bus if a transmit process is interrupted during transmission when the function is called.

3.2.2.2 Transmission of LIN messages

With the function *linControlWriteMessage*, messages can either be transmitted directly or entered in a response table in the controller. For better understanding, please refer to the following diagram.

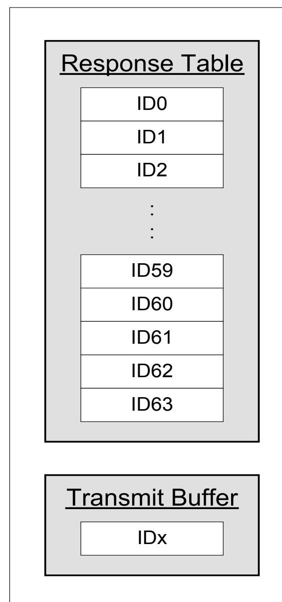


Fig. 3-10: Internal structure of the control unit

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 on the bus which is assigned to it and has been send by the master, it transmits the response data entered in the table at the corresponding position. The contents of the table can be changed or updated with the function *linControlWriteMessage* by entering the value FALSE in the parameter *fSend*. The message with the response data in the field *abData* of the structure *LINMSG* is transferred to the function in the parameter *pLinMsg*. Note that the message is of type LIN_MSGTYPE_DATA and contains a valid ID in the range 0 to 63. The table must be initialized before the controller is started, irrespective of the operating mode (master or slave), but can be updated at any time without stopping the controller. The response table is emptied when the function *linControlReset* is called.

With the function *linControlWriteMessage* messages can also be transmitted directly to the bus. For this, *fSend* must be set to the value TRUE. In this case the message is not entered in the response table but instead in the transmit buffer and transmitted to the bus, as soon as it is free, by the controller.

If the connection is operated as a master, in addition to the control messages LIN_MSGTYPE_SLEEP and LIN_MSGTYPE_WAKEUP, data messages of type LIN_MSGTYPE_DATA can also be transmitted directly.

If the connection is configured as a slave, only LIN_MSGTYPE_WAKEUP messages can be transmitted. With all other message types, the function returns an error code.

Messages of type LIN_MSGTYPE_SLEEP generate a Goto-Sleep frame on the bus, messages of type LIN_MSGTYPE_WAKEUP on the other hand a WAKEUP frame. Further information on this is given in the LIN specification in the section "Network Management".

In the master mode, the function *linControlWriteMessage* is also used to send IDs. For this, a LIN_MSGTYPE_DATA message with a valid ID and data length is transmitted, where the bit *uMsgInfo.Bits.ido* simultaneously has the value 1. Further information is given in section 5.3.4 in the description of the data structure *LINMSGINFO*.

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

3.2.3 Message monitor

A message monitor is generated or opened by calling the function *linMonitorOpen*. The parameter *fExclusive* defines whether the connection is to be used exclusively. If the value TRUE is entered here, no further message monitors can be opened after successful running of the function. If the LIN connection is not used exclusively, any number of message monitors can in principle be created. This is only limited by the installed memory.

In the case of exclusive use of the connection, the message monitor is directly connected to the LIN controller. The following diagram shows this configuration.

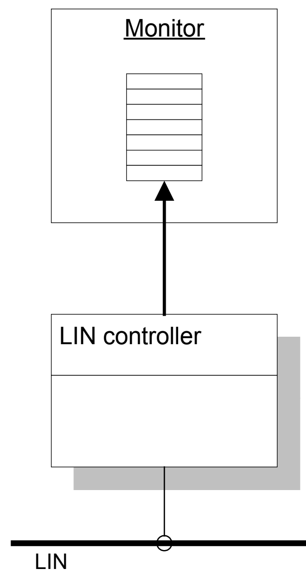


Fig. 3-11: Exclusive use of a LIN connection

In the case of non-exclusive use of the connection (*fExclusive* = FALSE), a distributor is connected between the controller and the message monitors.

The distributor forwards incoming messages from the LIN controller to all monitors. The messages are distributed in such a way that no monitor is treated preferentially. The following diagram shows a configuration with three monitors on one LIN connection.

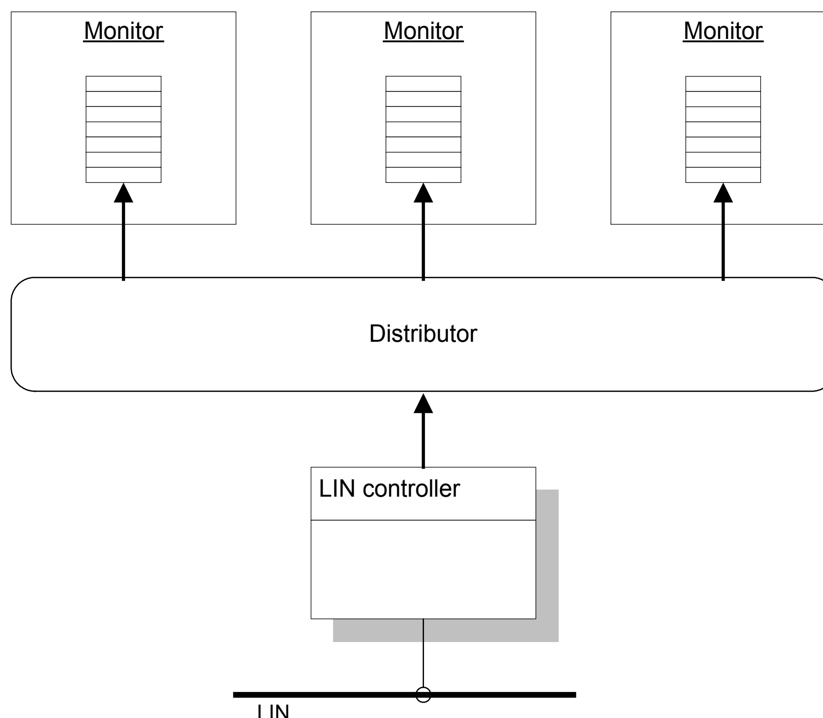


Fig. 3-12: LIN message distributor

After a message monitor is opened, it must be initialized. This is done with the function *linMonitorInitialize*. The function expects the size of the receive buffer in number of LIN messages and the threshold values for the receive event as the input parameters.

The memory reserved for the receive buffer comes from a limited system memory pool. The individual buffers of a message monitor should therefore not consist of more than approx. 2000 messages.

The receive event of a monitor is triggered when the receive buffer contains at least the number of messages defined in *wRxThreshold*.

When the monitor is configured, it can be activated or deactivated with the function *linMonitorActivate*. The monitor is activated when the function is called with the value TRUE in the parameter *fEnable*. When the function is called with the value FALSE in the parameter *fEnable*, the monitor is deactivated. In the default setting, a message monitor is deactivated when opened.

Messages are only received by the LIN controller when the message monitor is active. In addition, the LIN controller must be started, otherwise no message transfer between the LIN bus and the monitor takes place.

A message monitor is closed with the function *linMonitorClose*. The function should always be called when the monitor is no longer required.

3.2.3.1 Reception of LIN messages

The simplest way of reading received messages from the receive buffer is to call the function *linMonitorReadMessage*. If there are no messages available in the receive buffer, the function waits until a new message is received from the bus or the waiting time defined in the parameter *dwMsTimeout* has expired.

The function *linMonitorPeekMessage* reads the next message from the receive buffer. Unlike *linMonitorReadMessage*, however, the function does not wait for a new message but returns immediately to the calling program with a corresponding error code if no messages are available in the receive buffer.

With the function *linMonitorWaitRxEvent* it is possible to wait for a new receive message, or for the occurrence of the receive event. The receive event is triggered when the receive buffer contains at least the number of messages defined in *wThreshold* when *linMonitorInitialize* is called.

The following code fragment shows one possible use of the functions *linMonitorWaitRxEvent* and *linMonitorReadMessage*.

```
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;
}
```

Note that the thread procedure only ends when the function *linMonitorWaitRxEvent* returns an error code not equal to VCI_OK. However, when called correctly, all message monitor-specific functions only return an error code not equal to VCI_OK when a serious problem has occurred.

A program-controlled normal abort of the thread procedure from the above example therefore appears to be impossible. However, it is possible to close the handle of the message monitor from another thread, whereby all currently outstanding function calls or all new calls end with an error code unequal to VCI_OK.

4 Interface description

4.1 General functions

4.1.1 vciInitialize

The function initializes the VCI for the calling process. The complete syntax of the function is:

```
HRESULT VCIAPI vciInitialize ( void );
```

Parameter:

The function has no parameters.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.1.2 vciFormatError

The function converts a VCI error code into a text that can be read by users, or into a character string. The complete syntax of the function is:

```
void VCIAPI vciFormatError (
        HRESULT hrError,
        PCHAR   pszText);
```

Parameters:

hrError

[in] Error code that is to be converted into text.

pszText

[out] Pointer to a buffer for the text string. The buffer must provide space for at least VCI_MAX_ERRSTRLEN characters. The function saves the error text including a final 0 character in the specified memory area.

Return value:

The function has no return value.

4.1.3 vciDisplayError

The function displays a message window on the screen in accordance with the specified error code. The complete syntax of the function is:

```
void VCIAPI vciDisplayError (
        HWND    hwndParent,
        PCHAR   pszCaption,
        HRESULT hrError);
```

Parameters:

hwndParent
[in] Handle of the higher order window. If the value ZERO is specified here, the message window has no higher order window.

pszCaption
[in] Pointer to a 0-terminated character string with the text for the tile line of the message window. If the value ZERO is specified here, a pre-defined title line text is displayed.

hrError
[in] Error code for which the message is to be displayed.

Return value:

The function has no return value.

Comments:

If the value –1 is given in the parameter *hrError*, the function determines the last error code that occurred with the API function *GetLastError* and displays a corresponding message window. If the value 0 is given in the parameter *hrError*, no message window is displayed.

4.1.4 vciGetVersion

The function determines the version of the installed VCI. The complete syntax of the function is:

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

Parameters:

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

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

Copyright IXXAT Automation GmbH

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The function can be called at the beginning of a program to check whether the current VCI of the application is sufficient.

To get the complete VCI version number A.B.C.D please read the Version-Resource of the vciapi.dll. (Windows API: GetVersionInfo)

4.1.5 vciLuidToChar

The function converts a locally unique ID (*VCIID*) to a character string. The complete syntax of the function is:

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

Parameters:

rVciid

[in] Reference to the locally unique VCI ID to be converted into a character string.

pszLuid

[out] Pointer to a buffer for the 0-terminated character string. If run successfully, the function saves the converted VCI ID in the memory area specified here. The buffer must provide space for at least 17 characters including the final 0-character.

cbSize

[in] Size of the buffer specified in *pszLuid* in bytes.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise one of the following error codes:

VCI_E_INVALIDARG	The parameter *pszLuid* points to an invalid buffer
VCI_E_BUFFER_OVERFLOW	The buffer specified in *pszLuid* is not large enough for the character string.

