

VCI - Virtual CAN Interface

C-API Programmierhandbuch

Software Version 3

IXXAT

Hauptsitz

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

Geschäftsbereich USA

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

Sollten Sie zu diesem, oder einem unserer anderen Produkte Support benötigen, wenden Sie sich bitte schriftlich an:

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

Copyright

Die Vervielfältigung (Kopie, Druck, Mikrofilm oder in anderer Form) sowie die elektronische Verbreitung dieses Dokuments ist nur mit ausdrücklicher, schriftlicher Genehmigung von IXXAT Automation erlaubt. IXXAT Automation behält sich das Recht zur Änderung technischer Daten ohne vorherige Ankündigung vor. Es gelten die allgemeinen Geschäftsbedingungen sowie die Bestimmungen des Lizenzvertrags. Alle Rechte vorbehalten.

1	Systemübersicht	7
2	Geräteverwaltung und Gerätezugriff	9
2.1	Übersicht.....	9
2.2	Auflisten der verfügbaren Geräte und Interfacekarten	11
2.3	Aufsuchen bestimmter Geräte bzw. Interfacekarten	12
2.4	Zugriff auf ein Gerät bzw. eine Interfacekarte.....	12
3	Zugriff auf den Bus.....	14
3.1	Zugriff auf den CAN-Bus	14
3.1.1	Übersicht.....	14
3.1.2	Steuereinheit	15
3.1.2.1	Kontrollerzustände.....	16
3.1.2.2	Nachrichtenfilter	18
3.1.3	Nachrichtenkanal.....	20
3.1.3.1	Empfang von CAN-Nachrichten	22
3.1.3.2	Senden von CAN-Nachrichten	23
3.1.3.3	Verzögertes Senden von CAN-Nachrichten.....	24
3.1.4	Zyklische Sendeliste	24
3.2	Zugriff auf den LIN-Bus.....	28
3.2.1	Übersicht.....	28
3.2.2	Steuereinheit	29
3.2.2.1	Kontrollerzustände.....	29
3.2.2.2	Senden von LIN-Nachrichten	31
3.2.3	Nachrichtenmonitor	32
3.2.3.1	Empfang von LIN-Nachrichten	34
4	Schnittstellenbeschreibung	35
4.1	Generelle Funktionen.....	35
4.1.1	vciInitialize.....	35
4.1.2	vciFormatError	35
4.1.3	vciDisplayError	36
4.1.4	vciGetVersion	36
4.1.5	vciLuidToChar	37
4.1.6	vciCharToLuid	38
4.1.7	vciGuidToChar	38
4.1.8	vciCharToGuid	39
4.2	Die Funktionen der Geräteverwaltung.....	40

4.2.1 Funktionen für den Zugriff auf die Geräteliste.....	40
4.2.1.1 vciEnumDeviceOpen	40
4.2.1.2 vciEnumDeviceClose.....	40
4.2.1.3 vciEnumDeviceNext.....	41
4.2.1.4 vciEnumDeviceReset.....	41
4.2.1.5 vciEnumDeviceWaitEvent	42
4.2.1.6 vciFindDeviceByHwid.....	42
4.2.1.7 vciFindDeviceByClass	43
4.2.1.8 vciSelectDeviceDlg	44
4.2.2 Funktionen für den Zugriff auf Geräte bzw. Interfacekarten	45
4.2.2.1 vciDeviceOpen	45
4.2.2.2 vciDeviceOpenDlg	45
4.2.2.3 vciDeviceClose	46
4.2.2.4 vciDeviceGetInfo	46
4.2.2.5 vciDeviceGetCaps.....	47
4.3 Die Funktionen für den CAN-Zugriff	48
4.3.1 Steuereinheit	48
4.3.1.1 canControlOpen	48
4.3.1.2 canControlClose.....	49
4.3.1.3 canControlGetCaps.....	49
4.3.1.4 canControlGetStatus.....	50
4.3.1.5 canControlDetectBitrate.....	50
4.3.1.6 canControlInitialize	52
4.3.1.7 canControlReset.....	53
4.3.1.8 canControlStart	54
4.3.1.9 canControlSetAccFilter	55
4.3.1.10 canControlAddFilterIds	56
4.3.1.11 canControlRemFilterIds	57
4.3.2 Nachrichtenkanal.....	58
4.3.2.1 canChannelOpen	58
4.3.2.2 canChannelClose	59
4.3.2.3 canChannelGetCaps.....	59
4.3.2.4 canChannelGetStatus.....	60
4.3.2.5 canChannelInitialize.....	61
4.3.2.6 canChannelActivate	62
4.3.2.7 canChannelPeekMessage	63
4.3.2.8 canChannelPostMessage.....	63

4.3.2.9	canChannelWaitRxEvent.....	64
4.3.2.10	canChannelWaitTxEvent.....	65
4.3.2.11	canChannelReadMessage.....	66
4.3.2.12	canChannelSendMessage.....	67
4.3.3	Zyklische Sendeliste	68
4.3.3.1	canSchedulerOpen.....	68
4.3.3.2	canSchedulerClose	69
4.3.3.3	canSchedulerGetCaps	69
4.3.3.4	canSchedulerGetStatus	70
4.3.3.5	canSchedulerActivate.....	70
4.3.3.6	canSchedulerReset	71
4.3.3.7	canSchedulerAddMessage	71
4.3.3.8	canSchedulerRemMessage	72
4.3.3.9	canSchedulerStartMessage.....	73
4.3.3.10	canSchedulerStopMessage.....	73
4.4	Die Funktionen für den LIN-Zugriff	74
4.4.1	Steuereinheit	74
4.4.1.1	linControlOpen	74
4.4.1.2	linControlClose	75
4.4.1.3	linControlGetCaps.....	75
4.4.1.4	linControlGetStatus.....	76
4.4.1.5	linControlInitialize.....	76
4.4.1.6	linControlReset	77
4.4.1.7	linControlStart	78
4.4.1.8	linControlWriteMessage.....	79
4.4.2	Nachrichtenmonitor	79
4.4.2.1	linMonitorOpen	80
4.4.2.2	linMonitorClose	81
4.4.2.3	linMonitorGetCaps.....	81
4.4.2.4	linMonitorGetStatus	82
4.4.2.5	linMonitorInitialize.....	83
4.4.2.6	linMonitorActivate	84
4.4.2.7	linMonitorPeekMessage	85
4.4.2.8	linMonitorWaitRxEvent	85
4.4.2.9	linMonitorReadMessage	86
5	Typen und Strukturen.....	88
5.1	VCI spezifische Datentypen.....	88

5.1.1 VCIID	88
5.1.2 VCIDEVICEINFO	88
5.1.3 VCIDEVICECAPS	90
5.2 CAN spezifische Datentypen.....	91
5.2.1 CANCAPABILITIES	91
5.2.2 CANLINESTATUS.....	93
5.2.3 CANCHANSTATUS	94
5.2.4 CANSCHEDULERSTATUS	94
5.2.5 CANMSGINFO	95
5.2.6 CANMSG	98
5.2.7 CANCYCLICTXMSG	99
5.3 LIN spezifische Datentypen.....	100
5.3.1 LINCAPABILITIES	100
5.3.2 LINLINESTATUS.....	101
5.3.3 LINMONITORSTATUS	101
5.3.4 LINMSGINFO	102
5.3.5 LINMSG	105

1 Systemübersicht

Beim Virtual Card Interface (VCI) handelt es sich um eine Systemerweiterung dessen Aufgabe darin besteht Applikationen einen einheitlichen Zugriff auf unterschiedliche Geräte und Interfacekarten von IXXAT zu ermöglichen. Nachfolgende Abbildung zeigt den prinzipiellen Aufbau des Systems und dessen einzelne Komponenten.

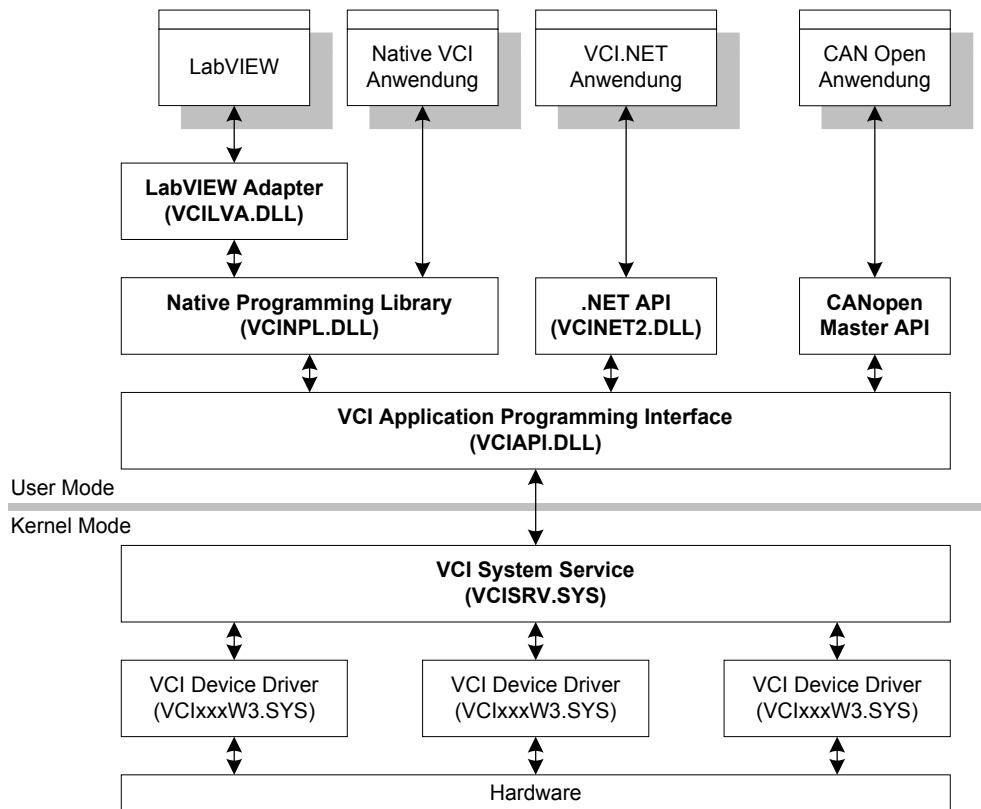


Bild 1-1: Systemkomponenten

Das VCI besteht im wesentlichen aus den folgenden Komponenten:

- Native VCI Programmierschnittstelle (VCINPL.DLL)
- VCI.NET Programmierschnittstelle (VCINET2.DLL)
- VCI System Service API (VCI-API.DLL)
- VCI System Service (VCISRV.SYS)
- Ein oder mehrere VCI Gerätetreiber (VCIxxxWy.SYS)

Die Programmierschnittstellen stellen die Verbindung zwischen dem VCI System Service oder kurz VCI Server und den Applikationsprogrammen über einen Satz vordefinierter Schnittstellen und Funktionen her. Der LabVIEW-Adapter dient ausschließlich der Anpassung an die von der nativen Programmierschnittstelle verwendeten Datentypen, bietet selbst jedoch keine zusätzliche Funktionalität.

Systemübersicht

Der im Betriebssystemkern laufende VCI Server übernimmt hauptsächlich die Verwaltung der VCI spezifischen Gerätetreiber, regelt den Zugriff auf die Geräte und Interfacekarten von IXXAT und stellt Mechanismen für den Austausch von Daten zwischen Applikations- und Betriebssystemebene bereit.

Die im folgenden betrachtete Programmierschnittstelle besteht aus folgenden Teilkomponenten und Funktionen.

Native VCI Programmier Schnittstelle (VCINPL.DLL)			
Geräteverwaltung und Gerätezugriff	CAN-Steuerung	CAN-Nachrichtenkanäle	Zyklische CAN Sendeliste
vciEnumDeviceOpen vciEnumDeviceClose vciEnumDeviceNext vciEnumDeviceReset vciEnumDeviceWaitEvent vciFindDeviceByHwid vciFindDeviceByClass vciSelectDeviceDlg vciDeviceOpen vciDeviceOpenDlg vciDeviceClose vciDeviceGetInfo vciDeviceGetCaps	canControlOpen canControlClose canControlGetCaps canControlGetStatus canControlDetectBtrrate canControlInitialize canControlReset canControlStart canControlSetAccFilter canControlAddFilterIds canControlRemFilterIds	canChannelOpen canChannelClose canChannelGetCaps canChannelGetStatus canChannelInitialize canChannelActivate canChannelPeekMessage canChannelPostMessage canChannelWaitRxEvent canChannelWaitTxEvent canChannelReadMessage canChannelSendMessage	canSchedulerOpen canSchedulerClose canSchedulerGetCaps canSchedulerGetStatus canSchedulerActivate canSchedulerReset canSchedulerAddMessage canSchedulerRemMessage canSchedulerStartMessage canSchedulerStopMessage
	LIN-Steuerung	LIN-Nachrichtenmonitore	
	linControlOpen linControlClose linControlGetCaps linControlGetStatus linControlInitialize linControlReset linControlStart linControlWriteMessage	linMonitorOpen linMonitorClose linMonitorGetCaps linMonitorInitialize linMonitorActivate linMonitorPeekMessage linMonitorWaitRxEvent linMonitorReadMessage	

2 Geräteverwaltung und Gerätezugriff

2.1 Übersicht

Die Geräteverwaltung ermöglicht das Auflisten und den primären Zugriff auf die beim VCI Server angemeldeten Geräte bzw. Interfacekarten von IXXAT. Hierzu stehen die in nachfolgender Abbildung gezeigten Funktionen zur Verfügung:

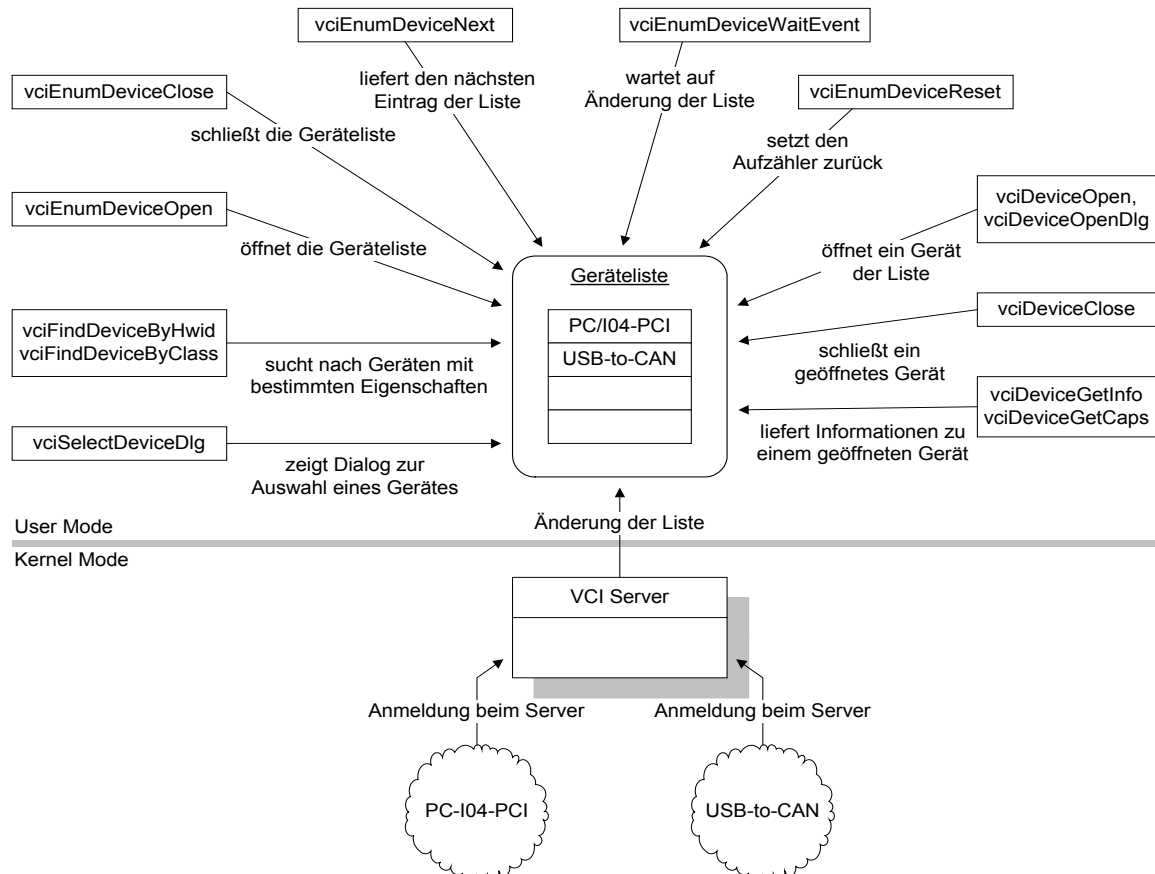


Bild 2-1: Komponenten und Funktionen der Geräteverwaltung

Der VCI Server verwaltet alle Geräte in einer systemweiten globalen Liste, die nachfolgend kurz als **Geräteliste** bezeichnet wird. Die Anmeldung eines Gerätes bzw. einer Interfacekarte beim Server erfolgt automatisch beim Start des Computers oder, wie z.B. beim USB-to-CAN compact, wenn eine Verbindung zwischen Computer und Gerät bzw. zur Interfacekarte hergestellt wird. Ist ein Gerät nicht mehr verfügbar, weil z.B. die Verbindung unterbrochen wurde, wird dieses automatisch aus der Geräteliste entfernt.

Geräteverwaltung und Gerätezugriff

Die An- oder Abmeldung von Geräten bzw. Interfacekarten erfolgt auch dann, wenn beim Gerätemanager vom Betriebssystem ein Gerätetreiber aktiviert oder deaktiviert wird (siehe nachfolgende Abbildung).