4.1.6 vciCharToLuid

The function converts a 0-terminated character string to a locally unique VCI ID (*VCIID*). The complete syntax of the function is:

```
HRESULT VCIAPI vciCharToLuid (
       PCHAR  pszLuid
       PVCIID pVciid );
```

Parameters:

pszLuid

[in] Pointer to the 0-terminated character string to be converted.

pVciid

[out] Address of a variable of type *VCIID*. If run successfully, the function returns the converted ID in this variable.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise one of the following error codes:

VCI_E_INVALIDARG	Parameter *pszLuid* or *pVciid* points to an invalid buffer.
VCI_E_FAIL	The character string specified in *pszLuid* could not be converted into a valid ID.

4.1.7 vciGuidToChar

The function converts a globally unique ID (GUID) into a character string. The complete syntax of the function is:

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

Parameters:

rGuid

[in] Reference to the globally unique ID that is to be converted into a character string.

pszGuid

[out] Pointer to the buffer for the 0-terminated character string. If run successfully, the function saves the converted GUID in the specified memory area. The buffer must have space for at least 39 characters including the final 0-character.

cbSize

[in] Size of the in *pszGuid* specified buffer in Bytes.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise one of the following error codes:

VCI_E_INVALIDARG	The parameter *pszLuid* points to an invalid buffer
VCI_E_BUFFER_OVERFLOW	The buffer specified in *pszLuid* is not large enough for the character string.

4.1.8 vciCharToGuid

The function converts a 0-terminated character string into a globally unique ID (GUID). The complete syntax of the function is:

```
HRESULT VCIAPI vciCharToGuid (
        PCHAR pszGuid
        PGUID pGuid );
```

Parameters:

pszGuid

[in] Pointer to the 0-terminated character string to be converted.

pGuid

[out] Address of a variable of type GUID. If run successfully, the function returns the converted ID in this variable.

Return value:

If run successfully, the function returns the value *VCI_OK*, otherwise one of the following error codes:

VCI_E_INVALIDARG	Parameter *pszGuid* or *pGuid* points to an invalid buffer.
VCI_E_FAIL	The character string specified in *pszGuid* could not be converted into a valid ID.

4.2 Functions for the device management

4.2.1 Functions for accessing the device list

4.2.1.1 vciEnumDeviceOpen

The function opens the list of all CAN interface boards registered with the VCI. The complete syntax of the function is:

```
HRESULT vciEnumDeviceOpen( PHANDLE phEnum )
```

Parameter:

phEnum

[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:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

4.2.1.2 vciEnumDeviceClose

The function closes the device list opened with the function *vciEnumDeviceOpen*. The complete syntax of the function is:

```
HRESULT vciEnumDeviceClose( HANDLE hEnum )
```

Parameter:

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

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.2.1.3 vciEnumDeviceNext

The function determines the description of a CAN interface board of the device list and increases the internal list index so that a subsequent call of the function supplies the description to the next CAN interface board.

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

Parameters:

hEnum
[in] Handle to the opened device list.

pInfo
[out] Address of a data structure of type *VCIDEVICEINFO*. If run successfully, the function saves information on the CAN interface board in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The function returns the value VCI_E_NO_MORE_ITEMS when the list does not contain any more entries.

4.2.1.4 vciEnumDeviceReset

The function resets the internal list index of the device list, so that a subsequent call of *vciEnumDeviceNext* returns the first entry of the list again. The complete syntax of the function is:

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

Parameter:

hEnum
[in] Handle of the opened device list.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

4.2.1.5 vciEnumDeviceWaitEvent

The function waits until the content of the device list has changed, or a certain waiting time has elapsed. The complete syntax of the function is:

```
HRESULT vciEnumDeviceWaitEvent(
        HANDLE hEnum
        UINT32 dwMsTimeout );
```

Parameters:

hEnum
[in] Handle of the opened device list.

dwMsTimeout
Maximum waiting time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT when the content of the device list has not changed within the specified time. With the value INFINITE (0xFFFFFFFF), the function waits until a change has occurred in the device list.

Return value:

If run successfully, the function returns the value VCI_OK. If the time period specified in the parameter *dwMsTimeout* has elapsed without the contents of the device list having changed, the function returns the error code VCI_E_TIMEOUT. In the event of an error, the function returns an error code not equal to VCI_OK or VCI_E_TIMEOUT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The contents of the device list are only changed when a CAN interface board is added or removed. To check whether the contents of the device list have changed, without blocking the calling program, the value 0 can be specified in the parameter *dwMsTimeout* when calling the program.

4.2.1.6 vciFindDeviceByHwid

The function searches for a CAN interface board with a certain hardware ID. The complete syntax of the function is:

```
HRESULT vciFindDeviceByHwid (
        REFGUID rHardwareId,
        PVCIID  pVciidDevice );
```

Parameters:

rHardwareId

[in] Reference to the unique hardware ID of the CAN interface board searched for.

pVciidDevice

[out] Address of a variable type *VCIID*. If run successfully, the function returns the device ID of the found CAN interface board in this variable.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

Each CAN interface board has a unique hardware ID, which also remains valid after a restart of the system.

Copyright IXXAT Automation GmbH

4.2.1.7 *vciFindDeviceByClass*

The function searches for a CAN interface board with a certain device class. The complete syntax of the function is:

```
HRESULT vciFindDeviceByHwid (
       REFGUID rDeviceClass,
       UINT32  dwInstNumber,
       PVCIID  pVciidDevice );
```

Parameters:

rDeviceClass
[in] Reference to the device class of the CAN interface board searched for.

dwInstNumber
[in] Instance number of the CAN interface board searched for. If more than one CAN interface board of the same class is available, this value defines the number of the CAN interface board searched for in the device list. The value 0 selects the first CAN interface board of the specified device class.

pVciidDevice
[out] Address of a variable type *VCIID*. If run successfully, the function returns the device ID of the found CAN interface board in this variable.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.2.1.8 *vciSelectDeviceDlg*

The function displays a dialog window to select a CAN interface board from the current device list on the screen. The complete syntax of the function is:

```
HRESULT vciSelectDeviceDlg (
       HWND   hwndParent,
       PVCIID pVciidDevice );
```

Parameters:

hwndParent
[in] Handle of the higher order window. If the value ZERO is specified here, the dialog window has no higher order window.

pVciidDevice

[out] Address of a variable type *VCIID*. If run successfully, the function returns the device ID of the selected CAN interface board in this variable.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. If the dialog window is closed without a CAN interface board having been selected, the function returns the error code VCI_E_ABORT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.2.2 Functions for accessing CAN interface boards

4.2.2.1 vciDeviceOpen

The function opens the CAN interface board with the specified device ID. The complete syntax of the function is:

```
HRESULT vciDeviceOpen( REFVCIID rVciidDevice, PHANDLE phDevice )
```

Parameters:

rVciidDevice

[in] Device ID of the CAN interface board to be opened.

phDevice

[out] Address of a variable of type HANDLE. If run successfully, the function returns the handle of the opened CAN interface board in this variable. In the event of an error, the variable is set to ZERO.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The ID of a CAN interface board to be opened can be determined with one of the functions from section 4.2.1.

4.2.2.2 vciDeviceOpenDlg

The function displays a dialog window to select a CAN interface board on the screen and opens the CAN interface board selected by the user. The complete syntax of the function is:

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

Parameters:

hwndParent

[in] Handle of the higher order window. If the value ZERO is specified here, the dialog window has no higher order window.

phDevice

[out] Address of a variable of type HANDLE. If run successfully, the function saves the handle of the selected and opened CAN interface board in this variable. In the event of an error the variable is set to ZERO.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. If the dialog window is closed without a CAN interface board having been selected, the function returns the error code VCI_E_ABORT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

4.2.2.3 vciDeviceClose

The function closes an opened CAN interface board. The complete syntax of the function is:

```
HRESULT vciDeviceClose( HANDLE hDevice )
```

Parameter:

hDevice

[in] Handle of the CAN interface board to be closed. The handle specified here must come from a call of one of the functions *vciDeviceOpen* or *vciDeviceOpenDlg*.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Copyright IXXAT Automation GmbH

Comments:

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

4.2.2.4 vciDeviceGetInfo

The function determines general information on a CAN interface board. The complete syntax of the function is:

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

Parameters:

hDevice

[in] Handle of the opened CAN interface board.

pInfo

[out] Address of a structure of type *VCIDEVICEINFO*. If run successfully, the function saves information on the CAN interface board in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *VCIDEVICEINFO* in section 5.1.2.

4.2.2.5 vciDeviceGetCaps

The function determines information on the technical equipment of a CAN interface board. The complete syntax of the function is:

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

Parameters:

hDevice

[in] Handle of the opened CAN interface board.

Copyright IXXAT Automation GmbH

pCaps

[out] Address of a structure of type *VCIDEVICECAPS*. If run successfully, the function saves the information on the technical equipment in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *VCIDEVICECAPS* in section 5.1.3.

4.3 Functions for CAN access

The following sections describe the functions provided by VCI for accessing the CAN connections of a CAN interface board. Introductory information on CAN access is given in section 3.

4.3.1 Control unit

The interfaces provides functions for configuration and control of a CAN controller and for setting message filters. General information on the controller interface is given in section 3.1.2.

4.3.1.1 canControlOpen

The function opens the control unit of a CAN connection on a CAN interface board. The complete syntax of the function is:

```
HRESULT canControlOpen(
        HANDLE  hDevice,
        UINT32  dwCanNo,
        PHANDLE phControl )
```

Parameters:

hDevice

[in] Handle of the CAN interface board.

dwCanNo

[in] Number of the CAN connection of the control unit to be opened. The value 0 selects the first connection, the value 1 the second connection and so on.

Copyright IXXAT Automation GmbH

phControl

[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:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.3.1.2 canControlClose

The function closes an opened CAN controller. The complete syntax of the function is:

```
HRESULT canControlClose( HANDLE hControl )
```

Parameter:

hControl

[in] Handle of the CAN controller to be closed. The handle specified here must come from a call of the function *canControlOpen*.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.3.1.3 canControlGetCaps

The function determines the features of a CAN connection. The complete syntax of the function is:

```
HRESULT canControlGetCaps (
        HANDLE           hControl,
        PCANCAPABILITIES pCaps );
```

Parameters:

hControl
[in] Handle of the opened CAN controller.

pCaps
[out] Pointer to a structure of type *CANCAPABILITIES*. If run successfully, the function saves the features of the CAN connection in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *CANCAPABILITIES* in section 5.2.1.