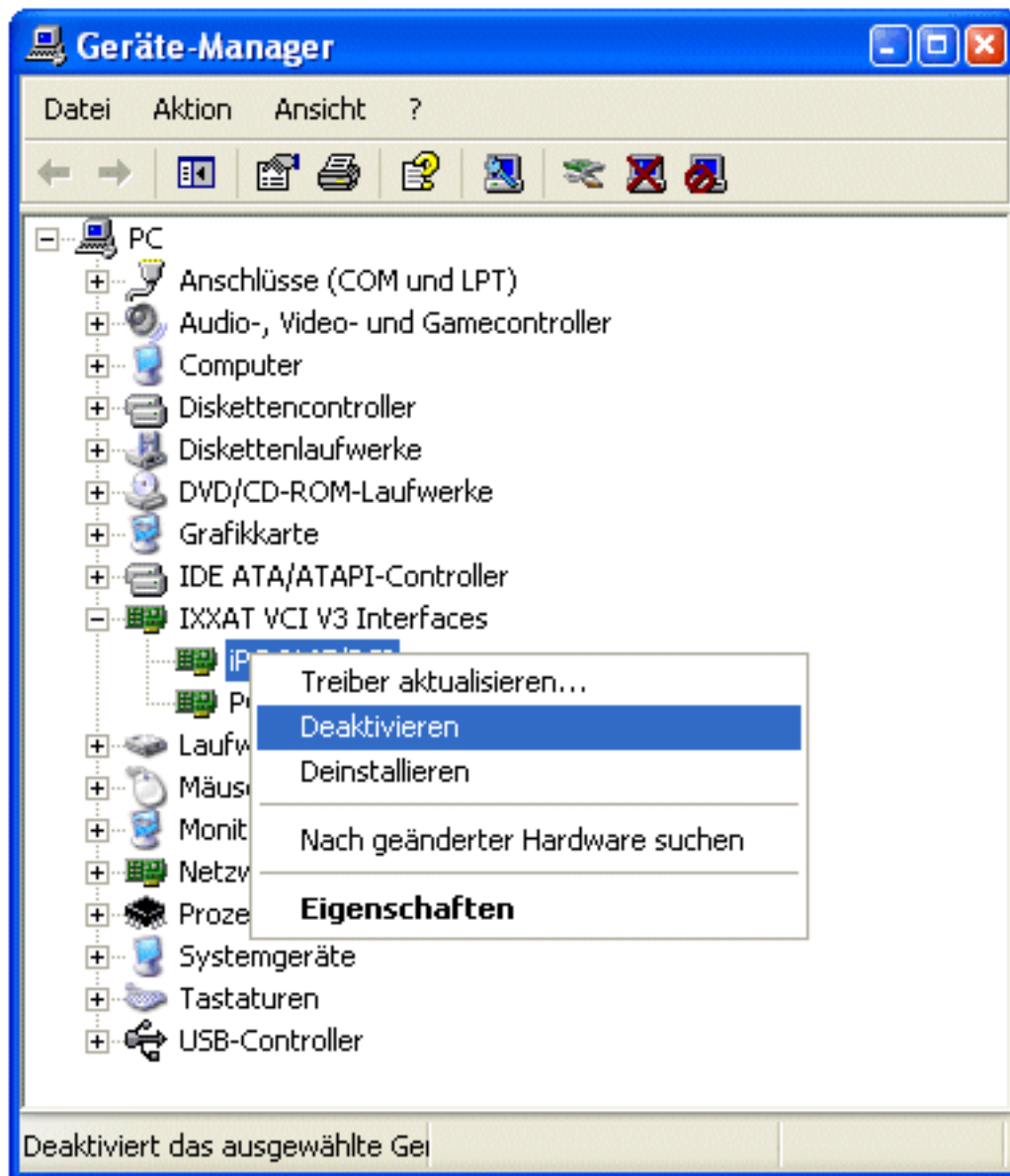


Bild 2-2: Gerätemanager vom Betriebssystem

2.2 Auflisten der verfügbaren Geräte und Interfacekarten

Der VCI Server verwaltet eine Liste aller momentan verfügbaren Geräte und Interfacekarten von IXXAT in einer systemweit globalen Geräteliste. Zugang zu dieser Liste erhält man durch Aufruf der Funktion *vciEnumDeviceOpen*. Bei erfolgreicher Ausführung liefert die Funktion einen Handle auf die Geräteliste zurück. Dieser Handle wird benötigt, um Informationen zu den einzelnen Geräten bzw. Interfacekarten abzurufen oder um Änderungen an der Geräteliste zu überwachen.

Die Funktion *vciEnumDeviceNext* liefert bei jedem Aufruf Informationen zu einem Gerät bzw. zu einer Interfacekarte aus der Liste. Dabei wird ein interner Index erhöht, so dass ein weiterer Aufruf der Funktion die Informationen zum jeweils nächsten Gerät bzw. zur nächsten Interfacekarte in der Liste liefert. Den hierzu notwendigen Speicher muss die Applikation in Form einer Struktur vom Typ *VCIDEVICEINFO* zur Verfügung stellen. Nachfolgend sind die wichtigsten Informationen über ein Gerät zusammengestellt:

- *VciObjectId*: Eindeutige Kennzahl des Gerätes. Der Server weist jedem Gerät bzw. jeder Interfacekarte bei deren Anmeldung eine systemweit eindeutige Kennzahl (*VCIID*) zu. Diese Kennzahl wird für spätere Zugriffe auf das Gerät bzw. die Interfacekarte benötigt.
- *DeviceClass*: Geräteklasse. Alle VCI-konforme Gerätetreiber kennzeichnen ihre unterstützte Geräte- bzw. Interfacekarte- Klasse mit einer weltweit eindeutigen und einmaligen Kennzahl (GUID). Unterschiedliche Geräte bzw. Interfacekarten gehören unterschiedlichen Geräteklassen an. So hat z.B. die IPC-I165/PCI eine andere Geräteklasse, als die PC-I04/PCI.
- *UniqueHardwareId*: Hardwarekennung. Alle VCI-konformen Geräte bzw. Interfacekarten besitzen eine eindeutige Hardwarekennung. Diese Kennung kann z.B. dazu verwendet werden, um zwischen zwei PC-I04/PCI Karten zu unterscheiden oder um nach einem Gerät mit bestimmter Kennung zu suchen.

Die Geräteliste ist vollständig durchlaufen, wenn die Funktion *vciEnumDeviceNext* den Wert *VCI_E_NO_MORE_ITEMS* zurückliefert. Andere Rückgabewerte als *VCI_OK* oder *VCI_E_NO_MORE_ITEMS* deuten dagegen auf einen Fehler hin.

Der interne Listenindex lässt sich mit der Funktion *vciEnumDeviceReset* wieder auf den Anfang zurückstellen, so dass ein nachfolgender Aufruf der Funktion *vciEnumDeviceNext* wieder die Informationen zum ersten Gerät bzw. zur ersten Interfacekarte in der Liste liefert.

Applikationen können mittels der Funktion *vciEnumDeviceWaitEvent* Änderungen an der Geräteliste überwachen. Ändert sich der Inhalt der Geräteliste, liefert die Funktion den Wert *VCI_OK* zurück. Andere Rückgabewerte als *VCI_OK* deuten entweder auf einen Fehler hin oder signalisieren eine Überschreitung der beim Funktionsaufruf angegebenen Wartezeit.

Die Funktion *vciEnumDeviceClose* schließt eine geöffnete Geräteliste und gibt den angegebenen Handle wieder frei. Um Systemressourcen zu sparen sollten Applikationen diese Funktion immer dann aufrufen, wenn kein weiterer Zugriff auf die Geräteliste erforderlich ist.

Im nachfolgenden Kapitel werden einige Funktionen vorgestellt, die eine einfachere Suche nach einem bestimmten Gerät bzw. einer bestimmten Interfacekarte, bzw. eine einfachere Auswahl eines Gerätes ermöglichen, als die bisher beschriebenen Funktionen.

2.3 Aufsuchen bestimmter Geräte bzw. Interfacekarten

Zur Suche nach einem Gerät bzw. einer Interfacekarte stehen die Funktionen *vciFindDeviceByHwid* und *vciFindDeviceByClass* zur Verfügung.

Die Funktion *vciFindDeviceByHwid* sucht nach einem Gerät bzw. einer Interfacekarte mit einer bestimmten Hardwarekennung. Alle VCI-konformen Geräte bzw. Interfacekarten besitzen eine eindeutige Hardwarekennung, die im Gegensatz zur Gerätekenzahl (*VCIID*) auch bei einem Neustart des Systems erhalten bleibt. Die Hardwarekennung kann daher z.B. in Konfigurationsdateien gespeichert werden und ermöglicht eine automatische Konfiguration der Anwendersoftware nach dem Programmstart, bzw. nach einem Systemstart.

Ähnliche Funktionalität bietet auch die Funktion *vciFindDeviceByClass*. Diese erwartet als Parameter die Geräteklasse (GUID) und die Instanznummer des gesuchten Gerätes bzw. der gesuchten Interfacekarte. Eine Applikation kann damit z.B. nach der ersten PC-I04/PCI im System suchen.

Applikationen können mittels der Funktion *vciSelectDeviceDlg* ein vordefiniertes Dialogfenster anzeigen. Mit diesem Dialog kann ein Benutzer ein Gerät bzw. eine Interfacekarte aus der Geräteliste auswählen. Das Dialogfenster kann auch dazu verwendet werden, um die Hardwarekennung oder die Geräteklasse eines Gerätes bzw. einer Interfacekarte herauszufinden.

Alle Funktionen liefern bei erfolgreicher Ausführung die Gerätekenzahl (*VCIID*) des gefundenen oder ausgewählten Gerätes bzw. der Interfacekarte zurück. Diese Gerätekenzahl wird für spätere Zugriffe benötigt.

2.4 Zugriff auf ein Gerät bzw. eine Interfacekarte

Zugriff auf ein Gerät bzw. eine Interfacekarte erhält man mittels den Funktionen *vciDeviceOpen* oder *vciDeviceOpenDlg*. Beide Funktionen liefern bei erfolgreicher Ausführung einen Handle auf das geöffnete Gerät zurück.

Die Funktion *vciDeviceOpen* erwartet als Eingabeparameter die Gerätekenzahl (*VCIID*) des zu öffnenden Gerätes bzw. der zu öffnenden Interfacekarte. Diese Kennzahl kann dabei wie in Kapitel 2.2 oder 2.3 beschrieben ermittelt werden.

Im Gegensatz dazu zeigt die Funktion *vciDeviceOpenDlg* ein Dialogfenster mit der aktuellen Geräteliste an und stellt dem Anwender eine visuelle Möglichkeit zur Auswahl des zu öffnenden Gerätes bereit.

Mit der Funktion *vciDeviceGetInfo* können Informationen über ein geöffnetes Gerät abgefragt werden. Den hierzu notwendigen Speicher stellt die Applikation in Form einer Struktur vom Typ *VCIDEVICEINFO* zur Verfügung. Die Funktion liefert die gleichen Informationen wie die in Kapitel 2.2 beschriebene Funktion *vciEnumDeviceNext*. Eine ausführliche Beschreibung der Struktur befindet sich in Kapitel 5.1.2.

Informationen über die technische Ausstattung eines Gerätes bzw. einer Interfacekarte liefert die Funktion *vciDeviceGetCaps*. Die Funktion benötigt neben dem Handle des Gerätes die Adresse einer Struktur vom Typ *VCIDEVICECAPS*. Bei erfolgreicher Ausführung liefert die Funktion die gewünschten Informationen in dieser Struktur zurück.

Die von *vciDeviceGetCaps* gelieferten Informationen geben Auskunft darüber wie viele Busanschlüsse auf einem Gerät oder einer Interfacekarte vorhanden sind. Nachfolgende Abbildung zeigt eine Interfacekarte mit zwei Busanschlüssen.

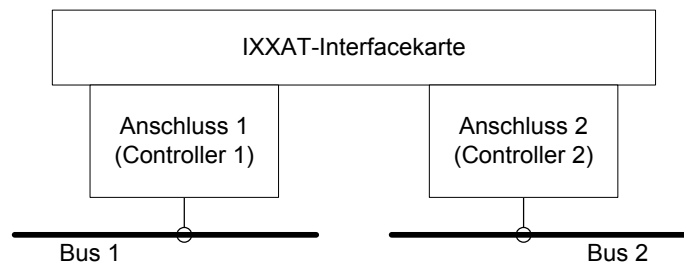


Bild 2-3: IXXAT-Interfacekarte mit zwei Busanschlüssen

Die Struktur *VCIDEVICECAPS* enthält eine Tabelle mit bis zu 32 Einträgen, die den jeweiligen Busanschluss bzw. Controller beschreiben. Der Tabelleneintrag 0 beschreibt dabei Busanschluss 1, Tabelleneintrag 1 Busanschluss 2, usw.. Weitere Informationen zur Struktur *VCIDEVICECAPS* finden sich in Kapitel 5.1.3.

Die Funktion *vciDeviceClose* schließt ein geöffnetes Gerät bzw. eine geöffnete Interfacekarte und gibt deren Handle wieder frei. Um Systemressourcen zu sparen sollten Applikationen diese Funktion immer dann aufrufen, wenn keine weiteren Zugriffe auf das Gerät erforderlich sind.

Das Handle des Gerätes wird neben den oben genannten Funktionen auch für den Zugang zu den Busanschlüssen benötigt. Weitere Informationen hierzu finden sich in Kapitel 3.

3 Zugriff auf den Bus

3.1 Zugriff auf den CAN-Bus

3.1.1 Übersicht

Nachfolgend ist eine typische Interfacekarte von IXXAT mit zwei CAN-Anschlüssen abgebildet.

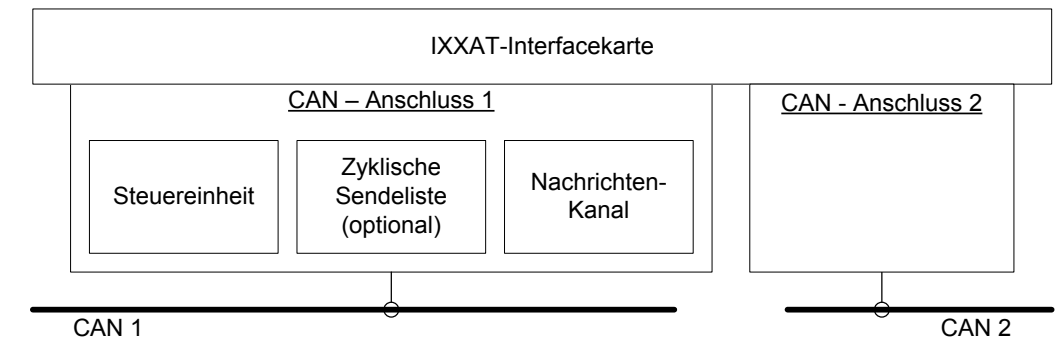


Bild 3-1: IXXAT- Interfacekarte mit zwei CAN- Anschlüssen

Neben den CAN-Anschlüssen kann ein Gerät bzw. eine Interfacekarte auch noch andere Typen von Busanschlüssen besitzen, die im Folgenden jedoch nicht weiter beachtet werden.

Wie in Bild 3-1 für den Anschluss 1 dargestellt ist, setzt sich jeder CAN-Anschluss aus bis zu drei unterschiedlichen Komponenten zusammen. Eine Steuereinheit, ein oder mehrere Nachrichtenkanäle und gegebenenfalls eine zyklische Sendeliste. Die Steuereinheit und die Nachrichtenkanäle sind immer vorhanden. Die zyklische Sendeliste ist normalerweise nur bei Geräten bzw. bei Interfacekarten vorhanden, die über einen eigenen Mikroprozessor verfügen.

Zugriff auf die Steuereinheit eines CAN-Anschlusses erhält man mit der Funktion *canControlOpen*. Die Funktion *canChannelOpen* öffnet einen Nachrichtenkanal und die Funktion *canSchedulerOpen* einen Zugang zur zyklischen Sendeliste des Anschlusses, falls vorhanden.

Alle drei Funktionen erwarten im ersten Parameter das Handle des Gerätes bzw. der Interfacekarte und im zweiten Parameter die Nummer des CAN-Anschlusses. Für Anschluss 1 wird dabei die Nummer 0, für Anschluss 2 die Nummer 1, usw. angegeben.

Zur Einsparung von Systemressourcen kann nach dem Öffnen einer Komponente das Handle des Gerätes bzw. der Interfacekarte wieder freigegeben werden. Für die weiteren Zugriffe auf den Anschluss ist nur noch das Handle der jeweiligen Steuereinheit, des Nachrichtenkanals oder der zyklischen Sendeliste erforderlich.

Die Funktionen *canControlOpen*, *canChannelOpen* und *canSchedulerOpen* lassen sich so aufrufen, dass dem Benutzer ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte und des CAN-Anschlusses präsentiert wird. Dies erreicht man, indem für die Anschlussnummer der Wert 0xFFFFFFFF angegeben wird. In diesem Fall erwarten die Funktionen im ersten Parameter nicht das Handle des Gerätes bzw. der Interfacekarte, sondern das Handle vom übergeordneten Fenster (Parent) oder den Wert NULL, falls kein übergeordnetes Fenster verfügbar ist. Bei erfolgreicher Ausführung liefern alle drei Funktionen ein Handle auf die geöffnete Komponente zurück. Tritt ein Fehler oder ein Zugriffskonflikt auf, liefern die Funktionen einen entsprechenden Fehlercode zurück.

Wird eine geöffnete Komponente nicht mehr benötigt kann diese durch Aufruf einer der Funktionen *canControlClose*, *canChannelClose* und *canSchedulerClose* wieder geschlossen werden.

Welche Möglichkeiten ein CAN-Anschluss bietet, bzw. wie der CAN-Anschluss zu verwenden ist, wird in den folgenden Unterkapiteln genauer beschrieben.

3.1.2 Steuereinheit

Die Steuereinheit stellt Funktionen zur Konfiguration des CAN-Controllers, dessen Übertragungseigenschaften sowie Funktionen zum Konfigurieren von CAN-Nachrichtenfiltern und zur Abfrage des aktuellen Controllerzustandes bereit.

Die Steuereinheit kann von mehreren Applikationen gleichzeitig geöffnet werden um den Status und die Eigenschaften des CAN-Controllers zu ermitteln. Der CAN-Controller kann jedoch zur gleichen Zeit immer nur von einer Applikation initialisiert werden. Dadurch lassen sich Situationen vermeiden, bei denen z.B. eine Applikation den CAN-Controller startet, eine andere aber stoppt.

Geöffnet wird die Steuereinheit durch Aufruf der Funktion *canControlOpen*.

Mittels der Funktion *canControlClose* wird eine geöffnete Steuereinheit wieder geschlossen und damit für andere Applikationen verfügbar. Ein Programm sollte die Steuereinheit daher freigeben, wenn diese nicht mehr benötigt wird.

Die Applikation, die als erste die Funktion *canControlOpen* aufruft, kann den CAN-Controller exklusiv steuern. Bevor eine andere Applikation die exklusive Steuerung übernehmen kann, müssen alle Applikationen die parallel geöffnete Steuereinheit mit der Funktion *canControlClose* schließen.

3.1.2.1 Kontrollerzustände

Die folgende Abbildung zeigt die verschiedenen Zustände einer Steuereinheit bzw. eines CAN-Controllers.

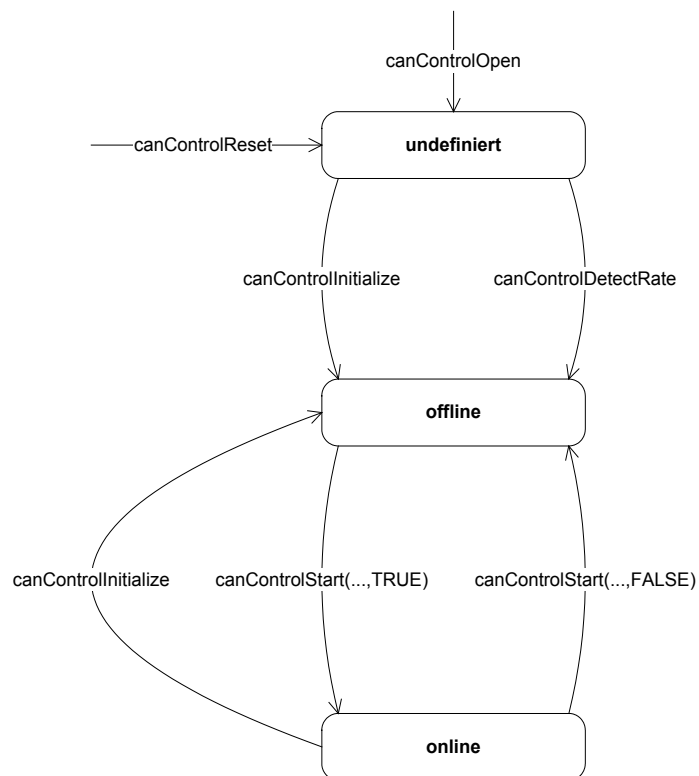


Bild 3-2: Kontrollerzustände

Nach dem Öffnen der Steuereinheit befindet sich der Controller normalerweise in einem nicht initialisiertem Zustand. Dieser Zustand wird durch Aufruf einer der Funktionen *canControlInitialize* oder *canControlDetectBitrate* verlassen. Danach befindet sich der CAN-Controller im Zustand „offline“.

Liefert die Funktion *canControlInitialize* einen Fehlercode entsprechend „Zugriff verweigert“ zurück, wird der CAN-Controller bereits von einem anderen Programm verwendet.

Mittels *canControlInitialize* wird die Betriebsart und Bitrate vom CAN-Controller eingestellt. Die Werte für die Bus Timing Register in den Parametern *bBtr0* und *bBtr1* entsprechen dabei den Werten für die Register BTR0 und BTR1 vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz.

Ausführliche Informationen zum Einstellen der Bitrate befinden sich im Datenblatt zum SJA 1000 in Kapitel 6.5. Eine Zusammenstellung der Bus Timing Werte mit allen CiA bzw. CANopen konformen Bitraten befinden sich in der folgenden Tabelle.

Bitrate (KBit)	Vordefinierte Konstanten für 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

Ist der CAN-Anschluss mit einem laufenden System verbunden bei dem die Bitrate unbekannt ist, kann die aktuelle Bitrate mit der Funktion *canControlDetectBitrate* ermittelt werden. Die von der Funktion ermittelten Bus Timing Werte können anschließend der Funktion *canControlInitialize* übergeben werden.

Gestartet, bzw. gestoppt wird der CAN-Controller durch Aufruf der Funktion *canControlStart*. Nach erfolgreichem Aufruf der Funktion mit dem Wert TRUE im Parameter *fStart*, befindet sich der Controller im Zustand „online“. In diesem Zustand ist der CAN-Controller aktiv mit dem Bus verbunden. Eingehende CAN-Nachrichten werden dabei an alle aktiven CAN-Nachrichtenkanäle weitergeleitet, bzw. Sendenachrichten von den Nachrichtenkanälen an den Bus ausgegeben.

Ein Aufruf der Funktion *canControlStart* mit dem Wert FALSE im Parameter *fStart* schaltet den CAN-Controller „offline“. Der Nachrichtentransport wird dabei unterbrochen und der Controller deaktiviert.

Die Funktion *canControlReset* schaltet den CAN-Controller immer in den Zustand „nicht initialisiert“, setzt dabei gleichzeitig die Controllerhardware zurück und löscht alle eingestellten Nachrichtenfilter. Beachten Sie, dass es beim Zurücksetzen der Controllerhardware zu einem fehlerhaften Nachrichtentelegrammen auf dem Bus kommt, wenn bei Aufruf der Funktion ein laufender Sendevorgang mitten in der Übertragung unterbrochen wird.

3.1.2.2 Nachrichtenfilter

Jede Steuereinheit verfügt über ein zweistufiges Nachrichtenfilter. Die Filterung von empfangenen Nachrichten erfolgt ausschließlich anhand deren ID (CAN-ID), Datenbytes werden dabei nicht berücksichtigt.

Bei einer Sendenachricht mit gesetztem 'Self reception request' bit wird die Nachricht, sobald Sie über den Bus gesendet wurde, in den Empfangsbuffer eingetragen. Der Nachrichtenfilter wird in diesem Fall umgangen.

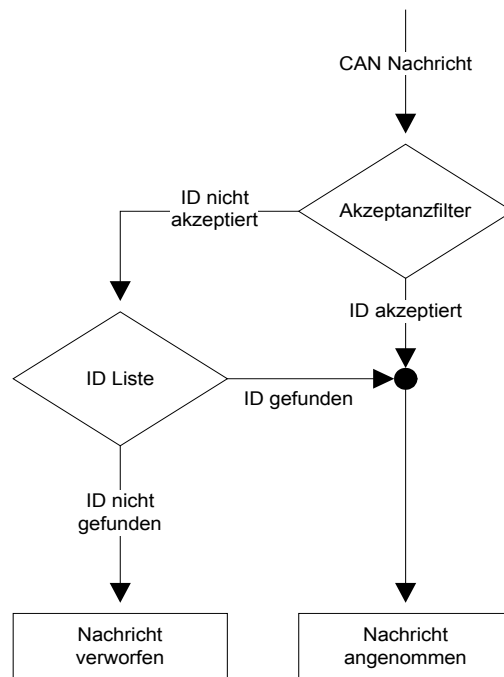


Bild 3-3: Filtermechanismus

Die erste Filterstufe, bestehend aus einem Akzeptanzfilter, vergleicht die ID einer empfangenen Nachricht mit einem binären Bitmuster. Korreliert die ID mit dem eingestellten Bitmuster wird die Nachricht angenommen, andernfalls wird sie der zweiten Filterstufe zugeführt. Die zweite Filterstufe besteht aus einer Liste mit registrierten IDs. Entspricht die ID der Nachricht einer ID in der Liste, wird die Nachricht ebenfalls angenommen, andernfalls wird sie verworfen.

Der CAN-Controller besitzt für 11-Bit und 29-Bit IDs getrennte und voneinander unabhängige Filter. Beim Zurücksetzen oder beim Initialisieren des Controllers werden die Filter so eingestellt, dass alle Nachrichten durchgelassen werden.

Die Filtereinstellungen lassen sich mit den Funktionen *canControlSetAccFilter*, *canControlAddFilterIds* und *canControlRemFilterIds* ändern. Als Eingabewerte erwarten die Funktionen in den Parametern *dwCode* und *dwMask* zwei Bitmuster welche die ID, bzw. die Gruppe von IDs bestimmen, die vom Filter durchgelassen werden. Ein Aufruf der Funktionen ist jedoch nur dann erfolgreich, wenn sich der Controller im Zustand „offline“ befindet.

Die Bitmuster in den Parametern *dwCode* und *dwMask* bestimmen welche IDs vom Filter durchgelassen werden. Der Wert von *dwCode* bestimmt dabei das Bitmuster der ID, wohingegen *dwMask* festlegt welche Bits in *dwCode* für den Vergleich herangezogen werden. Hat ein Bit in *dwMask* den Wert 0, wird das entsprechende Bit in *dwCode* nicht für den Vergleich herangezogen. Hat es dagegen den Wert 1, ist es beim Vergleich relevant.

Beim 11-Bit Filter sind nur die unteren 12 Bit relevant. Beim 29-Bit Filter werden die Bits 0 bis 29 verwendet. Die anderen Bits müssen vor Aufruf der Funktionen auf 0 gesetzt werden.

Nachfolgende Tabellen zeigen den Zusammenhang zwischen den Bits in den Parametern *dwCode* und *dwMask*, sowie den Bits der Nachrichten- ID (CAN-ID):

Bedeutung der Bits beim 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

Bedeutung der Bits beim 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

Die Bits 11 bis 1 bzw. 29 bis 1 entsprechen den ID Bits 10 bis 0 bzw. 28 bis 0. Bit 0 entspricht immer dem Remote Transmission Requests Bit (RTR) einer Nachricht.

Folgendes Beispiel zeigt die Werte für die Parameter *dwCode* und *dwMask*, um nur die Nachrichten im Bereich 100h bis 103h durchzulassen, bei denen das RTR-Bit 0 ist:

<i>dwCode</i> :	001 0000 0000 0
<i>dwMask</i> :	111 1111 1100 1
Gültige 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

Wie das Beispiel zeigt, lassen sich mit dem einfachen Akzeptanzfilter nur einzelne IDs oder Gruppen von IDs frei schalten. Entsprechen die gewünschten IDs jedoch nicht einem bestimmten Bitmuster, stößt der Akzeptanzfilter schnell an seine Grenzen. Hier kommt die zweite Filterstufe mit der ID- Liste ins Spiel. Jede Liste kann bis zu 2048 IDs, bzw. 4096 Einträge aufnehmen.

Über die Funktion *canControlAddFilterIds* können einzelne IDs oder Gruppen von IDs in die Liste eingetragen und mittels der Funktion *canControlRemFilterIds* wieder daraus entfernt werden. Die Parameter *dwCode* und *dwMask* haben dabei das gleiche Format wie beim Akzeptanzfilter.

Wird die Funktion *canControlAddFilterIds* z.B. mit den Werten aus dem vorherigen Beispiel aufgerufen, trägt die Funktion die IDs 100h bis 103h in die Liste ein. Soll bei einem Aufruf der Funktion nur eine einzige ID eingetragen werden, gibt man in *dwCode* die gewünschte ID (einschließlich RTR-Bit) an und setzt *dwMask* auf 0xFFF bzw. 0xFFFFFFFF.

Der Akzeptanzfilter kann durch einen Aufruf der Funktion *canControlSetAccFilter* vollständig gesperrt werden, wenn für *dwCode* der Wert **CAN_ACC_CODE_NONE** und für *dwMask* der Wert **CAN_ACC_MASK_NONE** angegeben wird. Die weitere Filterung erfolgt danach nur noch anhand der ID-Liste. Ein Aufruf der Funktion mit den Werten **CAN_ACC_CODE_ALL** und **CAN_ACC_MASK_ALL** hingegen öffnet den Akzeptanzfilter vollständig. Die ID-Liste ist in diesem Fall also wirkungslos.

3.1.3 Nachrichtenkanal

Erzeugt, bzw. geöffnet wird ein CAN-Nachrichtenkanal durch Aufruf der Funktion *canChannelOpen*. Der Parameter *fExclusive* bestimmt dabei, ob der Anschluss exklusiv verwendet werden soll. Wird hier der Wert **TRUE** angegeben, können nach erfolgreicher Ausführung der Funktion keine weiteren Nachrichtenkanäle mehr geöffnet werden. Wird der CAN-Anschluss nicht exklusiv verwendet, lassen sich prinzipiell beliebig viele Nachrichtenkanäle einrichten.

Bei exklusiver Verwendung des Anschlusses ist der Nachrichtenkanal direkt mit dem CAN-Controller verbunden. Folgende Abbildung zeigt diese Konfiguration.

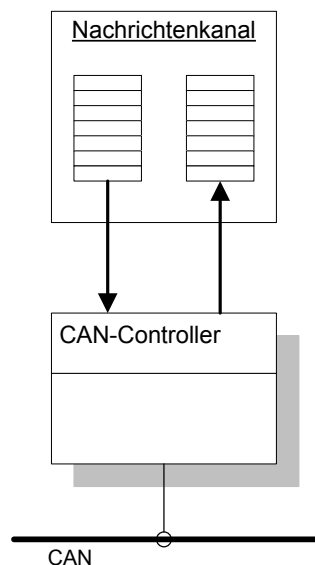


Bild 3-4: Exklusive Verwendung eines CAN-Anschlusses

Bei nicht exklusiver Verwendung des Anschlusses (*fExclusive* = FALSE) wird ein Verteiler zwischen Controller und die Nachrichtenkanäle geschaltet.

Der Verteiler leitet eingehenden Nachrichten vom CAN-Controller an alle Nachrichtenkanäle weiter und überträgt die Sendenachrichten der Kanäle an den Controller. Die Verteilung der Nachrichten erfolgt dabei so, dass kein Kanal bevorzugt behandelt wird. Nachfolgende Abbildung zeigt eine Konfiguration mit drei Kanälen an einem CAN-Anschluss.

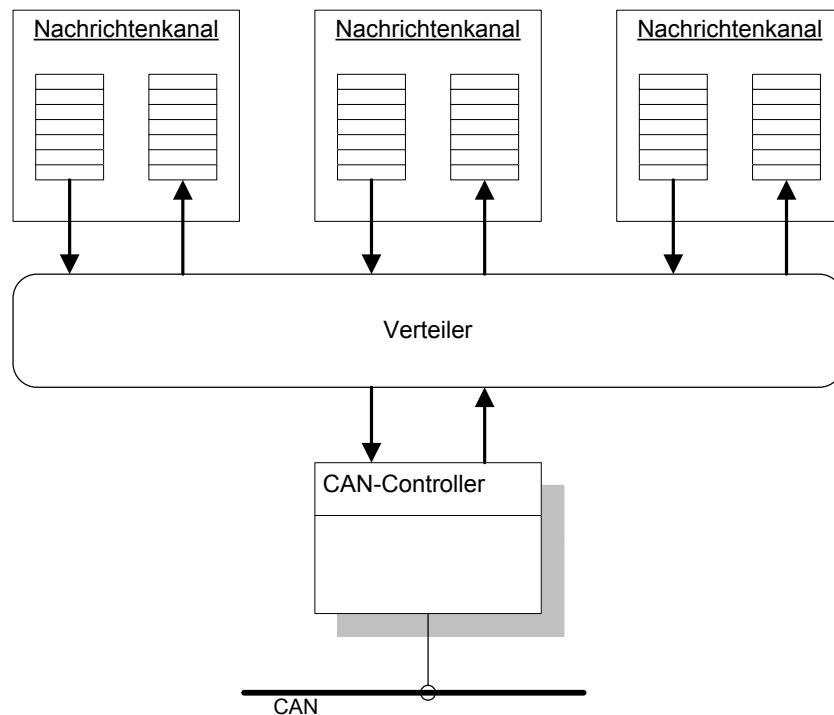


Bild 3-5: CAN-Nachrichtenverteiler

Nachdem ein Nachrichtenkanal geöffnet wurde, muss dieser zunächst initialisiert werden. Hierzu dient die Funktion *canChannelInitialize*. Als Eingabeparameter erwartet die Funktion die Größe vom Empfangs- und Sendepuffer in Anzahl CAN-Nachrichten sowie die Schwellwerte für den Empfangs- und Sendevent.

Der für den Empfangs- und Sendepuffer reservierte Speicher stammt aus einem begrenzten Speicherpool. Die einzelnen Puffer eines Nachrichtenkanals sollten daher nicht mehr als ca. 2000 Nachrichten umfassen.

Das Empfangs-Event eines Kanals wird ausgelöst, wenn der Empfangspuffer mindestens die in *wRxThreshold* angegebene Anzahl Nachrichten enthält. Das Sendevent, wenn der Sendepuffer mindestens Platz für die in *wTxThreshold* angegebene Anzahl Nachrichten hat.

Ist der Kanal eingerichtet, kann er mittels der Funktion *canChannelActivate* aktiviert bzw. deaktiviert werden. Aktiviert wird der Kanal, wenn die Funktion mit dem Wert TRUE im Parameter *fEnable* aufgerufen wird. Ein Aufruf der Funktion mit dem Wert FALSE im Parameter *fEnable*, deaktiviert den Kanal. Standardmäßig ist ein Kanal nach dem Öffnen deaktiviert.

Nachrichten werden nur dann vom CAN-Controller empfangen, bzw. an diesen gesendet, wenn der Kanal aktiv ist. Außerdem muss der CAN-Controller gestartet sein, andernfalls erfolgt kein Nachrichtentransfer zwischen CAN-Bus und Kanal. Einfluss auf die empfangenen Nachrichten hat auch der Nachrichtenfilter vom CAN-Controller. Näheres hierzu befindet sich in Kapitel 3.1.2.

Geschlossen wird ein Nachrichtenkanal mittels der Funktion *canChannelClose*. Die Funktion sollte immer dann aufgerufen werden, wenn ein Kanal nicht mehr benötigt wird.

3.1.3.1 Empfang von CAN-Nachrichten

Die einfachste Art empfangene Nachrichten aus dem Empfangspuffer zu lesen ist ein Aufruf der Funktion *canChannelReadMessage*. Sind keine Nachrichten im Empfangspuffer verfügbar, wartet die Funktion bis eine neue Nachricht vom Bus empfangen wurde oder die im Parameter *dwMsTimeout* angegebene Wartezeit abgelaufen ist. Eine Nachricht hat dabei immer einen zugewiesenen Nachrichtentyp (*data*, *info*, *error*, *status*, *timer-overflow*). (siehe §5.2.5 CANMSGINFO)

Die Funktion *canChannelPeekMessage* liest ebenfalls die nächste Nachricht aus dem Empfangspuffer. Anders als *canChannelReadMessage* wartet die Funktion jedoch nicht bis eine Nachricht zum Lesen bereitsteht, sondern kehrt sofort mit einem entsprechenden Fehlercode zum aufrufenden Programm zurück.

Mit der Funktion *canChannelWaitRxEvent* kann auf eine neue Empfangsnachricht, bzw. auf das Eintreffen des Empfangs- Events gewartet werden. Das Empfangs-Event wird ausgelöst, wenn der Empfangspuffer mindestens die bei Aufruf von *canChannelInitialize* in *wRxThreshold* angegebene Anzahl Nachrichten enthält.

Nachfolgendes Codefragment zeigt eine mögliche Verwendung der Funktionen *canChannelWaitRxEvent* und *canChannelPeekMessage*.