4.3.1.4 canControlGetStatus

The function determines the current settings and the current status of the controller of a CAN connection. The complete syntax of the function is:

```
HRESULT canControlGetStatus (
        HANDLE         hControl,
        PCANLINESTATUS pStatus );
```

Parameters:

hControl
[in] Handle of the opened CAN controller.

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

Copyright IXXAT Automation GmbH

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *CANLINESTATUS* in section 5.2.2.

4.3.1.5 canControlDetectBitrate

This function determines the current bitrate of the bus to which the CAN connection is connected. The complete syntax of the function is:

```
HRESULT canControlDetectBitrate (
        HANDLE hControl,
        UINT16 wTimeoutMs,
        UINT32 dwCount,
        PUINT8 pabBtr0,
        PUINT8 pabBtr1,
        PINT32 plIndex );
```

Parameters:

hControl

[in] Handle of the opened CAN controller.

wTimeoutMs

[in] Maximum waiting time in milliseconds between two messages on the bus.

dwCount

[in] Number of elements in the bit timing tables *pabBtr0* or *pabBtr1*.

pabBtr0

[in] Pointer to a table with the values to be tested for the bus timing register 0. The value of an entry corresponds to the BT0 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz. The table must contain at least *dwCount* elements.

pabBtr1

[in] Pointer to a table with the values to be tested for the bus timing register 1. The value of an entry corresponds to the BT1 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz. The table must contain at least *dwCount* elements.

plIndex

[out] Pointer to a variable of type INT32. If run successfully, the function returns the table index of the found bit timing values in this variable.

Copyright IXXAT Automation GmbH

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function.

Comments:

Further information on the bus timing values in the tables *pabBtr0* and *pabBtr1* is given in the datasheet of the Philips SJA 1000 CAN controller.

To detect the bitrate, the CAN controller is operated in "List 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 *wTimeoutMs*, the function returns the value VCI_E_TIMEOUT.

If run successfully, the function receives the variables to which the parameter *plIndex* shows the index (including 0) of the found values in the bus timing tables. The corresponding table values can then be used to initialize the CAN controller with thee function *canControlInitialize*.

The function can be called in the undefined and stopped status. Further information on this is given in section 3.1.2.1.

Copyright IXXAT Automation GmbH

4.3.1.6 canControlInitialize

The function sets the operating mode and bitrate of a CAN connection. The complete syntax of the function is:

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

Parameters:

hControl

[in] Handle of the opened CAN controller.

bMode

[in] Operating mode of the CAN controller. For the operating mode, a combination of the following constants can be specified:

CAN_OPMODE_STANDARD:
Accepts CAN messages with 11-bit identifiers.

CAN_OPMODE_EXTENDED:
Accepts CAN messages with 29-bit identifiers.

CAN_OPMODE_LISTONLY:
CAN controller is operated in "List only" mode.

CAN_OPMODE_ERRFRAME:
Errors are indicated to the application via special CAN messages.

CAN_OPMODE_LOWSPEED:
CAN controller uses low speed coupling.

bBtr1

[in] Value for the bus timing register 0 of the CAN controller. The value corresponds to the BTR0 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz.

bBtr1

[in] Value for the bus timing register 1 of the CAN controller. The value corresponds to the BTR1 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function.

Comments:

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.

Further information on the bus timing values in the parameters *bBtr0* and *bBtr1* is given in the datasheet of the Philips SJA 1000 CAN controller.

The following table shows the bus timing values for all bitrates specified by the CiA or CANopen.

Bitrate (kbit)	Pre-defined constants for BTR0, BTR1	BTR0	BTR1
10	CAN_BT0_10KB, CAN_BT1_10KB	0x31	0x1C
20	CAN_BT0_20KB, CAN_BT1_20KB	0x18	0x1C
50	CAN_BT0_50KB, CAN_BT1_50KB	0x09	0x1C
100	CAN_BT0_100KB, CAN_BT1_100KB	0x04	0x1C
125	CAN_BT0_125KB, CAN_BT1_125KB	0x03	0x1C
250	CAN_BT0_250KB, CAN_BT1_250KB	0x01	0x1C
500	CAN_BT0_500KB, CAN_BT1_500KB	0x00	0x1C
800	CAN_BT0_800KB, CAN_BT1_800KB	0x00	0x16
1000	CAN_BT0_1000KB, CAN_BT1_1000KB	0x00	0x14

4.3.1.7 canControlReset

The function resets the controller hardware and the set message filters of a CAN connection. The complete syntax of the function is:

```
HRESULT canControlReset ( HANDLE hControl );
```

<u>Parameter:</u>

hControl

[in] Handle of the opened CAN controller.

<u>Return value:</u>

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function.

<u>Comments:</u>

The function resets the controller hardware, removes the set acceptance filter, deletes the contents of the filter lists and switches the controller "not initialized". At the same time, the message flow between the controller and the message channels connected to it 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.

Further information is given in section 3.1.2.

Copyright IXXAT Automation GmbH

4.3.1.8 canControlStart

The function starts or stops the controller of a CAN connection. The complete syntax of the function is:

```
HRESULT canControlStart ( HANDLE hControl, BOOL fStart );
```

Parameters:

hControl
[in] Handle of the opened CAN controller.

fStart
[in] The value TRUE starts and the value FALSE stops the CAN controller.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function.

Comments:

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. Calling this function resets the time stamp. 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.

Copyright IXXAT Automation GmbH

4.3.1.9 *canControlSetAccFilter*

The function sets the 11- or 29-bit acceptance filter of a CAN connection. The complete syntax of the function is:

```
HRESULT canControlSetAccFilter (
        HANDLE hControl,
        BOOL   fExtended,
        UINT32 dwCode,
        UINT32 dwMask );
```

Parameters:

hControl
[in] Handle of the opened CAN controller.

fExtended
[in] Selection of the acceptance filter. The 11-bit acceptance filter is selected with the value FALSE and the 29-bit acceptance filter with the value TRUE.

dwCode
[in] Bit sample of the identifier(s) to be accepted including RTR-bit.

dwMask
[in] Bit sample of the relevant bits in *dwCode*. If a bit has the value 0 in *dwMask*, the corresponding bit in *dwCode* is not used for the comparison. If on the other hand it has the value 1, it is relevant for the comparison.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

A detailed description of how the filter works and of the values for the parameters *dwCode* and *dwMask* is given in section 3.1.2.2.

Copyright IXXAT Automation GmbH

4.3.1.10 canControlAddFilterIds

The function enters one or more IDs (CAN-IDs) in the 11- or 29-bit filter list of a CAN connection. The complete syntax of the function is:

```
HRESULT canControlAddFilterIds (
        HANDLE hControl,
        BOOL   fExtended,
        UINT32 dwCode,
        UINT32 dwMask);
```

Parameters:

hControl
[in] Handle of the opened CAN controller.

fExtended
[in] Selection of the filter list. The 11-bit filter list is selected with the value FALSE and the 29-bit filter list with the value TRUE.

dwCode
[in] Bit sample of the identifier(s) to be identified including RTR-bit.

dwMask
[in] Bit sample of the relevant bits in *dwCode*. If a bit has the value 0 in *dwMask*, the corresponding bit in *dwCode* is ignored. If on the other hand it has the value 1, it is relevant.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

A detailed description of how the filter works and of the values for the parameters *dwCode* and *dwMask* is given in section 3.1.2.2.

Copyright IXXAT Automation GmbH

4.3.1.11 canControlRemFilterIds

The function removes one or more IDs (CAN-IDs) from the 11- or 29-bit filter list of a CAN connection. The complete syntax of the function is:

```
HRESULT canControlRemFilterIds (
        HANDLE hControl,
        BOOL   fExtendend,
        UINT32 dwCode,
        UINT32 dwMask );
```

Parameters:

hControl
[in] Handle of the opened CAN controller.

fExtended
[in] Selection of the filter list. The 11-bit filter list is selected with the value FALSE and the 29-bit filter list with the value TRUE.

dwCode
[in] Bit sample of the identifier(s) to be removed including RTR-bit.

dwMask
[in] Bit sample of the relevant bits in *dwCode*. If a bit has the value 0 in *dwMask*, the corresponding bit in *dwCode* is ignored. If on the other hand it has the value 1, it is relevant.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

A detailed description of how the filter works and of the values for the parameters *dwCode* and *dwMask* is given in section 3.1.2.2.

4.3.2 Message channel

The interfaces provides functions to configure message channels between the application and the CAN bus . General information on the message channels is given in section 3.1.3.

4.3.2.1 canChannelOpen

The function opens or creates a message channel for a CAN connection of a CAN interface board. The complete syntax of the function is:

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

Parameters:

hDevice
[in] Handle of the CAN interface board.

dwCanNo
[in] Number of the CAN connection for which a message channel is to be opened. The value 0 selects the first connection, the value 1 the second connection and so on.

fExclusive
[in] Defines whether the connection is used exclusively for the channel to be opened. If the value TRUE is specified here, the CAN connection is used exclusively for the new message channel. With the value FALSE, more than one message channel can be opened for the CAN connection.

phChannel
[out] Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened CAN message channel in this variable. In the event of an of an error, the variable is set to ZERO.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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.

Copyright IXXAT Automation GmbH

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

If the message channel is no longer required, the handle returned in *phChannel* should be released again with the function *canChannelClose*.

4.3.2.2 canChannelClose

The function closes an opened message channel. The complete syntax of the function is:

```
HRESULT canChannelClose( HANDLE hChannel )
```

Parameter:

hChannel

[in] Handle of the message channel to be closed. The handle specified here must come from a call of the function *canChannelOpen* .

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.3.2.3 canChannelGetCaps

The function determines the features of a CAN connection. The complete syntax of the function is:

```
HRESULT canChannelGetCaps (
        HANDLE           hChannel,
        PCANCAPABILITIES pCaps );
```

Parameters:

hChannel

[in] Handle of the opened message channel.

Copyright IXXAT Automation GmbH

pCaps

[out] Pointer to a structure of type *CANCAPABILITIES*. If run successfully, the function saves the features of the CAN connection in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *CANCAPABILITIES* in section 5.2.1.

4.3.2.4 canChannelGetStatus

The function 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. The complete syntax of the function is:

```
HRESULT canChannelGetStatus (
        HANDLE         hChannel,
        PCANCHANSTATUS pStatus );
```

Parameters:

hChannel

[in] Handle of the opened message channel.

pStatus

[out] Pointer to a structure of type *CANCHANSTATUS*. If run successfully, the function saves the current status of the channel and controller in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *CANCHANSTATUS* in section 5.2.3.