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

    while (canChannelWaitRxEvent(hCanChn, INFINITE) == VCI_OK)
    {
        while (canChannelPeekMessage(hCanChn, &sCanMsg) == VCI_OK)
        {
            // Verarbeitung der Nachricht
        }
    }

    return 0;
}
```

Beachten Sie, dass die Thread- Prozedur nur dann endet, wenn die Funktion *canChannelWaitRxEvent* einen Fehlercode ungleich **VCI_OK** zurückliefert. Alle Nachrichtenkanal-spezifischen Funktionen liefern bei korrektem Aufruf jedoch nur dann einen Fehlercode ungleich **VCI_OK** zurück, wenn ein schwerwiegendes Problem aufgetreten ist.

Ein programmgesteuerter „normaler“ Abbruch der Thread- Prozedur aus vorigem Beispiel scheint daher nicht möglich zu sein. Es besteht jedoch die Möglichkeit, den Handle vom Nachrichtenkanal von einem anderen Thread aus zu schließen, wodurch alle momentan ausstehenden Funktionsaufrufe bzw. alle neuen Aufrufe mit einem Fehlercode ungleich **VCI_OK** enden. Nachteilig ist jedoch, dass dabei ein eventuell parallel laufender Sende- Thread ebenfalls beeinflusst wird.

3.1.3.2 Senden von CAN-Nachrichten

Die simpelste Art Nachrichten auf den Bus zu senden besteht in einem Aufruf der Funktion *canChannelSendMessage*. Die Funktion wartet bis ein freier Eintrag im Sendepuffer des Nachrichtenkanals verfügbar ist und schreibt anschließend die Nachricht in diesen freien Eintrag. Konnte die Nachricht innerhalb der in *dwMsTimeout* angegebenen Zeit in den Sendepuffer eingetragen werden, liefert die Funktion den Wert **VCI_OK** zurück. Ist die angegebene Zeitspanne verstrichen, ohne dass die Nachricht in den Sendepuffer geschrieben wurde, liefert die Funktion den Wert **VCI_E_TIMEOUT** zurück.

Die Funktion *canChannelPostMessage* schreibt ebenfalls eine Nachricht in den Sendepuffer des Nachrichtenkanals. Anders als *canChannelSendMessage* wartet die Funktion jedoch nicht bis ausreichend Platz verfügbar ist, sondern kehrt mit einem Fehlercode zum aufrufenden Programm zurück, falls die Nachricht nicht in den Sendepuffer eingetragen werden konnte.

Mit der Funktion *canChannelWaitTxEvent* kann auf das Eintreffen des Sende-Events gewartet werden. Der Sende- Event wird ausgelöst, wenn im Sendepuffer mindestens Platz für die bei Aufruf von *canChannelInitialize* in *wTxThreshold* angegebene Anzahl Nachrichten ist.

Nachfolgendes Codefragment zeigt eine mögliche Verwendung der Funktionen *canChannelWaitTxEvent* und *canChannelPostMessage*.

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

3.1.3.3 Verzögertes Senden von CAN-Nachrichten

Anschlüsse bei denen das Bit `CAN_FEATURE_DELAYEDTX` im Feld `dwFeatures` der Struktur **CANCAPABILITIES** gesetzt ist unterstützen das verzögerte Senden von Nachrichten. Durch diese verzögerte Senden kann z.B. verhindert werden, dass ein am CAN-Bus angeschlossenes Gerät zu viele Daten in zu kurzer Zeit erhält, was bei „langsamen“ Geräten zu Datenverlust führen kann.

Um eine CAN-Nachricht verzögert zu senden, wird im Feld `dwTime` der Struktur **CANMSG** die Zeit in Ticks angegeben, die mindestens verstreichen muss, bevor die Nachricht an den CAN-Controller weitergegeben wird. Der Wert 0 bewirkt keine Sendeverzögerung, die maximal mögliche Verzögerungszeit lässt sich aus dem Feld `dwDtxMaxTicks` der Struktur **CANCAPABILITIES** entnehmen. Die Auflösung eines Ticks in Sekunden berechnet sich aus den Werten in den Feldern `dwClockFreq` und `dwDtxDivisor` der Struktur **CANCAPABILITIES** nach folgender Formel:

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

Die angegebene Verzögerungszeit stellt nur ein Minimum dar, da nicht garantiert werden kann, dass die Nachricht nach Ablauf der Zeit auf den Bus gesendet wird. Weiterhin ist zu beachten, dass bei Verwendung mehrerer Nachrichtenkanäle an einem CAN-Anschluss die angegebene Zeit generell nicht eingehalten werden kann, da der Verteiler alle Kanäle parallel bearbeitet. Applikationen die eine genaue zeitliche Abfolge benötigen, müssen den CAN-Anschluss daher exklusiv verwenden.

3.1.4 Zyklische Sendeliste

Mit der optional vorhandenen zyklischen Sendeliste können pro CAN-Anschluss bis zu 16 Nachrichten zyklisch, d.h. wiederkehrend in bestimmten Zeitintervallen gesendet werden. Dabei besteht die Möglichkeit, dass nach jedem Sendevorgang ein bestimmter Teil einer Nachricht automatisch inkrementiert wird.

Der Zugriff auf die zyklische Sendeliste ist, wie bei der Steuereinheit ebenfalls auf eine einzige Applikation begrenzt. Sie kann also nicht von mehreren Programmen gleichzeitig benutzt werden.

Geöffnet wird die Sendeliste durch Aufruf der Funktion *canSchedulerOpen*. Liefert die Funktion einen Fehlercode entsprechend „Zugriff verweigert“ zurück, wird die Sendeliste bereits von einem anderen Programm verwendet. Mittels der Funktion *canSchedulerClose* wird eine geöffnete Sendeliste wieder geschlossen und für andere Applikationen freigegeben.

Mit der Funktion *canSchedulerAddMessage* wird ein Nachrichtenobjekt der Liste hinzugefügt. Die Funktion erwartet neben dem Handle auf die Sendeliste einen Zeiger auf eine Struktur vom Typ *CANCYCLICTXMSG* die das Nachrichtenobjekt spezifiziert, das der Liste hinzugefügt werden soll. Bei erfolgreicher Ausführung liefert die Funktion den Listenindex des hinzugefügten Objekts zurück.

Die Zykluszeit einer Nachricht wird in Anzahl Ticks im Feld *wCycleTime* der Struktur *CANCYCLICTXMSG* angegeben. Der Wert in diesem Feld muss größer 0 sein und darf den Wert im Feld *dwCmsMaxTicks* der Struktur *CANCAPABILITIES* nicht überschreiten.

Die Dauer eines Ticks, bzw. die Zykluszeit t_z der Sendeliste kann mittels der Felder *dwClockFreq* und *dwCmsDivisor* der Struktur *CANCAPABILITIES* nach folgender Formel berechnet werden.

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

Die Sendetask der zyklischen Sendeliste unterteilt die ihr zur Verfügung stehende Zeit in einzelne Abschnitte, so genannte Zeitschlitz. Die Dauer eines Zeitschlitzes entspricht dabei der Dauer eines Ticks bzw. der Zykluszeit. Die Anzahl kann dem Feld *dwCmsMaxTicks* der Struktur *CANCAPABILITIES* entnommen werden.

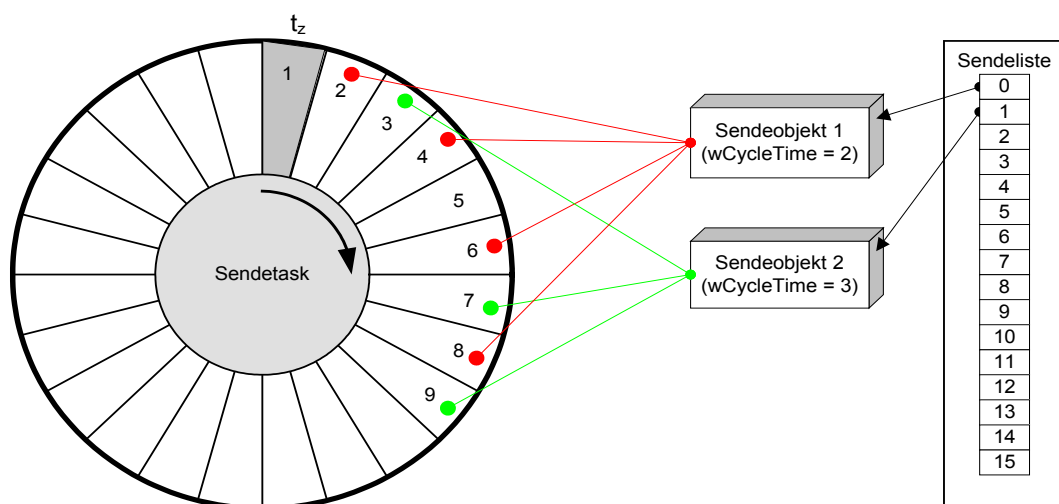


Bild 3-6: Sendetask der zyklischen Sendeliste

Die Sendetask kann pro Tick immer nur eine Nachricht versenden. Ein Zeitschlitz enthält also nur ein Sendeobjekt. Wird das erste Sendeobjekt mit einer Zykluszeit von 1 angelegt sind alle Zeitschlitze belegt und es können keine weiteren Objekte eingerichtet werden. Je mehr Sendeobjekte angelegt werden, desto größer muss deren Zykluszeit gewählt werden. Die Regel hierzu lautet: Die Summe aller $1/wCycleTime$ muss kleiner sein als eins. Soll z.B. eine Nachricht alle 2 Ticks und eine weitere Nachricht alle 3 Ticks gesendet werden, so ergibt $1/2 + 1/3 = 5/6 = 0,833$ und damit einen noch zulässigen Wert.

Das Beispiel in Bild 3-6 zeigt zwei Sendeobjekte mit den Zykluszeiten 2 und 3. Beim Einrichten von Sendeobjekt 1 werden die Zeitschlitze 2, 4, 6, 8, usw. belegt. Beim anschließenden Einrichten des zweiten Objekts (Zykluszeit = 3) kommt es in den Zeitschlitzen 6, 12, 18, usw. zu Kollisionen, da diese Zeitschlitze bereits von Objekt 1 belegt sind.

Derartige Kollisionen werden von der Sendetask aufgelöst, indem diese den jeweils nächsten freien Zeitschlitz verwendet. Objekt 2 aus vorigem Beispiel belegt damit die Zeitschlitze 3, 7, 9, 13, 19, usw.. Die Zykluszeit vom zweiten Objekt wird also nicht immer exakt eingehalten, was im Beispiel zu einer Ungenauigkeit von ± 1 Tick führt.

Die zeitliche Genauigkeit mit der die einzelnen Objekte versendet werden hängt auch von der allgemeinen Buslast ab, da der Sendezeitpunkt mit steigender Buslast immer ungenauer wird. Generell gilt, dass die Genauigkeit mit steigender Buslast, kleineren Zykluszeiten und steigender Anzahl von Sendeobjekte abnimmt.

Das Feld *bIncrMode* der Struktur **CANCYCLICTXMSG** bestimmt, ob ein Teil der Nachricht nach jedem Sendevorgang automatisch inkrementiert wird. Wird hier der Wert **CAN_CTXMSG_INC_NO** angegeben, bleibt der Inhalt unverändert. Beim Wert **CAN_CTXMSG_INC_ID** wird das Feld *dwMsgId* der Nachricht nach jedem Sendevorgang automatisch um 1 erhöht. Erreicht das Feld *dwMsgId* den Wert 2048 (11-bit ID) bzw. 536.870.912 (29-bit ID) erfolgt automatisch ein Überlauf auf 0.

Beim Wert **CAN_CTXMSG_INC_8** oder **CAN_CTXMSG_INC_16** im Feld *bIncrMode*, wird ein einzelner 8- oder 16-Bit Wert im Datenfeld *abData[]* der Nachricht inkrementiert. Das Feld *bByteIndex* der Struktur **CANCYCLICTXMSG** legt dabei den Index des Datenfeldes fest. Bei 16-Bit Werten liegt das niederwertige Byte (LSB) im Datenfeld *abData[bByteIndex]* und das höherwertige Byte (MSB) im Feld *abData[bByteIndex+1]*. Wird der Wert 255 (8-Bit) bzw. 65535 (16-Bit) erreicht, erfolgt ein Überlauf auf 0.

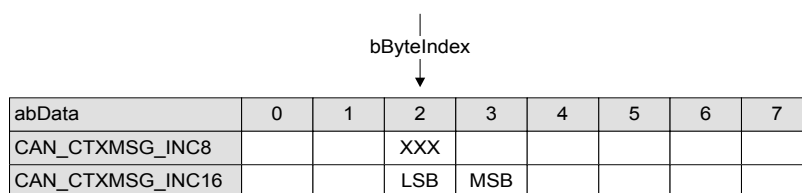


Bild 3-7: Auto-Inkrement von Datenfeldern

Mit der Funktion *canSchedulerRemMessage* kann ein Sendeobjekt wieder aus der Liste entfernt werden. Die Funktion erwartet neben dem Handle der Sendeliste den von der Funktion *canSchedulerAddMessage* gelieferten Listenindex des zu entfernenden Objekts.

Ein neu eingerichtetes Sendeobjekt befindet sich zunächst im Ruhezustand und wird vom Sendetask so lange nicht versendet, bis dieses durch einen Aufruf der Funktion *canSchedulerStartMessage* gestartet wird. Stoppen lässt sich der Sendevorgang für ein Objekt mittels der Funktion *canSchedulerStopMessage*.

Den momentanen Zustand der Sendetask und aller eingerichteten Sendeobjekte liefert die Funktion *canSchedulerGetStatus*. Den hierzu notwendigen Speicher stellt die Applikation in Form einer Struktur vom Typ *CANSCHEDULERSTATUS* zur Verfügung. Nach erfolgreicher Ausführung der Funktion befindet sich der Zustand der Sendeliste und Sendeobjekte in den Feldern *bTaskStat* und *abMsgStat*.

Um den Zustand eines einzelnen Sendeobjekts zu ermitteln, wird der von der Funktion *canSchedulerAddMessage* gelieferte Listenindex als Index in die Tabelle *abMsgStat* verwendet, d.h. *abMsgStat[Index]* enthält den Zustand des Objekts mit dem angegebenen Index.

Normalerweise ist die Sendetask nach dem Öffnen der Sendeliste deaktiviert. Die Sendetask versendet im deaktivierten Zustand prinzipiell keine Nachrichten, selbst dann nicht, wenn die Liste eingerichtete und gestartete Sendeobjekte enthält.

Aktivieren bzw. deaktivieren lässt sich die Sendetask einer Sendeliste durch Aufruf der Funktion *canSchedulerActivate*.

Die Funktion kann zum gleichzeitigen Start aller Sendeobjekte verwendet werden, indem zunächst alle Sendeobjekte mittels *canSchedulerStartMessage* gestartet werden und erst anschließend die Sendetask aktiviert wird. Ein gleichzeitiger Stopp aller Sendeobjekte ist ebenfalls möglich. Hierzu muss einfach die Sendetask deaktiviert werden.

Zurücksetzen lässt sich die Sendeliste mit der Funktion *canSchedulerReset*. Die Funktion stoppt die Sendetask und entfernt alle registrierten Sendeobjekte aus der angegebenen zyklischen Sendeliste.

3.2 Zugriff auf den LIN-Bus

3.2.1 Übersicht

Nachfolgende Abbildung zeigt beispielhaft eine IXXAT- Interfacekarte mit einem LIN- und zwei CAN-Anschlüssen.

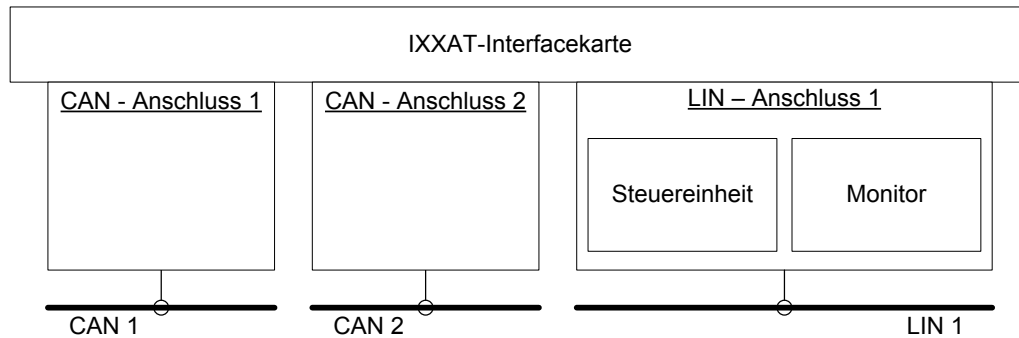


Bild 3-8: IXXAT- Interfacekarte mit einem LIN- und 2 CAN-Anschlüssen.

Wie in Bild 3-8 für den LIN-Anschluss 1 dargestellt ist, setzt sich dieser aus einer Steuereinheit und einem oder mehreren Nachrichtenmonitoren zusammen. Die neben dem LIN-Anschluss vorhandenen CAN-Anschlüsse spielen bei der weiteren Betrachtung keine Rolle.

Zugriff auf die Steuereinheit des LIN-Anschlusses erhält man mit der Funktion *linControlOpen*. Die Funktion *linMonitorOpen* öffnet einen Nachrichtenmonitor.

Beide Funktionen erwarten im ersten Parameter das Handle des Gerätes bzw. der Interfacekarte und im zweiten Parameter die Nummer des LIN-Anschlusses. Für Anschluss 1 wird dabei die Nummer 0, für Anschluss 2 die Nummer 1, usw. angegeben.

Zur Einsparung von Systemressourcen kann nach dem Öffnen einer Komponente das Handle des Gerätes bzw. der Interfacekarte wieder freigegeben werden. Für die weiteren Zugriffe auf den Anschluss ist nur noch das Handle der Steuereinheit bzw. des Nachrichtenmonitors erforderlich.

Die Funktionen *linControlOpen* und *linMonitorOpen* lassen sich so aufrufen, dass dem Benutzer ein Dialogfenster zur Auswahl des Gerätes bzw. der Interfacekarte und des LIN-Anschlusses präsentiert wird. Dies erreicht man, indem für die Anschlussnummer der Wert 0xFFFFFFFF angegeben wird. In diesem Fall erwarten die Funktionen im ersten Parameter nicht das Handle des Gerätes bzw. der Interfacekarte, sondern das Handle vom übergeordneten Fenster (Parent) oder den Wert NULL, falls kein übergeordnetes Fenster verfügbar ist.

Bei erfolgreicher Ausführung liefern alle drei Funktionen einen Handle auf die geöffnete Komponente zurück. Tritt ein Fehler oder ein Zugriffskonflikt auf, liefern die Funktionen einen entsprechenden Fehlercode zurück.

Wird eine geöffnete Komponente nicht mehr benötigt kann diese durch Aufruf einer der Funktionen *linControlClose* und *linMonitorClose* wieder geschlossen werden.

Welche Möglichkeiten ein LIN-Anschluss bietet, bzw. wie der LIN-Anschluss zu verwenden ist, wird in den folgenden Unterkapiteln genauer beschrieben.

3.2.2 Steuereinheit

Die Steuereinheit stellt Funktionen zur Konfiguration des LIN-Controllers und dessen Übertragungseigenschaften sowie Funktionen zur Abfrage des aktuellen Controllerzustandes bereit.

Die Komponente ist so konzipiert, dass sie immer nur von einer Applikation geöffnet werden kann. Gleichzeitiges mehrfaches Öffnen der Komponente durch unterschiedliche Programme ist nicht möglich. Dadurch lassen sich Situationen vermeiden, bei denen z.B. eine Applikation den LIN-Controller starten, eine andere aber stoppen möchte.

Geöffnet wird die Steuereinheit durch Aufruf der Funktion *linControlOpen*. Liefert die Funktion einen Fehlercode entsprechend „Zugriff verweigert“ zurück, wird der LIN-Controller bereits von einem anderen Programm verwendet.

Dies stellt in der Regel nur dann ein Problem dar, wenn eine Applikation nicht ohne direkte Kontrolle vom LIN-Controller ausführbar ist. In allen anderen Fällen sollte die Applikation jedoch normal weiterarbeiten können.

Mittels der Funktion *linControlClose* wird eine geöffnete Steuereinheit wieder geschlossen und damit für andere Applikationen verfügbar. Ein Programm sollte die Steuereinheit daher nur freigeben, wenn diese nicht mehr benötigt wird.

3.2.2.1 Kontrollerzustände

Die folgende Abbildung zeigt die verschiedenen Zustände eines Controllers.

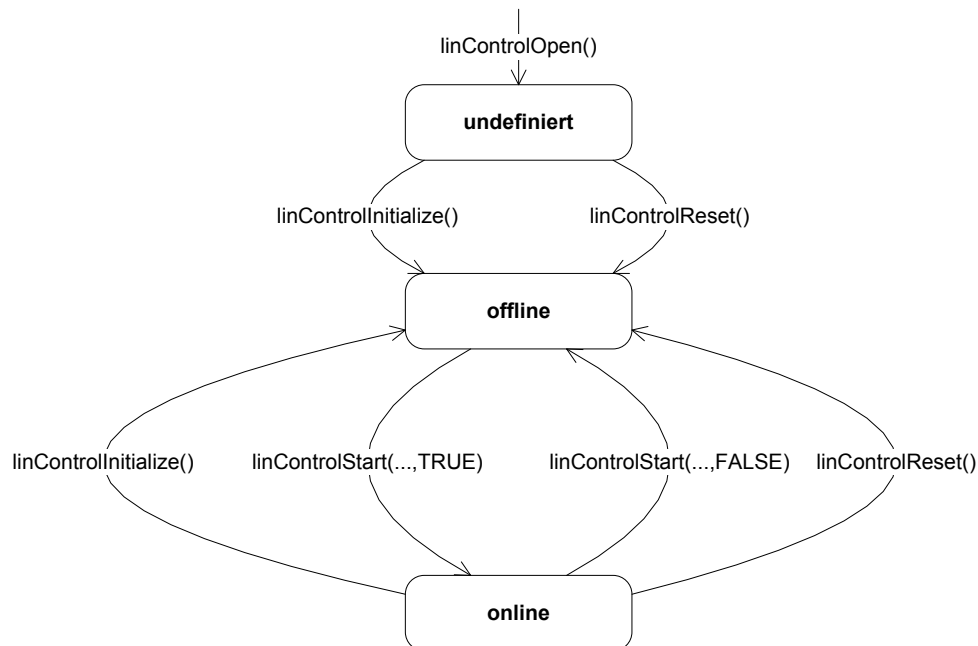


Bild 3-9: Kontrollerzustände

Zugriff auf den Bus

Nach dem Öffnen der Steuereinheit befindet sich der Controller normalerweise in einem nicht initialisiertem Zustand. Dieser Zustand wird durch Aufruf der Funktion *linControlInitialize* verlassen. Danach befindet sich der LIN-Controller im Zustand „offline“.

Mittels *linControlInitialize* wird die Betriebsart und Bitrate vom LIN-Controller eingestellt. Die Übertragungsrate in Bit pro Sekunde wird im Parameter *wBtrate* angegeben. Gültige Werte für die Bitrate liegen zwischen 1000 und 20000, bzw. zwischen **LIN_BITRATE_MIN** und **LIN_BITRATE_MAX**. Unterstützt der Anschluss die automatische Bitratenerkennung, kann diese mit **LIN_BITRATE_AUTO** im Feld *wBtrate* aktiviert werden. Nachfolgende Tabelle zeigt einige empfohlene Bitraten:

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

Gestartet, bzw. gestoppt wird der LIN-Controller durch Aufruf der Funktion *linControlStart*. Nach erfolgreichem Aufruf der Funktion mit dem Wert TRUE im Parameter *fStart*, befindet sich der Controller im Zustand „online“. In diesem Zustand ist der LIN-Controller aktiv mit dem Bus verbunden. Eingehende LIN-Nachrichten werden hierbei an alle aktiven Nachrichtenmonitore weitergeleitet.

Ein Aufruf der Funktion *linControlStart* mit dem Wert FALSE im Parameter *fStart* schaltet den LIN-Controller „offline“. Der Nachrichtentransport wird dabei unterbrochen und der Controller deaktiviert.

Die Funktion *linControlReset* schaltet den LIN-Controller ebenfalls in den Zustand „offline“. Zusätzlich setzt die Funktion die Controllerhardware zurück. Beachten Sie, dass es hierbei zu fehlerhaften Telegrammen auf dem Bus kommen kann, wenn bei Aufruf der Funktion ein Sendevorgang mitten in der Übertragung abgebrochen wird.

3.2.2.2 Senden von LIN-Nachrichten

Nachrichten lassen sich mit der Funktion *linControlWriteMessage* entweder direkt senden oder in eine Antworttabelle im Controller eingetragen. Zum besseren Verständnis betrachten Sie bitte folgende Abbildung.

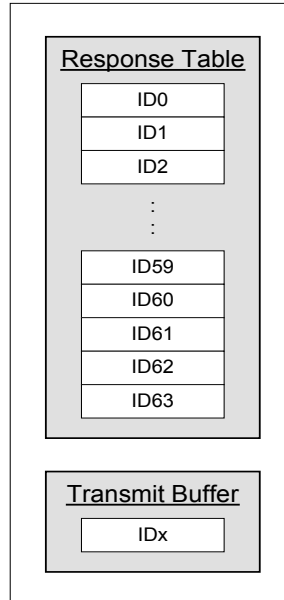


Bild 3-10: Interner Aufbau der Steuereinheit

Die Steuereinheit enthält intern eine Antworttabelle (Response Table) mit den Antwortdaten für die vom Master aufgeschalteten IDs. Erkennt der Controller eine ihm zugeordnete und vom Master gesendete ID auf dem Bus, überträgt er die in der Tabelle an entsprechender Position eingetragenen Antwortdaten. Der Inhalt der Tabelle kann mittels der Funktion *linControlWriteMessage* geändert, bzw. aktualisiert werden, indem im Parameter *fSend* der Wert FALSE angegeben wird. Die Nachricht mit den Antwortdaten im Feld *abData* der Struktur *LINMSG* wird der Funktion dabei im Parameter *pLinMsg* übergeben. Beachten Sie, dass die Nachricht vom Typ *LIN_MSGTYPE_DATA* ist und eine gültige ID im Bereich 0 bis 63 enthält. Die Tabelle muss unabhängig von der Betriebsart (Master oder Slave) noch vor dem Start des Controllers initialisiert werden, kann danach jedoch jederzeit aktualisiert werden ohne dass der Controller gestoppt wird. Geleert wird die Antworttabelle bei Aufruf der Funktion *linControlReset*.

Mittels der Funktion *linControlWriteMessage* lassen sich Nachrichten auch direkt auf den Bus senden. Hierzu muss *fSend* auf den Wert TRUE gesetzt werden. In diesem Fall wird die Nachricht nicht in die Antworttabelle eingetragen, sondern in den Sendepuffer (Transmit Buffer) und vom Controller auf den Bus geschaltet, sobald dieser frei ist.

Wird der Anschluss als Master betrieben, können neben den Steuernachrichten *LIN_MSGTYPE_SLEEP* und *LIN_MSGTYPE_WAKEUP* auch Datennachrichten vom Typ *LIN_MSGTYPE_DATA* direkt versendet werden.

Ist der Anschluss als Slave konfiguriert, lassen sich nur `LIN_MSGTYPE_WAKEUP` Nachrichten senden. Bei allen anderen Nachrichtentypen liefert die Funktion einen Fehlercode zurück.

Nachrichten vom Typ `LIN_MSGTYPE_SLEEP` erzeugen ein „Goto-Sleep“ Frame auf dem Bus, Nachrichten vom Typ `LIN_MSGTYPE_WAKEUP` dagegen einen WAKEUP Frame. Weitere Informationen hierzu findet sich in der LIN-Spezifikation im Kapitel „Network Management“.

In der Master-Betriebsart dient die Funktion *linControlWriteMessage* auch zum Aufschalten von IDs. Hierzu wird eine `LIN_MSGTYPE_DATA` Nachricht mit gültiger ID und Datenlänge gesendet, bei der das Bit `uMsgInfo.Bits.ido` gleichzeitig den Wert 1 hat. Weitere Informationen finden sich im Kapitel 5.3.4 bei der Beschreibung der Datenstruktur *LINMSGINFO*.

Die Funktion *linControlWriteMessage* kehrt unabhängig vom Wert des Parameters *fSend* immer sofort zum aufrufenden Programm zurück, ohne auf den Abschluss der Übertragung zu warten. Wird die Funktion erneut aufgerufen noch bevor die letzte Übertragung abgeschlossen bzw. bevor der Sendepuffer frei ist, kehrt die Funktion mit einem entsprechenden Fehlercode zurück.

3.2.3 Nachrichtenmonitor

Erzeugt, bzw. geöffnet wird ein Nachrichtenmonitor durch Aufruf der Funktion *linMonitorOpen*. Der Parameter *fExclusive* bestimmt dabei, ob der Anschluss exklusiv verwendet werden soll. Wird hier der Wert TRUE angegeben, können nach erfolgreicher Ausführung der Funktion keine weiteren Nachrichtenmonitore mehr geöffnet werden. Wird der LIN-Anschluss nicht exklusiv verwendet, lassen sich prinzipiell beliebig (durch den Speicher begrenzt) viele Nachrichtenmonitore einrichten.

Bei exklusiver Verwendung des Anschlusses ist der Nachrichtenmonitor direkt mit dem LIN-Controller verbunden. Folgende Abbildung zeigt diese Konfiguration.

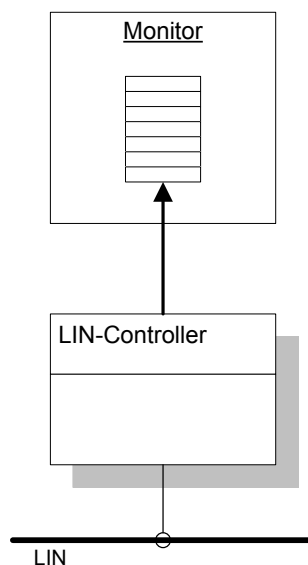


Bild 3-11: Exklusive Verwendung eines LIN-Anschlusses

Bei nicht exklusiver Verwendung des Anschlusses (*fExclusive* = FALSE) wird ein Verteiler zwischen Controller und die Nachrichtenmonitore geschaltet.

Der Verteiler leitet eingehenden Nachrichten vom LIN-Controller an alle Monitore weiter. Die Verteilung der Nachrichten erfolgt dabei so, dass kein Monitor bevorzugt behandelt wird. Nachfolgende Abbildung zeigt eine Konfiguration mit drei Monitoren an einem LIN-Anschluss.

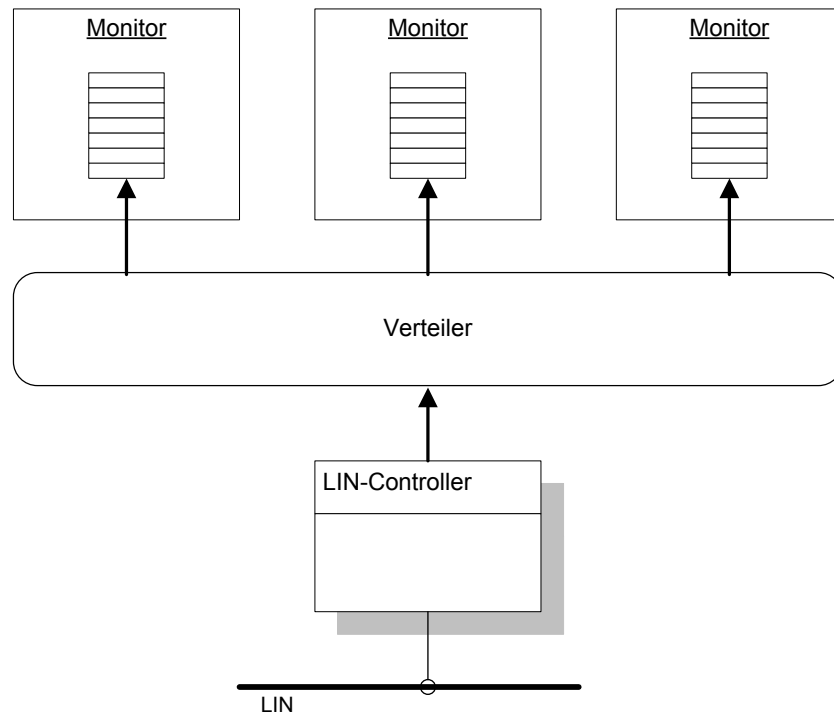


Bild 3-12: LIN-Nachrichtenverteiler

Nachdem ein Nachrichtenmonitor geöffnet wurde, muss dieser initialisiert werden. Hierzu dient die Funktion *linMonitorInitialize*. Als Eingabeparameter erwartet die Funktion die Größe vom Empfangspuffer in Anzahl LIN-Nachrichten sowie die Schwellwerte für das Empfangs- Event.

Der für den Empfangspuffer reservierte Speicher stammt aus einem begrenzten Systemspeicherpool. Die einzelnen Puffer eines Nachrichtenmonitors sollten daher nicht mehr als ca. 2000 Nachrichten umfassen.

Das Empfangs- Event eines Monitors wird ausgelöst, wenn der Empfangspuffer mindestens die in *wRxThreshold* angegebene Anzahl Nachrichten enthält.

Ist der Monitor eingerichtet, kann er mittels der Funktion *linMonitorActivate* aktiviert bzw. deaktiviert werden. Aktiviert wird der Monitor, wenn die Funktion mit dem Wert TRUE im Parameter *fEnable* aufgerufen wird. Ein Aufruf der Funktion mit dem Wert FALSE im Parameter *fEnable*, deaktiviert den Monitor. Standardmäßig ist ein Nachrichtenmonitor nach dem Öffnen deaktiviert.

Nachrichten werden nur dann vom LIN-Controller empfangen, wenn der Nachrichtenmonitor aktiv ist. Außerdem muss der LIN-Controller gestartet sein, andernfalls erfolgt kein Nachrichtentransfer zwischen LIN-Bus und Monitor.

Geschlossen wird ein Nachrichtenmonitor mittels der Funktion *linMonitorClose*. Die Funktion sollte immer dann aufgerufen werden, wenn dieser nicht mehr benötigt wird.

3.2.3.1 Empfang von LIN-Nachrichten

Die einfachste Art empfangene Nachrichten aus dem Empfangspuffer zu lesen ist ein Aufruf der Funktion *linMonitorReadMessage*. Sind keine Nachrichten im Empfangspuffer verfügbar, wartet die Funktion bis eine neue Nachricht vom Bus empfangen wurde oder die im Parameter *dwMsTimeout* angegebene Wartezeit abgelaufen ist.

Die Funktion *linMonitorPeekMessage* liest die nächste Nachricht aus dem Empfangspuffer. Anders als *linMonitorReadMessage* wartet die Funktion jedoch nicht auf eine neue Nachricht, sondern kehrt sofort mit einem entsprechenden Fehlercode zum aufrufenden Programm zurück, wenn keine Nachricht im Empfangspuffer bereitliegt.

Mit der Funktion *linMonitorWaitRxEvent* kann auf eine neue Empfangsnachricht, bzw. auf das Eintreffen des Empfangs-Events gewartet werden. Das Empfangs-Event wird ausgelöst, wenn der Empfangspuffer mindestens die bei Aufruf von *linMonitorInitialize* in *wThreshold* angegebene Anzahl Nachrichten enthält.

Nachfolgendes Codefragment zeigt eine mögliche Verwendung der Funktionen *linMonitorWaitRxEvent* und *linMonitorReadMessage*.

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

    while (linMonitorWaitRxEvent(hLinMon, INFINITE) == VCI_OK)
    {
        while (linMonitorPeekMessage(hLinMon, &sLinMsg) == VCI_OK)
        {
            // Verarbeitung der Nachricht
        }
    }

    return 0;
}
```

Beachten Sie, dass die Thread- Prozedur nur dann endet, wenn die Funktion *linMonitorWaitRxEvent* einen Fehlercode ungleich *VCI_OK* zurückliefert. Alle Nachrichtenmonitor-spezifischen Funktionen liefern bei korrektem Aufruf jedoch nur dann einen Fehlercode ungleich *VCI_OK* zurück, wenn ein schwerwiegendes Problem aufgetreten ist.

Ein programmgesteuerter „normaler“ Abbruch der Thread- Prozedur aus vorigem Beispiel scheint daher nicht möglich zu sein. Es besteht jedoch die Möglichkeit, das Handle vom Nachrichtenmonitor von einem anderen Thread aus zu schließen, wodurch alle momentan ausstehenden Funktionsaufrufe bzw. alle neuen Aufrufe mit einem Fehlercode ungleich *VCI_OK* enden.

4 Schnittstellenbeschreibung

4.1 Generelle Funktionen

4.1.1 `vciInitialize`

Die Funktion initialisiert das VCI für den aufrufenden Prozess. Die vollständige Syntax der Funktion lautet:

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

Parameter:

Die Funktion hat keine Parameter.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert `VCI_OK`, andernfalls einen Fehlercode ungleich `VCI_OK` zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *`vciFormatError`*.

Bemerkungen:

Die Funktion muss zu Beginn eines Programms aufgerufen werden um die DLL für den aufrufenden Prozess zu initialisieren.

4.1.2 `vciFormatError`

Die Funktion wandelt einen VCI Fehlercode in einen für Benutzer lesbaren Text, bzw. in eine Zeichenkette um. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hrError

[in] Fehlercode, der in Text umgewandelt werden soll.

pszText

[out] Zeiger auf einen Puffer für den Textstring. Der Puffer muss Platz für mindestens `VCI_MAX_ERRSTRLEN` Zeichen zur Verfügung stellen. Die Funktion speichert den Fehlertext einschließlich eines abschließenden 0-Zeichen im angegebenen Speicherbereich.

Rückgabewert:

Die Funktion hat keinen Rückgabewert.

4.1.3 vciDisplayError

Die Funktion zeigt ein Meldungsfenster entsprechend dem angegebenen Fehlercode auf dem Bildschirm an. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hwndParent

[in] Handle vom übergeordneten Fenster. Wird hier der Wert NULL angegeben, hat das Meldungsfenster kein übergeordnetes Fenster.

pszCaption

[in] Zeiger auf eine 0-terminierte Zeichenkette mit dem Text für die Titelzeile des Meldungsfensters. Wird hier der Wert NULL angegeben, wird ein vordefinierter Titelzeilentext angezeigt.

hrError

[in] Fehlercode, für den die Meldung angezeigt werden soll.

Rückgabewert:

Die Funktion hat keinen Rückgabewert.