4.3.2.5 canChannelInitialize

The function initializes the receive and transmit buffers of a message channel. The complete syntax of the function is:

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

Parameters:

hChannel
[in] Handle of the opened message channel.

wRxFifoSize
[in] Size of the receive buffer in number of CAN messages.

wRxThreshold
[in] Threshold value for the receive event. The event is triggered when the number of messages in the receive buffer reaches or exceeds the number specified here.

wTxFifoSize
[in] Size of the transmit buffer in number of CAN messages.

wTxThreshold
[in] Threshold value for the transmit event. The event is triggered when the number of free entries in the transmit buffer reaches or exceeds the number specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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.

Copyright IXXAT Automation GmbH

4.3.2.6 canChannelActivate

The function activates or deactivates a message channel. The complete syntax of the function is:

```
HRESULT canChannelActivate ( HANDLE hChannel, BOOL fEnable );
```

Parameters:

hChannel

[in] Handle of the opened message channel.

fEnable

With the value TRUE, the function activates the message flow between the CAN controller and the message channel, with the value FALSE the function deactivates the message flow.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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. Further information on this is given in the description of the function *canControlStart* and in section 3.1.2.

After activation of the channel, messages can be written in the transmit buffer with *canChannelPostMessage* or *canChannelSendMessage* in the transmit buffer, or read from the receive buffer with the functions *canChannelPeekMessage* and *canChannelReadMessage*.

4.3.2.7 canChannelPeekMessage

The function reads the next CAN message from the receive buffer of a message channel. The complete syntax of the function is:

```
HRESULT canChannelPeekMessage(
        HANDLE  hChannel,
        PCANMSG pCanMsg );
```

Parameters:

hChannel

[in] Handle of the opened message channel.

pCanMsg
[out] Pointer to a structure of type *CANMSG*. If run successfully, the function saves the read CAN message in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK. If no CAN message is available in the receive buffer when the function is called, the function returns the value VCI_E_RXQUEUE_EMPTY. If the function control fails for other reasons, the function returns an error code not equal to VCI_OK or VCI_E_RXQUEUE_EMPTY. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

If the value ZERO is specified in the parameter *pCanMsg*, the function removes the next CAN message from the receive buffer.

4.3.2.8 canChannelPostMessage

The function writes a CAN message in the transmit buffer of the specified message channel. The complete syntax of the function is:

```
HRESULT canChannelPostMessage (
        HANDLE  hChannel,
        PCANMSG pCanMsg );
```

Parameters:

hChannel
[in] Handle of the opened message channel.

pCanMsg
[in] Pointer to an initialized structure of type *CANMSG* with the CAN message to be transmitted.

Return value:

If run successfully, the function returns the value VCI_OK. If no space is available in the transmit buffer when the function is called, the function returns the value VCI_E_TXQUEUE_FULL. If the function control fails for other reasons, the function returns an error code not equal to VCI_OK or VCI_E_TXQUEUE_FULL. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The function does not wait until the message has been transmitted on the bus.

4.3.2.9 canChannelWaitRxEvent

The function waits until the receive event has occurred or until a certain waiting time has elapsed. The complete syntax of the function is:

```
HRESULT canChannelWaitRxEvent (
        HANDLE hChannel
        UINT32 dwMsTimeout );
```

Parameters:

hChannel

[in] Handle of the opened message channel.

dwMsTimeout

Maximum waiting time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT if the receive event has not occured in the time specified here. With the value INFINITE (0xFFFFFFFF), the function waits until the receive event has occured.

Return value:

If run successfully, the function returns the value VCI_OK. If the time period specified in the parameter *dwMsTimeout* has elapsed without the receive event occurring, the function returns the error code VCI_E_TIMEOUT. In the event of an error, the function returns an error code not equal to VCI_OK or VCI_E_TIMEOUT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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 *dwMsTimeout* when calling the function.

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

Copyright IXXAT Automation GmbH

4.3.2.10 canChannelWaitTxEvent

The function waits until the transmit event has occurred, or a certain waiting time has elapsed. The complete syntax of the function is:

```
HRESULT canChannelWaitTxEvent (
        HANDLE hChannel
        UINT32 dwMsTimeout );
```

Parameters:

hChannel

[in] Handle of the opened message channel.

dwMsTimeout

Maximum waiting time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT if the transmit event has not occurred within the time specified here. With the value INFINITE (0xFFFFFFFF), the function waits until the transmit event has occured.

Return value:

If run successfully, the function returns the value VCI_OK. If the time period specified in the parameter *dwMsTimeout* has elapsed without the transmit event having occured, the function returns the error code VCI_E_TIMEOUT. In the event of an error, the function returns an error code not equal to VCI_OK or VCI_E_TIMEOUT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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 *dwMsTimeout* when the function is called.

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

Copyright IXXAT Automation GmbH

4.3.2.11 canChannelReadMessage

The function reads the next CAN message from the receive buffer of a message channel. The complete syntax of the function is:

```
HRESULT canChannelReadMessage(
        HANDLE  hChannel,
        UINT32  dwMsTimeout,
        PCANMSG pCanMsg );
```

Parameters:

hChannel

[in] Handle of the opened message channel.

dwMsTimeout

Maximum waiting time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT if no message was read or received within the specified time. With the value INFINITE (0xFFFFFFFF), the function waits until a message has been read.

pCanMsg

[out] Pointer to a structure of type *CANMSG*. If run successfully, the function saves the read CAN message in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK. If the time period specified in the parameter *dwMsTimeout* has elapsed without a message having been received, the function returns the error code VCI_E_TIMEOUT. In the event of an error, the function returns an error code not equal to VCI_OK or VCI_E_TIMEOUT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

If the value ZERO is specified in the parameter *pCanMsg*, the function removes the next CAN message from the receive buffer.

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

4.3.2.12 canChannelSendMessage

The function waits until a message channel is ready to receive a message and then writes the specified CAN message in the transmit buffer of the message channel. The complete syntax of the function is:

```
HRESULT canChannelSendMessage (
        HANDLE  hChannel,
        UINT32  dwMsTimeout,
        PCANMSG pCanMsg );
```

Parameters:

hChannel
[in] Handle of the opened message channel.

dwMsTimeout
Maximum waiting time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT if the message could not be entered in the transmit buffer within the specified time. With the value INFINITE (0xFFFFFFFF), the function waits until the transmit event occurs and the messagehas been written in the transmit buffer.

pCanMsg
[in] Pointer to an initialized structure of type *CANMSG* with the CAN message to be transmitted.

Return value:

If run successfully, the function returns the value VCI_OK,. If the time period specified in the parameter *dwMsTimeout* has elapsed without the message having been written in the transmit buffer, the function returns the error code VCI_E_TIMEOUT. In the event of an error, the function returns an error code not equal to VCI_OK or VCI_E_TIMEOUT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The function only waits until the message has been written in the transmit buffer, but not until the message has been transmitted on the bus.

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

4.3.3 Cyclic transmit list

With the transmit list, up to 16 different CAN message objects can be transmitted cyclically. These interfaces provides functions to configure, start and stop the cyclic transmit messages.

4.3.3.1 canSchedulerOpen

The function opens the cyclic transmit list of a CAN connection on a CAN interface board. The complete syntax of the function is:

```
HRESULT canSchedulerOpen(
        HANDLE  hDevice,
        UINT32  dwCanNo,
        PHANDLE phScheduler )
```

Parameters:

hDevice
[in] Handle of the CAN interface board.

dwCanNo
[in] Number of the CAN connection of the transmit list to be opened. The value 0 selects the first connection, the value 1 the second connection and so on.

phScheduler
[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:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.3.3.2 canSchedulerClose

The function closes an opened cyclic transmit list. The complete syntax of the function is:

```
HRESULT canSchedulerClose( HANDLE hScheduler )
```

Parameter:

hScheduler

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

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.3.3.3 canSchedulerGetCaps

The function determines the features of the CAN connection of the specified cyclic transmit list. The complete syntax of the function is:

```
HRESULT canSchedulerGetCaps (
        HANDLE           hScheduler,
        PCANCAPABILITIES pCaps );
```

Parameters:

hScheduler

[in] Handle of the opened transmit list.

pCaps

[out] Pointer to a structure of type *CANCAPABILITIES*. If run successfully, the function saves the features of the CAN connection in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *CANCAPABILITIES* in section 5.2.1.

4.3.3.4 canSchedulerGetStatus

The function determines the current status of the transmit task and of all registered transmit objects of a cyclic transmit list. The syntax of the function is:

```
HRESULT canSchedulerGetStatus (
       HANDLE              hScheduler,
       PCANSCHEDULERSTATUS pStatus );
```

Parameters:

hScheduler
[in] Handle of the opened transmit list.

pStatus
[out] Pointer to a structure of type *CANSCHEDULERSTATUS* If run successfully, the function saves the current status of all cyclic transmit objects in the memory area specified here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

Further information on the data provided by the function is given in the description of the data structure *CANSCHEDULERSTATUS* in section 5.2.4.

4.3.3.5 canSchedulerActivate

The function starts or stops the transmit task of the cyclic transmit list and thus the cyclic transmit process of all currently registered transmit objects. The complete syntax of the function is:

```
HRESULT canSchedulerActivate ( HANDLE hScheduler, BOOL fEnable );
```

Parameters:

hScheduler
[in] Handle of the opened transmit list.

fEnable

With the value TRUE the function activates, and with the value FALSE deactivates, the cyclic transmit process of all currently registered transmit objects.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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.

4.3.3.6 canSchedulerReset

The function stops the transmit task and removes all transmit objects from the specified cyclic transmit list. The complete syntax of the function is:

```
HRESULT canSchedulerReset ( HANDLE hScheduler );
```

Parameter:

hScheduler

[in] Handle of the opened transmit list.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Copyright IXXAT Automation GmbH

4.3.3.7 canSchedulerAddMessage

The function adds a new transmit object to the specified cyclic transmit list. The complete syntax of the function is:

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

Parameters:

hScheduler
[in] Handle of the opened transmit list.

pMessage
[in] Pointer to an initialized structure of type *CANCYCLICTXMSG* with the transmit object.

pdwIndex
[out] Pointer to a variable of type UINT32. If run successfully, the function returns the list index of the newly added transmit object in this variable. In the event of an error, the variable is set to the value 0xFFFFFFFF (-1). This index is required for all further function calls.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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*).

4.3.3.8 canSchedulerRemMessage

The function stops processing of a transmit object and removes it from the specified cyclic transmit list. The complete syntax of the function is:

```
HRESULT canSchedulerRemMessage (
       HANDLE hScheduler,
       UINT32 dwIndex );
```

Parameters:

hScheduler
[in] Handle of the opened transmit list.

Copyright IXXAT Automation GmbH

dwIndex

[in] List index of the transmit object to be removed. The list index must come from a previous call of the function *canSchedulerAddMessage* .