Bemerkungen:

Wird im Parameter *hrError* der Wert -1 angegeben, ermittelt die Funktion den zuletzt aufgetretenen Fehlercode mittels der API Funktion *GetLastError* und zeigt ein entsprechendes Meldungsfenster an. Wird im Parameter *hrError* der Wert 0 bzw. NO_ERROR übergeben, wird kein Meldungsfenster angezeigt.

4.1.4 vciGetVersion

Die Funktion ermittelt die Version des installierten VCI. Die vollständige Syntax der Funktion lautet:

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

Parameter:

pdwMajorVersion

[out] Adresse einer Variable vom Typ UINT32. Bei erfolgreicher Ausführung liefert die Funktion die Hauptversionsnummer des VCI in dieser Variable zurück.

pdwMinorVersion

[out] Adresse einer Variable vom Typ UINT32. Bei erfolgreicher Ausführung liefert die Funktion die Nebenversionsnummer des VCI in dieser Variable zurück.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion ***vciFormatError***.

Bemerkungen:

Die Funktion kann zu Beginn eines Programms aufgerufen werden um zu überprüfen, ob das aktuell installierte VCI der Applikation genügt.

Um die komplette VCI version number (A.B.C.D) auszulesen, muss die Version-Resource der vciapi.dll mit der Windows API Funktion ‚GetVersionInfo‘ abgefragt werden.

4.1.5 vciLuidToChar

Die Funktion wandelt eine lokal eindeutige Kennzahl (***VCIID***) in eine Zeichenkette um. Die vollständige Syntax der Funktion lautet:

<pre>HRESULT VCIAPI vciLuidToChar (REFVCIID rVciid PCHAR pszLuid LONG cbSize);</pre>

Parameter:

rVciid

[in] Referenz auf die lokal eindeutige VCI Kennzahl, die in eine Zeichenkette umgewandelt werden soll.

pszLuid

[out] Zeiger auf einen Puffer für die 0-terminierte Zeichenkette. Bei erfolgreicher Ausführung speichert die Funktion die umgewandelte VCI Kennzahl im hier angegebenen Speicherbereich.

cbSize

[in] Größe des in *pszLuid* angegebene Puffers in Bytes. Um die komplette LUID aufzunehmen sollte der Puffer Platz für 17 Zeichen bieten.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen der folgenden Fehlercodes zurück:

VCI_E_INVALIDARG	Der Parameter <i>pszLuid</i> zeigt auf einen ungültigen Puffer
VCI_E_BUFFER_OVERFLOW	Der in <i>pszLuid</i> angegebene Puffer ist nicht groß genug für die Zeichenkette.

4.1.6 vciCharToLuid

Die Funktion wandelt eine 0-terminierte Zeichenkette in eine lokal eindeutige VCI Kennzahl (**VCIID**) um. Die vollständige Syntax der Funktion lautet:

```
HRESULT VCI_API vciCharToLuid (  
    PCHAR    pszLuid  
    PVCIID   pvciid );
```

Parameter:

pszLuid

[in] Zeiger auf die umzuwandelnde 0-terminierte Zeichenkette.

pVciid

[out] Adresse einer Variable vom Typ **VCIID**. Bei erfolgreicher Ausführung liefert die Funktion die umgewandelte Kennzahl in dieser Variable zurück.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen der folgenden Fehlercodes zurück:

VCI_E_INVALIDARG	Parameter <i>pszLuid</i> oder <i>pVciid</i> zeigt auf einen ungültigen Puffer.
VCI_E_FAIL	Die in <i>pszLuid</i> angegebene Zeichenkette konnte nicht in eine gültige Kennzahl umgewandelt werden.

4.1.7 vciGuidToChar

Die Funktion wandelt eine global eindeutige Kennzahl (GUID) in eine Zeichenkette um. Die vollständige Syntax der Funktion lautet:

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

Parameter:

rGuid

[in] Referenz auf die global eindeutige Kennzahl, die in eine Zeichenkette umgewandelt werden soll.

pszGuid

[out] Zeiger auf den Puffer für die 0-terminierte Zeichenkette. Bei erfolgreicher Ausführung speichert die Funktion die umgewandelte GUID im angegebenen Speicherbereich.

cbSize

[in] Größe des in *pszGuid* angegebenen Puffers in Bytes. Um die komplette GUID aufzunehmen sollte der Puffer Platz für 39 Zeichen bieten.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen der folgenden Fehlercodes zurück:

VCI_E_INVALIDARG	Der Parameter <i>pszLuid</i> zeigt auf einen ungültigen Puffer
VCI_E_BUFFER_OVERFLOW	Der in <i>pszLuid</i> angegebene Puffer ist nicht groß genug für die Zeichenkette.

4.1.8 vciCharToGuid

Die Funktion wandelt eine 0-terminierte Zeichenkette in eine global eindeutige Kennzahl (GUID) um. Die vollständige Syntax der Funktion lautet:

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

Parameter:

pszGuid

[in] Zeiger auf die umzuwandelnde 0-terminierte Zeichenkette.

pGuid

[out] Adresse einer Variable vom Typ GUID. Bei erfolgreicher Ausführung liefert die Funktion die umgewandelte Kennzahl in dieser Variable zurück.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen der folgenden Fehlercodes zurück:

VCI_E_INVALIDARG	Parameter <i>pszGuid</i> oder <i>pGuid</i> zeigt auf einen ungültigen Puffer.
VCI_E_FAIL	Die in <i>pszGuid</i> angegebene Zeichenkette konnte nicht in eine gültige Kennzahl umgewandelt werden.

4.2 Die Funktionen der Geräteverwaltung

4.2.1 Funktionen für den Zugriff auf die Geräteliste

4.2.1.1 *vciEnumDeviceOpen*

Die Funktion öffnet die Liste aller beim VCI registrierten Geräte bzw. Interfacekarten. Die vollständige Syntax der Funktion lautet:

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

Parameter:

phEnum

[out] Adresse einer Variable vom Typ HANDLE. Bei erfolgreicher Ausführung liefert die Funktion den Handle der geöffneten Geräteliste in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf NULL gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

4.2.1.2 *vciEnumDeviceClose*

Die Funktion schließt die mittels der Funktion *vciEnumDeviceOpen* geöffnete Geräteliste. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hEnum

[in] Handle der zu schließenden Geräteliste.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *hEnum* angegebene Handle nicht mehr gültig und darf nicht länger verwendet werden.

4.2.1.3 *vciEnumDeviceNext*

Die Funktion ermittelt die Beschreibung eines Gerätes bzw. einer Interfacekarte der Geräteliste und erhöht den internen Listenindex so, dass ein nachfolgender Aufruf der Funktion die Beschreibung zum nächsten Gerät bzw. zur nächsten Interfacekarte liefert. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hEnum

[in] Handle auf die geöffnete Geräteliste.

pInfo

[out] Adresse einer Datenstruktur vom Typ *VCIDEVICEINFO*. Bei erfolgreicher Ausführung speichert die Funktion Informationen über das Gerät bzw. über die Interfacekarte im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert *VCI_OK*, andernfalls einen Fehlercode ungleich *VCI_OK* zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion liefert den Wert *VCI_E_NO_MORE_ITEMS* zurück, wenn die Liste keine weiteren Einträge mehr enthält.

4.2.1.4 *vciEnumDeviceReset*

Die Funktion setzt den internen Listenindex der Geräteliste zurück, so dass ein nachfolgender Aufruf von *vciEnumDeviceNext* wieder den ersten Eintrag der Liste liefert. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hEnum

[in] Handle der geöffneten Geräteliste.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert *VCI_OK*, andernfalls einen Fehlercode ungleich *VCI_OK* zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

4.2.1.5 *vciEnumDeviceWaitEvent*

Die Funktion wartet bis sich der Inhalt der Geräteliste geändert hat, oder eine bestimmte Wartezeit vergangen ist. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hEnum

[in] Handle der geöffneten Geräteliste.

dwMsTimeout

Maximale Wartezeit in Millisekunden. Die Funktion kehrt mit dem Fehlercode **VCI_E_TIMEOUT** zum Aufrufer zurück, wenn sich der Inhalt der Geräteliste innerhalb der angegebenen Zeit nicht geändert hat. Beim Wert **INFINITE** (0xFFFFFFFF) wartet die Funktion so lange, bis eine Änderung der Geräteliste eingetreten ist.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, zurück. Ist die im Parameter *dwMsTimeout* angegebene Zeitspanne verstrichen, ohne dass sich der Inhalt der Geräteliste geändert hat, liefert die Funktion den Fehlercode **VCI_E_TIMEOUT** zurück. Im Fehlerfall liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TIMEOUT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Der Inhalt der Geräteliste ändert sich nur dann, wenn ein Gerät bzw. eine Interfacekarte hinzugefügt oder entfernt wird. Um zu überprüfen, ob sich der Inhalt der Geräteliste geändert hat, ohne das aufrufenden Programm zu blockieren, kann bei Aufruf der Funktion im Parameter *dwMsTimeout* der Wert 0 angegeben werden.

4.2.1.6 *vciFindDeviceByHwid*

Die Funktion sucht nach einem Gerät bzw. einer Interfacekarte mit bestimmter Hardwarekennung. Die vollständige Syntax der Funktion lautet:

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

Parameter:

rHardwareId

[in] Referenz auf die eindeutige Hardwarekennung des gesuchten Gerätes bzw. der gesuchten Interfacekarte.

pVciidDevice

[out] Adresse einer Variable Typ *VCIID*. Bei erfolgreicher Ausführung liefert die Funktion die Gerätekenzahl von der gefundenen Interfacekarte in dieser Variable zurück.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die von dieser Funktion zurück gelieferte Gerätekenzahl kann zum Öffnen des Gerätes mittels der Funktion *vciDeviceOpen* verwendet werden. Jedes Gerät bzw. jede Interfacekarte besitzt eine einmalige und eindeutige Hardwarekennung, die auch nach einem Neustart des System gültig bleibt.

4.2.1.7 vciFindDeviceByClass

Die Funktion sucht nach einem Gerät bzw. nach einer Interfacekarte die einer bestimmten Geräteklasse angehört. Die vollständige Syntax der Funktion lautet:

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

Parameter:

rDeviceClass

[in] Referenz auf die Klassenkennung des gesuchten Gerätes bzw. der gesuchten Interfacekarte.

dwInstNumber

[in] Instanznummer des Gerätes bzw. der Interfacekarte. Sind mehrerer Geräte bzw. Interfacekarten der gleichen Klasse vorhanden, bestimmt dieser Wert die Nummer des gesuchten Gerätes oder der gesuchten Interfacekarte innerhalb der Geräteliste. Der Wert 0 selektiert dabei das erste Gerät bzw. die erste Interfacekarte der angegebenen Geräteklasse.

pVciidDevice

[out] Adresse einer Variable Typ *VCIID*. Bei erfolgreicher Ausführung liefert die Funktion die Gerätekenzahl vom gefundenen Gerät bzw. von der gefundenen Interfacekarte in dieser Variable zurück.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die von dieser Funktion zurück gelieferte Gerätekenzahl kann zum Öffnen des Gerätes mittels der Funktion *vciDeviceOpen* verwendet werden.

4.2.1.8 vciSelectDeviceDlg

Die Funktion zeigt ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte aus der Geräteliste auf dem Bildschirm an. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hwndParent

[in] Handle vom übergeordneten Fenster. Wird hier der Wert NULL angegeben, hat das Dialogfenster kein übergeordnetes Fenster.

pVciidDevice

[out] Adresse einer Variable Typ *VCIID*. Bei erfolgreicher Ausführung liefert die Funktion die Kennzahl vom ausgewählten Gerät bzw. der ausgewählten Interfacekarte in dieser Variable zurück.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Wird das Dialogfenster geschlossen ohne dass ein Gerät oder eine Interfacekarte ausgewählt wurde, liefert die Funktion den Fehlercode **VCI_E_ABORT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die von dieser Funktion zurück gelieferte Gerätekenzahl kann zum Öffnen des Gerätes mittels der Funktion *vciDeviceOpen* verwendet werden.

4.2.2 Funktionen für den Zugriff auf Geräte bzw. Interfacekarten

4.2.2.1 *vciDeviceOpen*

Die Funktion öffnet das Gerät bzw. die Interfacekarte mit der angegebenen Gerätekenzahl. Die vollständige Syntax der Funktion lautet:

HRESULT vciDeviceOpen(REFVCIID rVciidDevice, PHANDLE phDevice)

Parameter:

rVciidDevice

[in] Gerätekenzahl des zu öffnenden Gerätes bzw. Interfacekarte.

phDevice

[out] Adresse einer Variable vom Typ HANDLE. Bei erfolgreicher Ausführung liefert die Funktion den Handle vom geöffneten Gerät bzw. Interfacekarte in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf NULL gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Kennzahl des zu öffnenden Gerätes bzw. der Interfacekarte kann mit einer der Funktionen aus Kapitel 4.2.1 ermittelt werden.

4.2.2.2 *vciDeviceOpenDlg*

Die Funktion zeigt ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte an und öffnet das vom Benutzer ausgewählte Gerät, bzw. die ausgewählte Interfacekarte. Die vollständige Syntax der Funktion lautet:

HRESULT vciDeviceOpenDlg (HWND hwndParent, PHANDLE phDevice);

Parameter:

hwndParent

[in] Handle vom übergeordneten Fenster. Wird hier der Wert NULL angegeben, hat das Dialogfenster kein übergeordnetes Fenster.

phDevice

[out] Adresse einer Variable vom Typ HANDLE. Bei erfolgreicher Ausführung speichert die Funktion den Handle des ausgewählten und geöffneten Gerätes bzw. der Interfacekarte in dieser Variable. Im Falle eines Fehlers wird die Variable auf NULL gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Wird das Dialogfenster geschlossen, ohne dass ein Gerät bzw. eine Interfacekarte ausgewählt wurde, liefert die Funktion den Fehlercode **VCI_E_ABORT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

4.2.2.3 *vciDeviceClose*

Die Funktion schließt ein geöffnetes Gerät bzw. eine geöffnete Interfacekarte. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hDevice

[in] Handle vom zu schließenden Gerät bzw. Interfacekarte. Der hier angegebene Handle muss von einem Aufruf einer der Funktion *vciDeviceOpen* oder *vciDeviceOpenDlg* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *hDevice* angegebene Handle nicht mehr gültig und darf nicht länger verwendet werden.

4.2.2.4 *vciDeviceGetInfo*

Die Funktion ermittelt allgemeine Informationen zu einem Gerät bzw. zu einer Interfacekarte. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hDevice

[in] Handle vom geöffneten Gerät bzw. Interfacekarte.

pInfo

[out] Adresse einer Struktur vom Typ **VCIDEVICEINFO**. Bei erfolgreicher Ausführung speichert die Funktion Informationen zum Gerät bzw. zur Interfacekarte im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion ***vciFormatError***.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur **VCIDEVICEINFO** in Kapitel 5.1.2.

4.2.2.5 vciDeviceGetCaps

Die Funktion ermittelt Informationen über die technische Ausstattung eines Gerätes bzw. einer Interfacekarte. Die vollständige Syntax der Funktion lautet:

<pre>HRESULT vciDeviceGetCaps (HANDLE hDevice, PVCIDEVICECAPS pCaps);</pre>
--

Parameter:

hDevice

[in] Handle vom geöffneten Gerät bzw. Interfacekarte.

pCaps

[out] Adresse einer Struktur vom Typ **VCIDEVICECAPS**. Bei erfolgreicher Ausführung speichert die Funktion die Informationen zur technischen Ausstattung im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion ***vciFormatError***.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur **VCIDEVICECAPS** in Kapitel 5.1.3.

4.3 Die Funktionen für den CAN-Zugriff

Die nachfolgenden Kapitel beschreiben die vom VCI zur Verfügung gestellten Funktionen für den Zugriff auf die CAN-Anschlüsse eines Gerätes bzw. einer Interfacekarte. Einführende Informationen zum CAN-Zugriff befinden sich in Kapitel 3.

4.3.1 Steuereinheit

Die Schnittstelle stellt Funktionen zur Konfiguration und Steuerung eines CAN-Controllers sowie zum Einstellen von Nachrichtenfiltern zur Verfügung. Weitere Informationen zur Steuereinheit finden sich in Kapitel 3.1.2.

4.3.1.1 *canControlOpen*

Die Funktion öffnet die Steuereinheit von einem CAN-Anschluss eines Gerätes bzw. einer Interfacekarte. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hDevice

[in] Handle vom geöffneten Gerät bzw. der Interfacekarte.

dwCanNo

[in] Nummer des CAN-Anschlusses der zu öffnenden Steuereinheit. Der Wert 0 wählt dabei den ersten CAN-Anschluss, der Wert 1 den zweiten CAN-Anschluss, usw. aus.

phCanCtl

[out] Zeiger auf eine Variable vom Typ HANDLE. Bei erfolgreicher Ausführung liefert die Funktion den Handle der geöffneten Steuereinheit in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf NULL gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Wird im Parameter *dwCanNo* der Wert 0xFFFFFFFF angegeben, zeigt die Funktion ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte und eines CAN-Anschlusses auf dem Bildschirm an. In diesem Fall erwartet die Funktion im Parameter *hDevice* nicht das Handle vom Gerät, sondern das Handle eines übergeordneten Fensters oder den Wert NULL, falls kein übergeordnetes Fenster verfügbar ist.

4.3.1.2 *canControlClose*

Die Funktion schließt eine geöffnete Steuereinheit. Die vollständige Syntax der Funktion lautet:

```
HRESULT canControlClose( HANDLE hCanCtl )
```

Parameter:

hCanCtl

[in] Handle der zu schließenden Steuereinheit. Der hier angegebene Handle muss von einem Aufruf der Funktion *canControlOpen* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen zum Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *hCanCtl* angegebene Handle nicht mehr gültig und darf nicht länger verwendet werden.

4.3.1.3 *canControlGetCaps*

Die Funktion ermittelt die Eigenschaften vom CAN-Anschluss der angegebenen Steuereinheit. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

pCanCaps

[out] Zeiger auf eine Struktur vom Typ **CANCAPABILITIES**. Bei erfolgreicher Ausführung speichert die Funktion die Eigenschaften vom CAN-Anschluss im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur **CANCAPABILITIES** in Kapitel 5.2.1.

4.3.1.4 *canControlGetStatus*

Die Funktion ermittelt die aktuellen Einstellungen und den momentanen Zustand vom Controller eines CAN-Anschlusses. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

pStatus

[out] Zeiger auf eine Struktur vom Typ *CANLINESTATUS*. Bei erfolgreicher Ausführung speichert die Funktion die aktuellen Einstellungen und den Zustand vom Controller im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert *VCI_OK*, andernfalls einen Fehlercode ungleich *VCI_OK* zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur *CANLINESTATUS* in Kapitel 5.2.2.