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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

4.3.3.9 canSchedulerStartMessage

The function starts a transmit object of the specified cyclic transmit list. The complete syntax of the function is:

```
HRESULT canSchedulerStartMessage (
        HANDLE hScheduler,
        UINT32 dwIndex,
        UINT16 dwCount );
```

Parameters:

hScheduler

[in] Handle of the opened transmit list.

dwIndex

[in] List index of the transmit object to be started. The list index must come from a previous call of the function *canSchedulerAddMessage* .

dwCount

[in] Number of the cyclic transmit repeats. With the value 0, the transmit process is repeated infinitely. The value specified here must be in the range 0 to 65535.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

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*.

4.3.3.10 canSchedulerStopMessage

The function stops a transmit object of the specified cyclic transmit list. The complete syntax of the function is:

```
HRESULT canSchedulerStopMessage (
       HANDLE hScheduler,
       UINT32 dwIndex );
```

Parameters:

hScheduler
[in] Handle of the opened transmit list.

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

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function.

Comments:

4.4 Functions for LIN access

The following sections describe the functions provided by the VCI for accessing the LIN connections of a device or of an interface board. Introductory information on LIN access is given in section 3.2.

4.4.1 Control unit

The interface provides functions for the configuration and control of a LIN controller and to request the properties of a LIN connection. Further information on the control unit is given in section 3.2.2.

4.4.1.1 linControlOpen

The function opens the control unit of a LIN connection of a device or of an interface board. The complete syntax of the function is:

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

Parameter:

hDevice
[in] Handle of the opened device or of the interface board.

dwLinNo
[in] Number of the LIN connection of the control unit to be opened. The value 0 selects the first LIN connection, the value 1 the second LIN connection and so on.

phLinCtl
[out] Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened control unit in this variable. In the event of an error, the variable is set to NULL.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

If the value 0xFFFFFFFF is entered in the parameter *dwLinNo*, the function displays a dialogue window on the screen to select a device or an interface board and a LIN connection. In this case the function expects the handle of a higher order window or the value NULL if no higher order window is available and not the handle of the device in the parameter *hDevice*.

4.4.1.2 linControlClose

The function closes an opened LIN controller. The complete syntax of the function is:

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

Parameter:

hLinCtl
[in] Handle of the LIN control unit to be closed. The handle defined here must come from a call of the function *linControlOpen*.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

When the function is called, the handle defined in *hLinCtl* is no longer valid and may no longer be used.

Copyright IXXAT Automation GmbH

4.4.1.3 linControlGetCaps

The function determines the properties of the LIN connection of the specified control unit. The complete syntax of the function is:

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

Parameter:

hLinCtl
[in] Handle of the opened LIN control unit.

pLinCaps
[out] Pointer to a structure of type *LINCAPABILITIES*. If run successfully, the function saves the properties of the LIN connection in the memory area defined here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *LINCAPABILITIES* in section 5.3.1.

4.4.1.4 linControlGetStatus

The function determines the current settings and the current status of the controller of a LIN connection. The complete syntax of the function is:

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

Parameter:

hLinCtl
[in] Handle of the opened LIN control unit.

pStatus
[out] Pointer to a structure of type *LINLINESTATUS*. If run successfully, the function saves the current settings and the status of the controller in the memory area defined here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Copyright IXXAT Automation GmbH

Comments:

Further information on the data provided by the function is given in the description of the data structure *LINLINESTATUS* in section 5.3.2.

4.4.1.5 linControlInitialize

The function sets the operating mode and bitrate of a LIN connection. The complete syntax of the function is:

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

Parameter:

hLinCtl

[in] Handle of the opened LIN control unit.

bMode

[in] Operating mode of the LIN controller. A combination of the following constants can be entered for the operating mode:

LIN_OPMODE_SLAVE:
Slave mode. This operating mode is active by default.

LIN_OPMODE_MASTER:
Activate master mode (if supported, see also *LINCAPABILITIES*).

LIN_OPMODE_ERRORS:
Errors are indicated to the application via special LIN messages.

wBitrate

[in] Bitrate in bits per second. The value entered here must be within the limits defined by the constants LIN_BITRATE_MIN and LIN_BITRATE_MAX. If the controller is operated as a slave, the bitrate is automatically determined by the controller by entering the value LIN_BITRATE_AUTO if this is supported.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The function internally resets the controller hardware in accordance with the function *linControlReset* and then initializes the controller with the entered parameters. The function can be called from any controller status.

Copyright IXXAT Automation GmbH

4.4.1.6 linControlReset

The function resets the controller hardware of a LIN connection. The complete syntax of the function is:

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

Parameter:

hLinCtl
[in] Handle of the opened LIN control unit.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The function resets the controller hardware and switches the controller "offline". Message transport between the controller and the currently opened message monitors is then also interrupted.

When the function is called, a currently active transmit process of the controller is aborted. This can lead to transmission errors or to a faulty message telegram on the bus.

Further information is given in section 3.2.2.

4.4.1.7 linControlStart

The function starts or stops the controller of the LIN connection. The complete syntax of the function is:

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

Parameter:

hLinCtl
[in] Handle of the opened LIN control unit.

fStart
[in] The value TRUE starts and the value FALSE stops the LIN controller.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

A call of the function is only successful if the LIN controller was previously configured with the function *linControlInitialize*.

After a successful call of the function, the LIN controller is actively connected to the bus (“online”). Incoming LIN messages are then forwarded to all active message monitors, or transmit messages from the controller are transmitted on the bus.

When the function is called with the value FALSE in the parameter *fStart*, the LIN controller is switched “offline”. Message transport is then interrupted and the LIN controller is switched passively. In contrast to the function *linControlReset*, the function does not simply interrupt a current transmit process of the controller but instead ends it so that no faulty telegram is transferred to the bus.

4.4.1.8 linControlWriteMessage

This function either transmits the specified message directly to the LIN bus connected to the controller or enters the message in the response table of the controller. The complete syntax of the function is:

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

Parameter:

hLinCtl
[in] Handle of the opened LIN control unit.

fSend
[in] Defines whether the message is transmitted directly to the LIN bus or whether it is entered in the response table of the controller. With TRUE the message is transmitted directly, with FALSE the message is entered in the response table.

pLinMsg
[in] Pointer to an initialized structure of type *LINMSG* with the LIN message to be transmitted.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Detailed information on this function is given in section 3.2.2.2.

4.4.2 Message monitor

The interface provides functions to install a message monitor between an application and a LIN bus. Detailed information on message monitors is given in section 3.2.3.

4.4.2.1 linMonitorOpen

The function creates a message monitor for the LIN connection of a device or of an interface board. The complete syntax of the function is:

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

Parameter:

hDevice

[in] Handle of the opened device or of the interface board.

dwLinNo

[in] Number of the LIN connection of the control unit to be opened. The value 0 selects the first LIN connection, the value 1 the second LIN connection and so on.

fExclusive

[in] Determines whether the LIN connection is used exclusively for the monitor to be generated. If the value TRUE is entered here, no further monitors can be created after a successful call of the function until the generated monitor has been released again. With the value FALSE, any number of message monitors can be created for the LIN connection.

phLinMon

[out] Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened monitor in this variable. In the event of an error, the variable is set to the value *NULL*.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

If the value TRUE is entered in the parameter *fExclusive*, no further message monitors can be opened after a successful call of the function. This means that the first program to call the function with the value TRUE in the parameter *fExclusive* has exclusive control over the message reception of the LIN connection.

If the parameter *dwLinNo* is set to the value 0xFFFFFFFF, the function displays a dialogue window on the screen to select a device or an interface board and a LIN connection. In this case the function expects the handle of a higher order window or the value NULL if no higher order window is available and not the handle of the device in the parameter *hDevice*.

If the message monitor is no longer required, the handle returned in *phLinMon* should be released again with the function *linMonitorClose*.

4.4.2.2 linMonitorClose

The function closes an opened message monitor for the LIN connection. The complete syntax of the function is:

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

Parameter:

hLinMon

[in] Handle of the message monitor to be closed. The handle used here must come from a call of the function *linMonitorOpen*.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

When the function was called, the handle specified in *hLinMon* is no longer valid and may no longer be used.

4.4.2.3 linMonitorGetCaps

The function determines the properties of the LIN connection of the specified message monitor. The complete syntax of the function is:

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

Parameter:

hLinMon

[in] Handle of the opened LIN message monitor.

pLinCaps

[out] Pointer to a structure of type *LINCAPABILITIES*. If run successfully, the function saves the properties of the LIN connection in the memory area defined here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *LINCAPABILITIES* in section 5.3.1.

Copyright IXXAT Automation GmbH

4.4.2.4 linMonitorGetStatus

The function determines the current status of a message monitor as well as the current settings and the current status of the controller that is connected to the message monitor. The complete syntax of the function is:

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

Parameter:

hLinMon
[in] Handle of the opened LIN message monitor.

pStatus
[out] Pointer to a structure of type *LINMONITORSTATUS*. If run successfully, the function saves the current status of the monitor and controller in the memory area defined here.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

Further information on the data provided by the function is given in the description of the data structure *LINMONITORSTATUS* in section 5.3.3.

4.4.2.5 linMonitorInitialize

The function initializes the receive buffer of a message monitor. The complete syntax of the function is:

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

Parameter:

hLinMon
[in] Handle of the opened LIN message monitor.

wFifoSize
[in] Size of the receive buffer in number of LIN messages.

wThreshold
[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 defined here.

Copyright IXXAT Automation GmbH

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

A value of more than 0 must be entered for the size of the receive buffer, otherwise the function returns an error code "Invalid Parameter".

The value entered in the parameter *wFifoSize* defines the lower limit for the size of the buffers. The actual size of the buffer may in some cases be larger than the value specified, as the memory used for this is reserved page-wise.

If the function is called for an already initialized monitor, the function first deactivates the monitor, then releases the available buffer and generates a new buffer of the required size.

4.4.2.6 linMonitorActivate

The function activates or deactivates a message monitor. The complete syntax of the function is:

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

Parameter:

hLinMon

[in] Handle of the opened LIN message monitor.

fEnable

[in] With the value TRUE, the function activates the message reception between the LIN controller and the message monitor, with the value FALSE the function deactivates the message flow.

Return value:

If run successfully, the function returns the value VCI_OK, otherwise an error code not equal to VCI_OK. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The message monitor is deactivated by default when it is opened or initialized. In order for the monitor to receive messages, it must be activated. At the same time, the LIN controller must be in the "online" state. Further information on this is given in the sections 3.2.2 and 3.2.3.

After successful activation of the monitor, messages can be read out of the receive buffer with the functions *linMonitorPeekMessage* and *linMonitorReadMessage*.

Copyright IXXAT Automation GmbH

4.4.2.7 linMonitorPeekMessage

The function reads the next LIN message from the receive buffer of a monitor. The complete syntax of the function is:

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

Parameter:

hLinMon
[in] Handle of the opened LIN message monitor.

pLinMsg
[out] Pointer to a structure of type *LINMSG*. If run successfully, the function saves the read LIN messages in the memory area defined here.

Return value:

If run successfully, the function returns the value VCI_OK. If no LIN message is available in the receive buffer when the function is called, the function returns the value VCI_E_RXQUEUE_EMPTY. If the function call fails for other reasons, the function returns an error code not equal to VCI_OK or VCI_E_RXQUEUE_EMPTY. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The function returns immediately to the calling program if no message is available for reading. If the value NULL is defined in the parameter *pLinMsg*, the function removes the next LIN message from the receive buffer.

4.4.2.8 linMonitorWaitRxEvent

The function waits until the receive event has occurred, or a certain delay time has expired. The complete syntax of the function is:

```
HRESULT linMonitorWaitRxEvent (
        HANDLE hLinMon,
        UINT32 dwMsTimeout );
```

Parameter:

hLinMon
[in] Handle of the opened LIN message monitor.

dwMsTimeout
[in] Maximum delay time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT if the receive event has not occurred within the time defined here. With the value INFINITE (0xFFFFFFFF), the function waits until the recceive event has occurred.

Copyright IXXAT Automation GmbH

Return value:

If run successfully, the function returns the value VCI_OK,. If the time specified in the parameter *dwMsTimeout* has expired without the receive event occurring, the function returns the error code VCI_E_TIMEOUT. In the event of an error, the function returns an error code not equal to VCI_OK or VCI_E_TIMEOUT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

The receive event is triggered as soon as the number of messages in the receive buffer reaches the defined threshold. See also the description of the function *linMonitorInitialize*.

To check whether the receive event has already occurred without blocking the calling program, the value 0 can be entered in the parameter *dwMsTimeout* when the function is called.

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

4.4.2.9 linMonitorReadMessage

The function reads the next LIN message from the receive buffer of a monitor. The complete syntax of the function is:

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

Parameter:

hLinMon

[in] Handle of the opened LIN message monitor.

dwMsTimeout

[in] Maximum delay time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT if no message is read or received with in the defined time. With the value INFINITE (0xFFFFFFFF) the function waits until a message has been read.

pLinMsg

[out] Pointer to a structure of type *LINMSG*. If run successfully, the function saves the read LIN message in the memory area defined here.

Return value:

If run successfully, the function returns the value VCI_OK. If the time specified in the parameter *dwMsTimeout* has expired without a message being received, the function returns the error code VCI_E_TIMEOUT. In the event of an error, the function returns an error code not equal to VCI_OK or VCI_E_TIMEOUT. Further information on the error code is provided by the function *vciFormatError*.

Comments:

If the value NULL is entered in the parameter *pLinMsg*, the function removes the next LIN message from the receive buffer.

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

Copyright IXXAT Automation GmbH

5 Types and structures

5.1 VCI-specific data types

The declarations of all VCI-specific data types and constants are located in the file <vci*type.h*>.

5.1.1 VCIID

The data type describes a VCI locally unique ID. The data type has the following structure:

```
typedef union
{
  LUID  AsLuid;
  INT64 AsInt64
} VCIID, *PVCIID;
```

- *AsLuid*:
  ID in the form of an LUID. The data type LUID is defined in Windows.
- *AsInt64*:
  ID as a signed 64-bit integer.

5.1.2 VCIDEVICEINFO

The structure describes the information on a CAN interface board. The structure is as follows:

```
typedef struct _VCIDEVICEINFO
{
  VCIID  VciObjectId;          // unique VCI object identifier
  GUID   DeviceClass;          // device class identifier

  UINT8  DriverMajorVersion;   // major driver version number
  UINT8  DriverMinorVersion;   // minor driver version number
  UINT16 DriverBuildVersion;   // build driver version number

  UINT8  HardwareBranchVersion;// branch hardware version number
  UINT8  HardwareMajorVersion; // major hardware version number
  UINT8  HardwareMinorVersion; // minor hardware version number
  UINT8  HardwareBuildVersion; // build hardware version number

  union _UniqueHardwareId      // unique hardware identifier
  {
    CHAR AsChar[16];
    GUID AsGuid;
  } UniqueHardwareId;

  CHAR Description [128];      // device description (e.g: "PC-I04-PCI")
  CHAR Manufacturer[126];      // device manufacturer (e.g: "IXXAT")

  UINT16 DriverReleaseVersion; // release driver version number
} VCIDEVICEINFO, *PVCIDEVICEINFO;
```


Copyright IXXAT Automation GmbH

- *VciObjectId*:
  [out] Unique ID (*VCIID*) of the CAN interface board. The VCI system service allocates a system-wide ID and unique ID to each CAN interface board. This ID serves as a key for subsequent accesses to the CAN interface board.
- *DeviceClass*:
  [out] ID of the device class. Each device driver identifies its device class in the form of a globally unique ID (GUID). Different CAN interface boards belong to different device classes. Applications can use the device class to distinguish between an IPC-I165/PCI and a PC-I04/PCI board, for example.
- *DriverMajorVersion*:
  [out] Major version number of the device driver.
- *DriverMinorVersion*:
  [out] Minor version number of the device driver.
- *DriverBuildVersion*:
  [out] Build version number of the device driver.
- *DriverReleaseVersion*:
  [out] Release (revision) version number of the device driver.
- *HardwareBranchVersion*:
  [out] branch version number of the hardware.
- *HardwareMajorVersion*:
  [out] Major version number of the hardware.
- *HardwareMinorVersion*:
  [out] Minor version number of the hardware.
- *HardwareBuildVersion*:
  [out] build version number of the hardware.
- *UniqueHardwareId*:
  [out] Unique ID of the CAN interface board. Each CAN interface board has a unique ID that can be used to distinguish between two PC-I04/PCI cards, for example. The value can be interpreted either as a GUID or as a character string. If the first two bytes contain the characters "HW", it is an ASCII character string in accordance with ISO-8859-1 (Latin-1) with the serial number of the CAN interface board.
- *Description*:
  [out] Description of the CAN interface board as a 0-terminated ASCII character string in accordance with ISO-8859-1 (Latin-1).
- *Manufacturer*:
  [out] Manufacturer of the CAN interface board as a 0-terminated ASCII character string in accordance with ISO-8859-1 (Latin-1).

Copyright IXXAT Automation GmbH

5.1.3 VCIDEVICECAPS

The structure describes the hardware equipment of a CAN interface board. The structure is as follows:

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

- *BusCtrlCount*:
  [out] Number of available bus connections or fieldbus controllers.
- *BusCtrlTypes*:
  [out] Table with up to VCI_MAX_BUSCTRL 16-bit values, which describe the type of the individual connection. Valid entries of the table are in the range from 0 to *BusCtrlCount*-1. The upper 8 bits of each value of the table define the type of the fieldbus supported, the lower 8 bits the type of the controller used. With the macros VCI_BUS_TYPE or VCI_CTL_TYPE defined in *vcitype.h*, the bus type or the controller type can be extracted from the table values. The file *vcitype.h* also contains pre-defined constants for the possible bus and controller types.

5.2 CAN-specific data types

The declarations of all CAN-specific data types and constants are given in the file *<cantype.h>*.

5.2.1 CANCAPABILITIES

The data type describes the features of a CAN connection. The structure is as follows:

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

- *wCtrlType*:
  [out] Type of CAN controller. The value of the field corresponds to one of the CAN_TYPE_ constants defined in *<cantype.h>*.

- *wBusCoupling*:
  [out] Type of bus coupling. The following values are defined for the bus coupling:

  CAN_BUSC_LOWSPEED:
  The CAN controller has a low-speed coupling.

  CAN_BUSC_HIGHSPEED:
  The CAN controller has a high-speed coupling.
- *dwFeatures*:
  [out] Supported features. The value is a combination of one or more of the following constants:

  CAN_FEATURE_STDOREXT:
  The CAN controller supports 11-bit or 29-bit messages, but not both formats simultaneously.

  CAN_FEATURE_STDANDEXT:
  The CAN controller supports 11- and 29-bit messages simultaneously.

  CAN_FEATURE_RMTFRAME:
  The CAN controller supports Remote Transmission Request messages.

  CAN_FEATURE_ERRFRAME:
  The CAN controller returns error messages.

  CAN_FEATURE_BUSLOAD:
  The CAN controller supports calculation of the bus load.

  CAN_FEATURE_IDFILTER:
  The CAN controller allows exact filtering of messages.

  CAN_FEATURE_LISTONLY:
  The CAN controller supports the operating mode "List only".

  CAN_FEATURE_SCHEDULER:
  Cyclic transmit list available.

  CAN_FEATURE_GENERRFRM:
  The CAN controller supports the generation of error frames.

  CAN_FEATURE_DELAYEDTX:
  The CAN controller supports delayed transmission of messages.
- *dwClockFreq*:
  [out] Frequency of primary timers in Hz.
- *dwTscDivisor*:
  [out] Divisor factor of the time stamp counter. The time stamp counter provides the time stamp for CAN messages. The frequency of the time stamp counter is calculated from the frequency of the primary timer divided by the value specified here.
- *dwCmsDivisor*:
  [out] Divisor factor for the timer of the cyclic transmit list. The frequency of the timer is calculated from the frequency of the primary timer divided by the value specified here. If no cyclic transmit list is available, the field has the value 0.

Copyright IXXAT Automation GmbH

- *dwCmsMaxTicks*:
  [out] Maximum cycle time of the cyclic transmit list in timer ticks. If no cyclic transmit list is available, the field has the value 0.
- *dwDtxDivisor*:
  [out] Divisor factor for the timer used for delayed transmission of messages. The frequency of the timer is calculated from the frequency of the primary timer divided by the value specified here. If delayed transmission is not supported by the CAN interface board, the field has the value 0.
- *dwDtxMaxTicks*:
  [out] Maximum delay time of the delayed message transmitter in timer ticks. If delayed transmission is not supported by the CAN interface board, the field has the value 0.

5.2.2 CANLINESTATUS

The data type describes the current status of a CAN controller. The structure is as follows:

```
typedef struct
{
  UINT8  bOpMode;
  UINT8  bBtReg0;
  UINT8  bBtReg1;
  UINT8  bBusLoad;
  UINT32 dwStatus;
} CANLINESTATUS, *PCANLINESTATUS;
```

- *bOpMode*:
  [out] Current operating mode of the CAN controller. The value is a combination of one or more CAN_OPMODE_ constants and contains the value specified in the parameter *bMode* when the function *canControlInitialize* is called.
- *bBtReg0*:
  [out] Current value of the bus timing register 0. The value corresponds to the BTR0 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz. Further information on this is given in the datasheet of the SJA 1000.
- *bBtReg1*:
  [out] Current value of the bus timing register 1. The value corresponds to the BTR1 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz. Further information on this is given in the datasheet of the SJA 1000.
- *bBusLoad*:
  [out] Current bus load in per cent (0 to 100). The value is only valid if the bus load calculation is supported by the controller. Further information is given in the section on *CANCAPABILITIES*.