4.3.1.5 *canControlDetectBitrate*

Diese Funktion ermittelt die aktuelle Bitrate vom Bus, mit dem der CAN-Anschluss verbunden ist. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

wMsTimeout

[in] Maximale Wartezeit in Millisekunden zwischen zwei Nachrichten auf dem Bus.

dwCount

[in] Anzahl Elemente in den Bit Timing Tabellen *pabBtr0* bzw. *pabBtr1*.

pabBtr0

[in] Zeiger auf eine Tabelle mit den zu testenden Werten für das Bus Timing Register 0. Der Wert eines Eintrags entspricht dem BT0 Register vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz. Die Tabelle muss mindestens *dwCount* Elemente enthalten.

pabBtr1

[in] Zeiger auf eine Tabelle mit den zu testenden Werten für das Bus Timing Register 1. Der Wert eines Eintrags entspricht dem BT1 Register vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz. Die Tabelle muss mindestens *dwCount* Elemente enthalten.

plIndex

[out] Zeiger auf eine Variable vom Typ INT32. Bei erfolgreicher Ausführung liefert die Funktion den Tabellenindex der gefundenen Bit-Timing-Werte in dieser Variable zurück.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion.

Bemerkungen:

Weitere Informationen zu den Bus Timing Werten in den Tabellen *pabBtr0* und *pabBtr1* finden sich im Datenblatt zum Philips SJA 1000 CAN-Controller.

Zum detektieren der Bitrate wird der CAN-Controller im „Listen-Only“ Modus betrieben. Es ist daher notwendig, dass bei Aufruf der Funktion zwei weitere Busteilnehmer Nachrichten senden. Werden innerhalb der in *wTimeoutMs* angegebenen Zeit keine Nachrichten gesendet, liefert die Funktion den Wert **VCI_E_TIMEOUT** zurück.

Bei erfolgreicher Ausführung der Funktion enthält die Variable auf die der Parameter *plIndex* zeigt den Index (einschließlich 0) der gefundenen Werte innerhalb der Bus Timing Tabellen. Die entsprechenden Tabellenwerte können anschließend zum Initialisieren des CAN-Controllers mittels der Funktion *canControllInitialize* verwendet werden.

Die Funktion kann im undefinierten und gestoppten Zustand aufgerufen werden. Weitere Informationen hierzu finden sich in Kap. 3.1.2.1.

4.3.1.6 *canControlInitialize*

Die Funktion stellt die Betriebsart und Bitrate eines CAN-Anschlusses ein. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

bMode

[in] Betriebsart vom CAN-Controller. Für die Betriebsart kann eine Kombination aus folgenden Konstanten angegeben werden:

CAN_OPMODE_STANDARD:

Akzeptiert CAN-Nachrichten mit 11-Bit Identifiern.

CAN_OPMODE_EXTENDED:

Akzeptiert CAN-Nachrichten mit 29-Bit Identifiern.

CAN_OPMODE_LISTONLY:

CAN-Controller wird im „Listen Only“ Modus betrieben.

CAN_OPMODE_ERRFRAME:

Fehler werden über spezielle CAN-Nachrichten an die Applikation gemeldet.

CAN_OPMODE_LOWSPEED:

CAN-Controller verwendet Low Speed Busankopplung.

bBtr0

[in] Wert für das Bus Timing Register 0 vom CAN-Controller. Der Wert entspricht dem BTR0 Register vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz.

bBtr1

[in] Wert für das Bus Timing Register 1 vom CAN-Controller. Der Wert entspricht dem BTR1 Register vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion.

Bemerkungen:

Die Funktion setzt intern die Controllerhardware entsprechend der Funktion *canControlReset* zurück und initialisiert anschließend den Controller mit den angegebenen Parametern. Die Funktion kann von jedem Controllerzustand aus aufgerufen werden.

Weitere Informationen zu den Bus Timing Werten in den Parametern *bBtr0* und *bBtr1* finden sich im Datenblatt zum Philips SJA 1000 CAN-Controller.

Nachfolgende Tabelle zeigt die Bus Timing Werte für alle von der CiA bzw. von CANopen spezifizierten Bitraten.

Bitrate (Kbit/s)	Vordefinierte Konstanten für 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*

Die Funktion setzt die Controllerhardware und die eingestellten Nachrichtenfilter eines CAN-Anschlusses zurück. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion.

Bemerkungen:

Die Funktion setzt die Controllerhardware zurück, entfernt die eingestellten Akzeptanzfilter, löscht den Inhalt der Filterlisten und schaltet den Controller in den Zustand „nicht initialisiert“. Gleichzeitig wird der Nachrichtenfluss zwischen Controller und den damit verbundenen Nachrichtenkanälen unterbrochen.

Bei einem Aufruf der Funktion wird ein momentan aktiver Sendevorgang des Controllers abgebrochen. Dies kann zu Übertragungsfehlern bzw. zu einem fehlerhaften Nachrichtentelegramm auf dem Bus führen.

Weitere Informationen finden sich in Kapitel 3.1.2.

4.3.1.8 *canControlStart*

Die Funktion startet oder stoppt den Controller eines CAN-Anschlusses. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

fStart

[in] Der Wert TRUE startet, der Wert FALSE stoppt den CAN-Controller.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion.

Bemerkungen:

Ein Aufruf der Funktion ist nur dann erfolgreich, wenn der CAN-Controller zuvor mittels der Funktion *canControlInitialize* konfiguriert wurde.

Nach erfolgreichem Start des CAN-Controllers ist dieser aktiv mit dem Bus verbunden. Eingehende CAN-Nachrichten werden dabei an alle eingerichteten und aktivierten Nachrichtenkanälen weitergeleitet, bzw. Sendenachrichten von den Nachrichtenkanälen an den Bus ausgegeben. Beim Aufruf dieser Funktion wird der Zeitstempel zurückgesetzt. Ein Aufruf der Funktion mit dem Wert FALSE im Parameter *fStart* schaltet den CAN-Controller „offline“. Dabei wird der Nachrichtentransport unterbrochen und der CAN-Controller passiv geschaltet.

Im Gegensatz zur Funktion *canControlReset* werden beim Stoppen die eingestellten Akzeptanzfilter und Filterlisten nicht verändert. Auch bricht die Funktion einen laufenden Sendevorgang des Controllers nicht einfach ab, sondern beendet diesen so, dass dabei kein fehlerhaftes Telegramm auf den Bus übertragen wird.

4.3.1.9 *canControlSetAccFilter*

Die Funktion stellt den 11- oder 29 Bit- Akzeptanzfilter eines CAN-Anschlusses ein. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

fExtended

[in] Auswahl des Akzeptanzfilters. Mit dem Wert FALSE wird der 11-Bit, mit dem Wert TRUE der 29-Bit Akzeptanzfilter ausgewählt.

dwCode

[in] Bitmuster des/der zu akzeptierenden Identifier einschließlich RTR-Bit.

dwMask

[in] Bitmuster der relevanten Bits in *dwCode*. Hat ein Bit in *dwMask* den Wert 0, wird das entsprechende Bit in *dwCode* nicht für den Vergleich herangezogen. Hat es dagegen den Wert 1, ist es beim Vergleich relevant.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Eine ausführliche Beschreibung zur Funktionsweise des Filters und der Werte für die Parameter *dwCode* und *dwMask* befinden sich in Kapitel 3.1.2.2.

Diese Funktion kann nur im „OFFLINE“ Zustand des Controllers erfolgreich ausgeführt werden.

4.3.1.10 *canControlAddFilterIds*

Die Funktion trägt eine oder mehrere Kennziffern (CAN-ID) in die 11- oder 29-Bit Filterliste eines CAN-Anschlusses ein. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

fExtended

[in] Auswahl der Filterliste. Mit dem Wert FALSE wird die 11-Bit, mit dem Wert TRUE die 29-Bit Filterliste ausgewählt.

dwCode

[in] Bitmuster des/der zu registrierenden Identifier einschließlich RTR-Bit.

dwMask

[in] Bitmuster der relevanten Bits in *dwCode*. Hat ein Bit in *dwMask* den Wert 0, wird das entsprechende Bit in *dwCode* nicht berücksichtigt. Hat es dagegen den Wert 1, ist es relevant.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Eine ausführliche Beschreibung zur Funktionsweise des Filters und der Werte für die Parameter *dwCode* und *dwMask* befinden sich in Kapitel 3.1.2.2.

Diese Funktion kann nur im „OFFLINE“ Zustand des Controllers erfolgreich ausgeführt werden.

4.3.1.11 *canControlRemFilterIds*

Die Funktion entfernt eine oder mehrere Kennziffern (CAN-ID) aus der 11- oder 29-Bit Filterliste eines CAN-Anschlusses. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanCtl

[in] Handle der geöffneten CAN-Steuereinheit.

fExtended

[in] Auswahl der Filterliste. Mit dem Wert FALSE wird die 11-Bit, mit dem Wert TRUE die 29-Bit Filterliste ausgewählt.

dwCode

[in] Bitmuster des/der zu entfernenden Identifier einschließlich RTR-Bit.

dwMask

[in] Bitmuster der relevanten Bits in *dwCode*. Hat ein Bit in *dwMask* den Wert 0, wird das entsprechende Bit in *dwCode* nicht berücksichtigt. Hat es dagegen den Wert 1, ist es relevant.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Eine ausführliche Beschreibung zur Funktionsweise des Filters und der Werte für die Parameter *dwCode* und *dwMask* befinden sich in Kapitel 3.1.2.2.

Diese Funktion kann nur im „OFFLINE“ Zustand des Controllers erfolgreich ausgeführt werden.

4.3.2 Nachrichtenkanal

Die Schnittstelle stellt Funktionen zum Einrichten eines Nachrichtenkanals zwischen einer Applikation und einem CAN-Bus zur Verfügung. Ausführliche Informationen zu Nachrichtenkanälen finden sich in Kapitel 3.1.3.

4.3.2.1 *canChannelOpen*

Die Funktion öffnet einen Nachrichtenkanal für einen CAN-Anschluss eines Gerätes bzw. einer Interfacekarte. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hDevice

[in] Handle vom geöffneten Gerät bzw. der Interfacekarte.

dwCanNo

[in] Nummer des CAN-Anschlusses für den ein Nachrichtenkanal geöffnet werden soll. Der Wert 0 wählt dabei den ersten CAN-Anschluss, der Wert 1 den zweiten CAN-Anschluss, usw. aus.

fExclusive

[in] Bestimmt ob der CAN-Anschluss ausschließlich für den zu öffnenden Kanal verwendet wird. Wird hier der Wert TRUE angegeben, wird der CAN-Anschluss exklusiv für den neuen Nachrichtenkanal verwendet. Beim Wert FALSE können mehrere Nachrichtenkanäle für den CAN-Anschluss geöffnet werden.

phCanChn

[out] Zeiger auf eine Variable vom Typ HANDLE. Bei erfolgreicher Ausführung liefert die Funktion den Handle vom geöffneten CAN-Nachrichtenkanal in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf NULL gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Wird im Parameter *fExclusive* der Wert TRUE angegeben, können nach erfolgreichem Aufruf der Funktion keine weiteren Nachrichtenkanäle mehr geöffnet werden. D.h. das Programm, das die Funktion als erstes mit dem Wert TRUE im Parameter *fExclusive* aufruft, besitzt exklusive Kontrolle über den Nachrichtenfluss auf dem CAN-Anschluss.

Wird im Parameter *dwCanNo* der Wert 0xFFFFFFFF angegeben, zeigt die Funktion ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte und eines CAN-Anschlusses auf dem Bildschirm an. In diesem Fall erwartet die Funktion im Parameter *hDevice* nicht das Handle vom Gerät, sondern das Handle eines übergeordneten Fensters oder den Wert NULL, falls kein übergeordnetes Fenster verfügbar ist.

Wird der Nachrichtenkanal nicht mehr benötigt, sollte das in *phCanChn* zurück gelieferte Handle mittels der Funktion *canChannelClose* freigegeben werden.

4.3.2.2 *canChannelClose*

Die Funktion schließt einen geöffneten Nachrichtenkanal. Die vollständige Syntax der Funktion lautet:

```
HRESULT canChannelClose( HANDLE hCanChn )
```

Parameter:

hCanChn

[in] Handle vom zu schließenden Nachrichtenkanal. Der hier angegebene Handle muss von einem Aufruf der Funktion *canChannelOpen* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *hCanChn* angegebene Handle nicht mehr gültig und darf nicht länger verwendet werden.

4.3.2.3 *canChannelGetCaps*

Die Funktion ermittelt die Eigenschaften vom CAN-Anschluss des angegebenen Nachrichtenkanals. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

pCanCaps

[out] Zeiger auf eine Struktur vom Typ **CANCAPABILITIES**. Bei erfolgreicher Ausführung speichert die Funktion die Eigenschaften vom CAN-Anschluss im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur **CANCAPABILITIES** in Kapitel 5.2.1.

4.3.2.4 canChannelGetStatus

Die Funktion ermittelt den aktuellen Zustand eines Nachrichtenkanals, sowie die aktuellen Einstellungen und den momentanen Zustand vom Controller, der mit dem Kanal verbunden ist. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

pStatus

[out] Zeiger auf eine Struktur vom Typ **CANCHANSTATUS**. Bei erfolgreicher Ausführung speichert die Funktion den aktuellen Zustand von Kanal und Controller im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur **CANCHANSTATUS** in Kapitel 5.2.3.

4.3.2.5 *canChannelInitialize*

Die Funktion initialisiert den Empfangs- und Send-Puffer eines Nachrichtenkanals. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

wRxFifoSize

[in] Größe vom Empfangspuffer in Anzahl CAN-Nachrichten.

wRxThreshold

[in] Schwellwert für das Empfangs- Event. Das Event wird ausgelöst, wenn die Anzahl Nachrichten im Empfangspuffer die hier angegebene Anzahl erreicht bzw. überschreitet.

wTxFifoSize

[in] Größe vom Sendepuffer in Anzahl CAN-Nachrichten.

wTxThreshold

[in] Schwellwert für das Sende- Event. Das Event wird ausgelöst, wenn die Anzahl freier Einträge im Sendepuffer die hier angegebene Anzahl erreicht bzw. überschreitet.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Für die Größe vom Empfangs- als auch vom Sendepuffer muss ein Wert größer 0 angegeben werden, andernfalls liefert die Funktion einen Fehlercode entsprechend „Invalid Parameter“ zurück.

Die in den Parametern *wRxFifoSize* und *wTxFifoSize* angegebenen Werte legen die untere Grenze für die Größe der Puffer fest. Die tatsächliche Größe eines Puffers ist unter Umständen größer als der angegebene Wert, da der hierfür verwendete Speicher Seitenweise reserviert wird.

Wird die Funktion für einen bereits initialisierten Kanal aufgerufen, deaktiviert die Funktion zunächst den Kanal, gibt anschließend die vorhandenen FIFOs frei und erzeugt zwei neue FIFOs mit den angeforderten Dimensionen.

4.3.2.6 *canChannelActivate*

Die Funktion aktiviert oder deaktiviert einen Nachrichtenkanal. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

fEnable

Beim Wert TRUE aktiviert die Funktion den Nachrichtenfluss zwischen CAN-Controller und dem Nachrichtenkanal, beim Wert FALSE deaktiviert die Funktion den Nachrichtenfluss.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Standardmäßig ist der Nachrichtenkanal nach dem Öffnen, bzw. Initialisieren deaktiviert. Damit der Kanal Nachrichten vom Bus empfängt, bzw. an diesen sendet, muss dieser aktiviert werden. Gleichzeitig muss sich der CAN-Controller im Zustand „online“ befinden. Weitere Informationen hierzu befinden sich bei der Beschreibung der Funktion *canControlStart* und in Kapitel 3.1.2.

Nach Aktivierung des Kanals können Nachrichten mit *canChannelPostMessage* oder *canChannelSendMessage* in den Sendepuffer geschrieben, bzw. mit den Funktionen *canChannelPeekMessage* und *canChannelReadMessage* aus dem Empfangspuffer gelesen werden.

4.3.2.7 *canChannelPeekMessage*

Die Funktion liest die nächste CAN-Nachricht aus dem Empfangspuffer eines Nachrichtenkanals. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

pCanMsg

[out] Zeiger auf eine Struktur vom Typ *CANMSG*. Bei erfolgreicher Ausführung speichert die Funktion die gelesene CAN-Nachricht im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert *VCI_OK* zurück. Ist bei Aufruf der Funktion keine CAN-Nachricht im Empfangspuffer verfügbar, liefert die Funktion den Wert *VCI_E_RXQUEUE_EMPTY* zurück. Schlägt der Funktionsaufruf aus anderen Gründen fehl, liefert die Funktion einen Fehlercode ungleich *VCI_OK* oder *VCI_E_RXQUEUE_EMPTY* zurück. Weitere Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion kehrt sofort zum aufrufenden Programm zurück, falls keine Nachricht zum Lesen bereitsteht.

Wird im Parameter *pCanMsg* der Wert NULL angegeben, entfernt die Funktion die nächste CAN-Nachricht aus dem Empfangspuffer.

4.3.2.8 *canChannelPostMessage*

Die Funktion schreibt eine CAN-Nachricht in den Sendepuffer des angegebenen Nachrichtenkanals. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

pCanMsg

[in] Zeiger auf eine initialisierte Struktur vom Typ *CANMSG* mit der zu sendenden CAN-Nachricht.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK** zurück. Ist bei Aufruf der Funktion kein Platz im Sendepuffer verfügbar, liefert die Funktion den Wert **VCI_E_TXQUEUE_FULL** zurück. Schlägt der Funktionsaufruf aus anderen Gründen fehl, liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TXQUEUE_FULL** zurück. Weitere Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion wartet nicht, bis die Nachricht auf dem Bus übertragen wurde.

4.3.2.9 *canChannelWaitRxEvent*

Die Funktion wartet bis das Empfangs-Event eingetroffen ist, oder eine bestimmte Wartezeit vergangen ist. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

dwMsTimeout

Maximale Wartezeit in Millisekunden. Die Funktion kehrt mit dem Fehlercode **VCI_E_TIMEOUT** zum Aufrufer zurück, wenn das Empfangs-Event innerhalb der hier angegebenen Zeit nicht eingetroffen ist. Beim Wert **INFINITE** (0xFFFFFFFF) wartet die Funktion so lange, bis das Empfangs-Event eingetreten ist.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, zurück. Ist die im Parameter *dwMsTimeout* angegebene Zeitspanne verstrichen, ohne dass das Empfangs-Event eingetroffen ist, liefert die Funktion den Fehlercode **VCI_E_TIMEOUT** zurück. Im Fehlerfall liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TIMEOUT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Das Empfangs-Event wird ausgelöst, sobald die Anzahl Nachrichten im Empfangspuffer die eingestellte Schwelle erreicht bzw. überschreitet. Siehe hierzu die Beschreibung der Funktion *canChannelInitialize*.