- *dwStatus*:
  [out] Current status of the CAN controller. The value is a combination of one or more of the following constants:
  CAN_STATUS_TXPEND:
  The CAN controller is currently sending a message to the bus.
  CAN_STATUS_OVRRUN:
  A data overrun in the receive buffer of the CAN controller has taken place. This status can only be reseted through a CAN controller reset (see §4.3.1.7).
  CAN_STATUS_ERRLIM:
  An overrun of an error counter of the CAN controller has taken place.
  CAN_STATUS_BUSOFF:
  The CAN controller has changed to the "BUS-OFF" status.
  CAN_STATUS_ININIT:
  The CAN controller is in the stopped status.

5.2.3 CANCHANSTATUS

The data type describes the current status of a CAN message channel. The structure is as follows:

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

- *sLineStatus*:
  [out] Current status of the CAN controller. Further information is given in the description of the data structure *CANLINESTATUS*.
- *fActivated*:
  [out] Indicates whether the message channel is currently active (TRUE) or inactive (FALSE).
- *fRxOverrun*:
  [out] Indicates with the value TRUE whether an overrun of the receive buffer of the CAN controller has taken place. (see §5.2.2 CAN_STATUS_OVRRUN).
- *bRxFifoLoad*:
  [out] Current level of the receive buffer in per cent.
- *bTxFifoLoad*:
- [out] Current level of the transmit buffer in per cent.

Copyright IXXAT Automation GmbH

5.2.4 CANSCHEDULERSTATUS

The data type describes the current status of a cyclic transmit list. The structure is as follows:

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

- *bTaskStat*:
  [out] Current status of the transmit task.

  CAN_CTXTSK_STAT_STOPPED:
  The transmit task is currently stopped, or deactivated.

  CAN_CTXTSK_STAT_RUNNING:
  The transmit task is running, or is active.

- *abMsgStat*:
  Table with the status of all 16 transmit objects. Each table entry can have one of the following values:

  CAN_CTXMSG_STAT_EMPTY:
  The entry is not allocated a transmit object, or the entry is not currently being used.

  CAN_CTXMSG_STAT_BUSY:
  The transmit object is currently being used.

  CAN_CTXMSG_STAT_DONE:
  Processing of the transmit object is completed.

Copyright IXXAT Automation GmbH

5.2.5 CANMSGINFO

The data type summarizes various information on CAN messages in a 32-bit value. The value can be addressed either byte-wise or via individual bit fields.

```
typedef union
{
  struct
  {
    UINT8  bType;
    UINT8  bReserved;
    UINT8  bFlags;
    UINT8  bAccept;
  } Bytes;

  struct
  {
    UINT32 type: 8;
    UINT32 res : 8;
    UINT32 dlc : 4;
    UINT32 ovr : 1;
    UINT32 srr : 1;
    UINT32 rtr : 1;
    UINT32 ext : 1;
    UINT32 afc : 8;
  } Bits;

} CANMSGINFO, *PCANMSGINFO;
```

The information of a CAN message can be addressed byte-wise via the element *Bytes*. The following fields are defined for this:

- *Bytes.bType*:
  [in/out] Type of the message. See also *Bits.type*.
- *Bytes.bReserved*:
  Reserved. See also *Bits.res*.
- *Bytes.bFlags*
  [in/out] Various flags. See also *Bits.dlc*, *Bits.ovr*, *Bits.srr*, *Bits.rtr* and *Bits.ext*.
- *Bytes.bAccept*:
  [out] With receive messages displays which filter has accepted the message. See also *Bits.afc*.

The information of a CAN message can be accessed bit-wise with the element *Bits*. The following bit fields are defined:

- *Bits.type*:
  [in/out] Type of the message. The types listed in the following are defined for receive messages. For transmit messages, only the message type CAN_MSGTYPE_DATA is currently defined. Other values are not permitted here.

CAN_MSGTYPE_DATA:
Normal message. All regular receive messages are of this type. The field *CANMSG.dwMsgId* contains the ID of the message, the field *CANMSG.dwTime* the time of reception. The fields *CANMSG.abData* contain according to length (see *Bits.dlc*) the data bytes of the message. With transmit messages the IDs are to be entered in the field *CANMSG.dwMsgId* and the data bytes according to length in the fields *CANMSG.abData*. The field *CANMSG.dwTime* is normally set to 0, unless the message is to be transmitted with a delay. In this case the delay time is to be specified in ticks. See also the description of the structure *CANMSG*, or section 3.1.3.3.

CAN_MSGTYPE_INFO:
Information message. This message type is entered in the receive buffers of all activated message channels with certain events or with changes to the status of the controller. The field *CANMSG.dwMsgId* of the message always has the value 0xFFFFFFFF. The field *ab CANMSG.Data[0]* contains one of the following values:

Constant	Meaning
CAN_INFO_START	The CAN controller was started. The field *CANMSG.dwTime* contains the relative start time (normally 0).
CAN_INFO_STOP	The CAN controller was stopped. The field *CANMSG.dwTime* contains the value 0.
CAN_INFO_RESET	The CAN controller was reset. The field *CANMSG.dwTime* contains the value 0.

CAN_MSGTYPE_ERROR:
Error message. This message type is entered in the receive buffers of all activated message channels when bus errors occur if the flag CAN_OPMODE_ERRFRAME was specified in the parameter *CANMSG.bMode* when the function *canControlInitialize* was called. The field *CANMSG.dwMsgId* of the message always has the value 0xFFFFFFFF. The time of the event is marked in the field *CANMSG.dwTime* of the message. The field *abData[0]* contains one of the following values:

Constant	Meaning
CAN_ERROR_STUFF	Bit stuff error
CAN_ERROR_FORM	Format error
CAN_ERROR_ACK	Acknowledge error
CAN_ERROR_BIT	Bit error
CAN_ERROR_CRC	CRC error
CAN_ERROR_OTHER	Other unspecified error

The field *CANMSG.abData[1]* of the message contains the least significant byte of the current CAN status (see also *CANLINESTATUS.dwStatus*). The contents of the other data fields are undefined.

CAN_MSGTYPE_STATUS:
Status message. This message type is entered in the receive buffers of all activated message channels when the controller status changes. The field *CANMSG.dwMsgId* of the message always has the value 0xFFFFFFFF. The time of the event is marked in the field *dwTime* of the message. The field *CANMSG.abData[0]* contains the least significant byte of the current CAN status (see also ***CANLINESTATUS****.dwStatus*). The contents of the other data fields are undefined.

CAN_MSGTYPE_WAKEUP:
Not currently used, or reserved for future extensions.

CAN_MSGTYPE_TIMEOVR:
Timer overrun. Messages of this type are generated when an overrun of the 32-bit time stamp of CAN messages occurs. The time of the event (normally 0) is given in the field *CANMSG.dwTime* of the message and the number of timer overruns after the last timer overrun message in the field *CANMSG.dwMsgId*. The contents of the data fields *abData* are undefined.

CAN_MSGTYPE_TIMERST:
Not currently used, or reserved for future extensions

- *Bits.res*:
  Reserved for future extension. For reasons of compatibility, the field should always be set to 0.
- *Bits.dlc*:
  [in/out] Data length code. Defines the number of valid data bytes in the fields *CANMSG.abData* of the message.
- *Bits.ovr*:
  [out] Data overrun. The bit is set to 1 if the receive buffer is full after this message is entered.
- *Bits.srr*:
  [in/out] Self reception request. If the bit is set for transmit messages, the message is entered in the receive buffer as soon as it has been transmitted on the bus. With receive messages, a set bit indicates that it is a received self- reception message.
- *Bits.rtr*:
  [in/out] Remote transmission request.
- *Bits.ext*:
  [in/out] Message with extended 29-bit ID.
- *Bits.afc*:
  [out] Acceptance filter code. With receive messages, this field specifies the filter that let the message through.

Copyright IXXAT Automation GmbH

5.2.6 CANMSG

The data type describes the structure of a CAN message telegram.

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

- *dwTime*:
  [in/out] With receive messages, this field contains the relative reception time of the message in ticks. The resolution of a tick can be calculated from the fields *dwClockFreq* and *dwTscDivisor* of the structure *CANCAPABILITIES* in accordance with the following formula:

  Resolution [s] = dwTscDivisor / dwClockFreq

  An overrun of the 32-Bit dwTime value is received with a Timer Overrun messages (CAN_MSGTYPE_TIMEOVR $5.2.5)

  With transmit messages, the field defines with how many ticks delay the message is to be transmitted to the bus. The delay time between the last message transmitted and the new message can be calculated with the fields *dwClockFreq* and *dwDtxDivisor* of the structure *CANCAPABILITIES* in accordance with the following formula.

  delay time [s] = (dwDtxDivisor / dwClockFreq) * dwTime

  The maximum possible delay time is defined by the field *dwDtxMaxTicks* of the structure *CANCAPABILITIES*.
- *dwMsgId*:
  [in/out] CAN ID of the message in Intel format (right-aligned) without RTR bit.
- *uMsgInfo*:
  [in/out] Bit field with information on the message type. A description of the bit field is given in section 5.2.5.
- *abData*:
  [in/out] Array for up to 8 data bytes. The number of valid data bytes is defined by the field *uMsgInfo.Bits.dlc*.

5.2.7 CANCYCLICTXMSG

The data type describes the structure of a cyclic CAN transmit message.

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

wCycleTime:
[in/out] Cycle time of the transmit message in ticks. The cycle time can be calculated with the fields *dwClockFreq* and *dwCmsDivisor* of the structure *CANCAPABILITIES* in accordance with the following formula.

cycle time [s] = (dwCmsDivisor / dwClockFreq) * wCycleTime

The maximum possible cycle time is restricted to the value in the field *dwCmsMaxTicks* of the structure *CANCAPABILITIES*.

- *bIncrMode*:
  [in/out] This field defines whether part of the cyclic transmit message is automatically incremented after every transmit process.

  CAN_CTXMSG_INC_NO:
  No automatic increment of a message field occurs.

  CAN_CTXMSG_INC_ID:
  Increments the field *dwMsgId* of the message after every transmit process by 1. If the field reaches the value 2048 (11-bit ID) or 536.870.912 (29-bit ID), there is an automatic overrun to 0.

  CAN_CTXMSG_INC_8:
  Increments an 8-bit value in the data field *abData* of the message. The data field to be incremented is defined in the field *bByteIndex*. If the maximum value 255 is exceeded, there is an automatic overrun to 0.

  CAN_CTXMSG_INC_16:
  Increments a 16-bit value in the data field *abData* of the message. The least significant byte of the 16-bit value to be incremented is defined in the field *bByteIndex*. The most significant byte is in *abData[bByteIndex+1]*. If the maximum value 65535 is exceeded, there is an automatic overrun to 0.

Copyright IXXAT Automation GmbH

- *bByteIndex*:
  [in/out] If the value CAN_CTXMSG_INC_8 is specified in the field *bIncrMode*, this field defines the index of the data byte in the data field *abMsgData* that is to be automatically incremented after every transmit process. If the value CAN_CTXMSG_INC_16 is specified for *bIncrMode*, this field defines the index of the least significant byte (LSB) of the 16-bit value in the data field *abMsgData*. The most significant byte (MSB) of the 16-bit value is in the data field *abMsgData[bByteIndex+1]*.
- *dwMsgId*:
  [in/out] ID of the message (CAN-ID) in Intel format (right-aligned) without RTR bit.
- *uMsgInfo*:
  [in/out] Bit field with information on the message type. A description of the bit field is given in section 5.2.5.
- *abData*:
  [in/out] Array for up to 8 data bytes. The number of valid data bytes is defined by the field *uMsgInfo.Bits.dlc*.

5.3 LIN-specific data types

The declarations of all LIN-specific data types and constants are found in the file *<lintype.h>*.

5.3.1 LINCAPABILITIES

The data type describes the properties of a LIN connection. The structure is as follows:

```
typedef struct _LINCAPABILITIES
{
  UINT16 dwFeatures;
  UINT32 dwClockFreq;
  UINT32 dwTscDivisor;
} LINCAPABILITIES, *PLINCAPABILITIES;
```

- *dwFeatures*:
  [out] Supported properties. The value is a combination of one or more of the following constants:

  LIN_FEATURE_MASTER:
  The LIN controller supports the "master" mode.

  LIN_FEATURE_AUTORATE:
  The LIN controller supports automatic bitrate detection.

  LIN_FEATURE_ERRFRAME:
  The LIN controller supplies error messages.

  LIN_FEATURE_BUSLOAD:
  The LIN controller supports calculation of the bus load.

Copyright IXXAT Automation GmbH

- *dwClockFreq*:
  [out] Frequency of the primary timer in Hz.
- *dwTscDivisor*:
  [out] Divisor factor of the time stamp counter. The time stamp counter supplies the time stamp for LIN messages. The frequency of the time stamp counter is calculated from the frequency of the primary timer divided by the value defined here.

5.3.2 LINLINESTATUS

The data type describes the current status of a LIN controller. The structure is as follows:

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

- *bOpMode*:
  [out] Current operating mode of the LIN controller. The value is a combination of one or more LIN_OPMODE_ constants from *<lintype.h>* and contains the value defined in the parameter *bMode* when the function ***linControlInitialize*** is called.
- *bReserved*:
  [out] not used.
- *wBitrate*:
  [out] Currently set bitrate in bits per second.
- *dwStatus*:
  [out] Current status of the LIN controller. The value is a combination of one or more of the following constants:

  LIN_STATUS_OVRRUN:
  A data overrun has occurred in the receive buffer of the controller.

  LIN_STATUS_ININIT:
  The controller is in stopped status.

5.3.3 LINMONITORSTATUS

The data type describes the current status of a LIN message monitor. The structure is as follows:

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


Copyright IXXAT Automation GmbH

- *sLineStatus*:
  [out] Current status of the LIN controller. Further information is given in the description of the data structure ***LINLINESTATUS***.
- *fActivated*:
  [out] Indicates whether the message monitor is currently active (TRUE) or inactive (FALSE).
- *fRxOverrun*:
  [out] Indicates with the value TRUE whether the receive buffer has overrun.
- *bRxFifoLoad*:
  [out] Current level of the receive buffer in per cent.

5.3.4 LINMSGINFO

The data type contains various information on LIN messages in a 32-bit value. The value can be addressed either bytewise or via individual bit fields.

```
typedef union _LINMSGINFO
{
  struct
  {
    UINT8  bPid;
    UINT8  bType;
    UINT8  bDlen;
    UINT8  bFlags;
  } Bytes;

  struct
  {
    UINT32 pid  : 8;
    UINT32 type : 8;
    UINT32 dlen : 8;
    UINT32 ecs  : 1;
    UINT32 sor  : 1;
    UINT32 ovr  : 1;
    UINT32 ido  : 1;
    UINT32 res  : 4;
  } Bits;

} LINMSGINFO, *PLINMSGINFO;
```

The information of a LIN message can be addressed bytewise via the element *Bytes*. The following fields are defined for this:

- *Bytes.bPid*:
  [in/out] Protected identifier. See also *Bits.pid*.
- *Bytes.bType*:
  [in/out] Type of message. See also *Bits.type* and *Bits.ecs*.
- *Bytes.bDlen*:
  [in/out] Data length, see also *Bits.dlen*.
- *Bytes.bFlags*
  [in/out] Various flags. see also *Bits.ecs*, *Bits.sor*, *Bits.ovr* and *Bits.ido*.

The information of a LIN message can be accessed bitwise with the element *Bits*. The following bit fields are defined:

- *Bits.pid*:
  [in/out] Protected identifier of the message.
- *Bits.type*:
  [in/out] Type of message. The types listed in the following are defined for receive messages. For transmit messages, only the message types LIN_MSGTYPE_DATA and LIN_MSGTYPE_WAKEUP are currently defined, other values are not allowed here

  LIN_MSGTYPE_DATA:
  Normal message. All regular receive messages are of this type. The field *LINMSG.bPid* contains the ID of the message, the field *LINMSG.dwTime* the receive time. The field *LINMSG.abData* contains, depending on the length (see *Bits.dlen*), the data bytes of the message. In master mode, messages of this type can also be transmitted. The ID must be entered in the field *LINMSG.bPid* and in the field *LINMSG.abData*, depending on the length (*Bits.dlen*), the data to be transmitted. The field *LINMSG.dwTime* is set to 0. To transmit only the ID without data, *Bits.ido* is set to 1.

  LIN_MSGTYPE_INFO:
  Information message. This message type is entered in the receive buffer of all activated message monitors for certain events or in the event of changes in the status of the controller. The field *LINMSG.bPid* of the message always has the value 0xFF. The field *LINMSG.abData[0]* contains one of the following values:

Constant	Meaning
LIN_INFO_START	The controller was started. The field *LINMSG.dwTime* contains the relative start time (normally 0).
LIN_INFO_STOP	The controller was stopped. The field *LINMSG.dwTime* contains the value 0.
LIN_INFO_RESET	The controller was reset. The field *LINMSG.dwTime* contains the value 0.

Copyright IXXAT Automation GmbH

LIN_MSGTYPE_ERROR:
Error message. This message type is entered in the receive buffer of all activated message monitors when bus errors occur, as far as the flag LIN_OPMODE_ERRORS was defined at initialization of the controller. The field *LINMSG.bPid* of the message always has the value 0xFF. The time of the event is recorded in the field *LINMSG.dwTime* of the message. The field *LINMSG.abData[0]* contains one of the following values:

Constant	Meaning
LIN_ERROR_BIT	Bit error
LIN_ERROR_CHKSUM	Checksum error
LIN_ERROR_PARITY	Parity error of the identifier
LIN_ERROR_SLNORE	"Slave" does not respond
LIN_ERROR_SYNC	Invalid synchronization field
LIN_ERROR_NOBUS	No bus activity
LIN_ERROR_OTHER	Other, unspecified error

The field *LINMSG.abData[1]* of the message contains the low value byte of the current status (see also *LINLINESTATUS.dwStatus*). The content of the other data fields is undefined.

LIN_MSGTYPE_STATUS:
Status message. This message type is entered in the receive buffer of all activated message channels at changes in the controller status. The field *LINMSG.bPid* of the message always has the value 0xFF. The time of the event is recorded in the field *LINMSG.dwTime* of the message. The field *LINMSG.abData[0]* contains the low value byte of the current status (see also *LINLINESTATUS.dwStatus*). The content of the other data fields is undefined.

LIN_MSGTYPE_WAKEUP:
Only for transmit messages. Messages of this type generate a wake-up signal on the bus. The fields *LINMSG.dwTime*, *LINMSG.bPid* and *LINMSG.bDlen* have no significance.

LIN_MSGTYPE_TMOVR:
Counter overrun. Messages of this type are generated in the event of an overrun of the 32 bit time stamp of LIN messages. The field *LINMSG.dwTime* of the message contains the time of the event (normally 0) and in the field *LINMSG.bDlen* the number of timer overruns. The content of the data fields *LINMSG.abData* is undefined, the field *LINMSG.bPid* always has the value 0xFF.

LIN_MSGTYPE_SLEEP:
Goto Sleep message. The fields *LINMSG.dwTime*, *LINMSG.bPid* and *LINMSG.bDlen* have no significance.

- *Bits.dlen*:
  [in/out] Number of valid data bytes in the field *LINMSG.abData* of the message.
- *Bits.ecs*:
  [in/out] Enhanced checksum. The bit is set to 1 if it is a message with extended checksum in accordance with LIN 2.0.
- *Bits.sor*:
  [out] Sender of response. The bit is set with messages which the LIN controller itself has transmitted, i.e. with messages for which the controller has an entry in the response table.

- *Bits.ovr*:
  [out] Data overrun. The bit is set to 1 if the receive FIFO is full after this message is entered.
- *Bits.ido*:
  [in] ID only. The bit is only relevant for messages of type LIN_MSGTYPE_DATA which are transmitted directly. If the bit is set to 1 for transmit messages, only the ID is transmitted without data and is therefore used in master mode to send the IDs. With all other message types, this bit has no significance.
- *Bits.res*:
  Reserved for future extensions. This field is normally 0.

5.3.5 LINMSG

The data type describes the structure of LIN message telegrams.

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

- *dwTime*:
  [in/out] With receive messages, this field contains the relative receive time of the message in ticks. The resolution of a tick can be calculated from the fields *dwClockFreq* and *dwTscDivisor* of the structure ***LINCAPABILITIES*** in accordance with the following formula:

  Frequency [s] = dwTscDivisor / dwClockFreq

- *uMsgInfo*:
  [in/out] Bit field with information on the message. A description of the bit field is given in section 5.3.4.
- *abData*:
  [in/out] Array for up to 8 data bytes. The number of valid data bytes is defined by the field *uMsgInfo.Bits.dlen*.

Copyright IXXAT Automation GmbH