Um zu überprüfen, ob das Empfangs-Event bereits eingetroffen ist, ohne das aufrufenden Programm zu blockieren, kann bei Aufruf der Funktion im Parameter *dwMsTimeout* der Wert 0 angegeben werden.

Falls der in *hCanChn* angegebene Handle von einem anderen Thread aus geschlossen wird, beendet die Funktion den momentanen Funktionsaufruf und kehrt mit einem Rückgabewert ungleich **VCI_OK** zurück.

4.3.2.10 *canChannelWaitTxEvent*

Die Funktion wartet bis das Sende-Event eingetroffen ist, oder eine bestimmte Wartezeit vergangen ist. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

dwMsTimeout

Maximale Wartezeit in Millisekunden. Die Funktion kehrt mit dem Fehlercode **VCI_E_TIMEOUT** zum Aufrufer zurück, wenn das Sende-Event innerhalb der hier angegebene Zeit nicht eingetroffen ist. Beim Wert INFINITE (0xFFFFFFFF) wartet die Funktion so lange, bis das Sende-Event eingetreten ist.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, zurück. Ist die im Parameter *dwMsTimeout* angegebene Zeitspanne verstrichen, ohne dass das Sende-Event eingetroffen ist, liefert die Funktion den Fehlercode **VCI_E_TIMEOUT** zurück. Im Fehlerfall liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TIMEOUT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Das Sende-Event wird ausgelöst, sobald der Sendepuffer gleich viele oder mehr freie Einträge enthält als die eingestellte Schwelle. Siehe hierzu die Beschreibung der Funktion *canChannelInitialize*.

Um zu überprüfen, ob das Sende-Event bereits eingetroffen ist, ohne das aufrufenden Programm zu blockieren, kann bei Aufruf der Funktion im Parameter *dwMsTimeout* der Wert 0 angegeben werden.

Falls der in *hCanChn* angegebene Handle von einem anderen Thread aus geschlossen wird, beendet die Funktion den momentanen Funktionsaufruf und kehrt mit einem Rückgabewert ungleich **VCI_OK** zurück.

4.3.2.11 *canChannelReadMessage*

Die Funktion liest die nächste CAN-Nachricht aus dem Empfangspuffer eines Nachrichtenkanals. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

dwMsTimeout

Maximale Wartezeit in Millisekunden. Die Funktion kehrt mit dem Fehlercode **VCI_E_TIMEOUT** zum Aufrufer zurück, wenn innerhalb der angegebene Zeit keine Nachricht gelesen bzw. empfangen wurde. Beim Wert INFINITE (0xFFFFFFFF) wartet die Funktion so lange, bis eine Nachricht gelesen wurde.

pCanMsg

[out] Zeiger auf eine Struktur vom Typ **CANMSG**. Bei erfolgreicher Ausführung speichert die Funktion die gelesene CAN-Nachricht im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, zurück. Ist die im Parameter *dwMsTimeout* angegebene Zeitspanne verstrichen, ohne dass eine Nachricht empfangen wurde, liefert die Funktion den Fehlercode **VCI_E_TIMEOUT** zurück. Im Fehlerfall liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TIMEOUT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Wird im Parameter *pCanMsg* der Wert NULL angegeben, entfernt die Funktion die nächste CAN-Nachricht aus dem Empfangspuffer.

Falls der in *hCanChn* angegebene Handle von einem anderen Thread aus geschlossen wird, beendet die Funktion den momentanen Funktionsaufruf und kehrt mit einem Rückgabewert ungleich **VCI_OK** zurück.

4.3.2.12 *canChannelSendMessage*

Die Funktion wartet bis ein Nachrichtenkanal zur Aufnahme einer Nachricht bereit ist und schreibt anschließend die angegebene CAN-Nachricht in den Sendepuffer des Nachrichtenkanals. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanChn

[in] Handle vom geöffneten CAN-Nachrichtenkanal.

dwMsTimeout

Maximale Wartezeit in Millisekunden. Die Funktion kehrt mit dem Fehlercode **VCI_E_TIMEOUT** zum Aufrufer zurück, falls die Nachricht nicht innerhalb der angegebenen Zeit in den Sendepuffer eingetragen werden konnte. Beim Wert INFINITE (0xFFFFFFFF), wartet die Funktion bis das Sende-Event eintrifft und die Nachricht in den Sendepuffer geschrieben wurde.

pCanMsg

[in] Zeiger auf eine initialisierte Struktur vom Typ **CANMSG** mit der zu sendenden CAN-Nachricht.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, zurück. Ist die im Parameter *dwMsTimeout* angegebene Zeitspanne verstrichen, ohne dass die Nachricht in den Sendepuffer geschrieben wurde, liefert die Funktion den Fehlercode **VCI_E_TIMEOUT** zurück. Im Fehlerfall liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TIMEOUT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion wartet nur solange bis die Nachricht in den Sendepuffer geschrieben wurde, jedoch nicht, bis die Nachricht auf dem Bus übertragen wurde.

Falls der in *hCanChn* angegebene Handle von einem anderen Thread aus geschlossen wird, beendet die Funktion den momentanen Funktionsaufruf und kehrt mit einem Rückgabewert ungleich **VCI_OK** zurück.

4.3.3 Zyklische Sendeliste

Die Schnittstelle stellt Funktionen zum Einrichten einer optional vorhandenen zyklischen Sendeliste zur Verfügung. Ausführliche Informationen zu zyklischen Sendelisten findet sich in Kapitel 3.1.4.

4.3.3.1 *canSchedulerOpen*

Die Funktion öffnet die zyklische Sendeliste eines CAN-Anschlusses eines Gerätes bzw. einer Interfacekarte. Die vollständige Syntax der Funktion lautet:

<pre>HRESULT canSchedulerOpen(HANDLE hDevice, UINT32 dwCanNo, PHANDLE phCanShd)</pre>

Parameter:

hDevice

[in] Handle vom geöffneten Gerät bzw. der Interfacekarte.

dwCanNo

[in] Nummer des CAN-Anschlusses der zu öffnenden Sendeliste. Der Wert 0 wählt dabei den ersten CAN-Anschluss, der Wert 1 den zweiten CAN-Anschluss, usw. aus.

phCanShd

[out] Zeiger auf eine Variable vom Typ HANDLE. Bei erfolgreicher Ausführung liefert die Funktion den Handle der geöffneten zyklischen Sendeliste in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf NULL gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Wird im Parameter *dwCanNo* der Wert 0xFFFFFFFF angegeben, zeigt die Funktion ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte und eines CAN-Anschlusses auf dem Bildschirm an. In diesem Fall erwartet die Funktion im Parameter *hDevice* nicht das Handle vom Gerät, sondern das Handle eines übergeordneten Fensters oder den Wert NULL, falls kein übergeordnetes Fenster verfügbar ist.

4.3.3.2 *canSchedulerClose*

Die Funktion schließt eine geöffnete zyklische Sendeliste. Die vollständige Syntax der Funktion lautet:

```
HRESULT canSchedulerClose( HANDLE hCanShd )
```

Parameter:

hCanShd

[in] Handle der zu schließenden zyklischen Sendeliste. Der hier angegebene Handle muss von einem Aufruf der Funktion *canSchedulerOpen* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *hCanShd* angegebene Handle nicht mehr gültig und darf nicht länger verwendet werden.

4.3.3.3 *canSchedulerGetCaps*

Die Funktion ermittelt die Eigenschaften vom CAN-Anschluss der angegebenen zyklischen Sendeliste. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

pCanCaps

[out] Zeiger auf eine Struktur vom Typ **CANCAPABILITIES**. Bei erfolgreicher Ausführung speichert die Funktion die Eigenschaften vom CAN-Anschluss im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur **CANCAPABILITIES** in Kapitel 5.2.1.

4.3.3.4 *canSchedulerGetStatus*

Die Funktion ermittelt den momentanen Zustand der Sendetask und aller registrierten Sendeobjekte einer zyklischen Sendeliste. Die Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

pStatus

[out] Zeiger auf eine Struktur vom Typ *CANSCHEDULERSTATUS*. Bei erfolgreicher Ausführung speichert die Funktion den aktuellen Zustand aller zyklischen Sendeobjekte im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert *VCI_OK*, andernfalls einen Fehlercode ungleich *VCI_OK* zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion liefert in der Tabelle *CANSCHEDULERSTATUS.abMsgStat* den momentanen Zustand aller 16 Sendeobjekte zurück. Der von der Funktion *canSchedulerAddMessage* gelieferte Listenindex kann verwendet werden, um den Zustand eines einzelnen Sendeobjekts abzufragen. D.h. *abMsgStat[Index]* enthält den Zustand des Sendeobjekts mit dem angegebenen Index.

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur *CANSCHEDULERSTATUS* in Kapitel 5.2.4.

4.3.3.5 *canSchedulerActivate*

Die Funktion startet oder stoppt die Sendetask der zyklischen Sendeliste und damit den zyklischen Sendevorgang aller momentan registrierten Sendeobjekte. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

fEnable

Beim Wert TRUE aktiviert, beim Wert FALSE deaktiviert die Funktion den zyklischen Sendevorgang aller momentan registrierten Sendeobjekte.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion kann zum gleichzeitigen Start aller registrierten Sendeobjekte verwendet werden. Hierzu werden zunächst alle Sendeobjekte mittels der Funktion *canSchedulerStartMessage* in den gestarteten Zustand versetzt. Ein anschließender Aufruf dieser Funktion mit dem Wert TRUE für den Parameter *fEnable* garantiert dann einen zeitgleichen Start.

Wird die Funktion mit dem Wert FALSE für den Parameter *fEnable* aufgerufen, wird die Bearbeitung aller registrierten Sendeobjekte gleichzeitig gestoppt.

4.3.3.6 canSchedulerReset

Die Funktion stoppt die Sendetask und entfernt alle Sendeobjekte aus der angegebenen zyklischen Sendeliste. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

4.3.3.7 canSchedulerAddMessage

Die Funktion fügt ein neues Sendeobjekt zur angegebenen zyklischen Sendeliste hinzu. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

pMessage

[in] Zeiger auf eine initialisierte Struktur vom Typ *CANCYCLICTXMSG* mit dem Sendeobjekt.

pdwIndex

[out] Zeiger auf eine Variable vom Typ *UINT32*. Bei erfolgreicher Ausführung liefert die Funktion den Listenindex des neu hinzugefügten Sendeobjekts in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf den Wert *0xFFFFFFFF* (-1) gesetzt. Dieser Index wird für alle weiteren Funktionsaufrufe benötigt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert *VCI_OK*, andernfalls einen Fehlercode ungleich *VCI_OK* zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Der zyklische Sendevorgang des neu hinzugefügten Sendeobjekts beginnt erst nach erfolgreichem Aufruf der Funktion *canSchedulerStartMessage*. Zusätzlich muss die Sendeliste aktiv sein (siehe *canSchedulerActivate*).

4.3.3.8 *canSchedulerRemMessage*

Die Funktion stoppt die Bearbeitung eines Sendeobjekts und entfernt dieses aus der angegebenen zyklischen Sendeliste. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

dwIndex

[in] Listenindex des zu entfernenden Sendeobjekts. Der Listenindex muss von einem früheren Aufruf der Funktion *canSchedulerAddMessage* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert *VCI_OK*, andernfalls einen Fehlercode ungleich *VCI_OK* zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *dwIndex* angegebene Listenindex ungültig und darf nicht weiter verwendet werden.

4.3.3.9 *canSchedulerStartMessage*

Die Funktion startet ein Sendeobjekt der angegebenen zyklischen Sendeliste. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

dwIndex

[in] Listenindex des zu startenden Sendeobjekts. Der Listenindex muss von einem früheren Aufruf der Funktion *canSchedulerAddMessage* stammen.

dwCount

[in] Anzahl der zyklischen Sendewiederholungen. Beim Wert 0 wird der Sendevorgang endlos oft wiederholt. Der hier angegebene Wert muss im Bereich 0 bis 65535 liegen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Der zyklische Sendevorgang startet nur dann, wenn die Sendetask bei Aufruf der Funktion aktiv ist. Ist die Sendetask inaktiv, wird der Sendevorgang bis zum nächsten Aufruf der Funktion *canSchedulerActivate* verzögert.

4.3.3.10 *canSchedulerStopMessage*

Die Funktion stoppt ein Sendeobjekt der angegebenen zyklischen Sendeliste. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hCanShd

[in] Handle der geöffneten zyklischen Sendeliste.

dwIndex

[in] Listenindex des zu stoppenden Sendeobjekts. Der Listenindex muss von einem früheren Aufruf der Funktion *canSchedulerAddMessage* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion.

4.4 Die Funktionen für den LIN-Zugriff

Die nachfolgenden Kapitel beschreiben die vom VCI zur Verfügung gestellten Funktionen für den Zugriff auf die LIN-Anschlüsse eines Gerätes bzw. einer Interfacekarte. Einführende Informationen zum LIN-Zugriff befinden sich in Kapitel 3.2.

4.4.1 Steuereinheit

Die Schnittstelle stellt Funktionen zur Konfiguration und Steuerung eines LIN-Controllers sowie zum Abfragen der Eigenschaften eines LIN-Anschlusses zur Verfügung. Weitere Informationen zur Steuereinheit finden sich in Kapitel 3.2.2.

4.4.1.1 *linControlOpen*

Die Funktion öffnet die Steuereinheit von einem LIN-Anschluss eines Gerätes bzw. einer Interfacekarte. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hDevice

[in] Handle vom geöffneten Gerät bzw. der Interfacekarte.

dwLinNo

[in] Nummer des LIN-Anschlusses der zu öffnenden Steuereinheit. Der Wert 0 wählt dabei den ersten LIN-Anschluss, der Wert 1 den zweiten LIN-Anschluss, usw. aus.

phLinCtl

[out] Zeiger auf eine Variable vom Typ HANDLE. Bei erfolgreicher Ausführung liefert die Funktion den Handle der geöffneten Steuereinheit in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf NULL gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Wird im Parameter *dwLinNo* der Wert 0xFFFFFFFF angegeben, zeigt die Funktion ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte und eines LIN-Anschlusses auf dem Bildschirm an. In diesem Fall erwartet die Funktion im Parameter *hDevice* nicht das Handle vom Gerät, sondern das Handle eines übergeordneten Fensters oder den Wert NULL, falls kein übergeordnetes Fenster verfügbar ist.

4.4.1.2 *linControlClose*

Die Funktion schließt einen geöffneten LIN-Controller. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinCtl

[in] Handle der zu schließenden LIN-Steuereinheit. Der hier angegebene Handle muss von einem Aufruf der Funktion *linControlOpen* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *hLinCtl* angegebene Handle nicht mehr gültig und darf nicht länger verwendet werden.

4.4.1.3 *linControlGetCaps*

Die Funktion ermittelt die Eigenschaften vom LIN-Anschluss der angegebenen Steuereinheit. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinCtl

[in] Handle der geöffneten LIN-Steuereinheit.

pLinCaps

[out] Zeiger auf eine Struktur vom Typ *LINCAPABILITIES*. Bei erfolgreicher Ausführung speichert die Funktion die Eigenschaften vom LIN-Anschluss im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur *LINCAPABILITIES* in Kapitel 5.3.1.

4.4.1.4 linControlGetStatus

Die Funktion ermittelt die aktuellen Einstellungen und den momentanen Zustand vom Controller eines LIN-Anschlusses. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinCtl

[in] Handle der geöffneten LIN-Steuereinheit.

pStatus

[out] Zeiger auf eine Struktur vom Typ *LINLINESTATUS*. Bei erfolgreicher Ausführung speichert die Funktion die aktuellen Einstellungen und den Zustand vom Controller im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur *LINLINESTATUS* in Kapitel 5.3.2.

4.4.1.5 linControlInitialize

Die Funktion stellt die Betriebsart und Bitrate eines LIN-Anschlusses ein. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinCtl

[in] Handle der geöffneten LIN-Steuereinheit.

bMode

[in] Betriebsart vom LIN-Controller. Für die Betriebsart kann eine Kombination aus folgenden Konstanten angegeben werden:

LIN_OPMODE_SLAVE:

Slave Betrieb. Diese Betriebsart ist standardmäßig immer aktiv.

LIN_OPMODE_MASTER:

Master Betrieb aktivieren (falls unterstützt, siehe auch *LINCAPABILITIES*).

LIN_OPMODE_ERRORS:

Fehler werden über spezielle LIN-Nachrichten an die Applikation gemeldet.

wBtrate

[in] Bitrate in Bit pro Sekunde. Der hier angegebenen Wert muss innerhalb der durch die Konstanten **LIN_BITRATE_MIN** und **LIN_BITRATE_MAX** definierten Grenzen liegen. Wird der Controller als Slave betrieben, kann die Bitrate durch Angabe des Wertes **LIN_BITRATE_AUTO** vom Controller automatisch ermittelt werden falls dies unterstützt wird.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion setzt intern die Controllerhardware entsprechend der Funktion *linControlReset* zurück und initialisiert anschließend den Controller mit den angegebenen Parametern. Die Funktion kann von jedem Controllerzustand aus aufgerufen werden.

4.4.1.6 linControlReset

Die Funktion setzt die Controllerhardware eines LIN-Anschlusses zurück. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinCtl

[in] Handle der geöffneten LIN-Steuereinheit.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion setzt die Controllerhardware zurück und schaltet den Controller „offline“. Dabei wird auch der Nachrichtentransport zwischen Controller und den momentan geöffneten Nachrichtenmonitoren unterbrochen.

Bei einem Aufruf der Funktion wird ein momentan aktiver Sendevorgang des Controllers abgebrochen. Dies kann zu Übertragungsfehlern bzw. zu einem fehlerhaften Nachrichtentelegramm auf dem Bus führen.

Weitere Informationen finden sich in Kapitel 3.2.2.

4.4.1.7 *linControlStart*

Die Funktion startet oder stoppt den Controller des LIN-Anschlusses. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinCtl

[in] Handle der geöffneten LIN-Steuereinheit.

fStart

[in] Der Wert TRUE startet, der Wert FALSE stoppt den LIN-Controller.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Ein Aufruf der Funktion ist nur dann erfolgreich, wenn der LIN-Controller zuvor mittels der Funktion *linControlInitialize* konfiguriert wurde.

Nach erfolgreichem Aufruf der Funktion ist der LIN-Controller aktiv mit dem Bus verbunden („online“). Eingehende LIN-Nachrichten werden dabei an alle aktiven Nachrichtenmonitore weitergeleitet, bzw. Sendenachrichten vom Controller auf dem Bus übertragen. Ein Aufruf der Funktion mit dem Wert FALSE im Parameter *fStart* schaltet den LIN-Controller „offline“. Dabei wird der Nachrichtentransport unterbrochen und der LIN-Controller passiv geschaltet.

Im Gegensatz zur Funktion *linControlReset* bricht die Funktion einen laufenden Sendevorgang des Controllers nicht einfach ab, sondern beendet diesen so, dass dabei kein fehlerhaftes Telegramm auf den Bus übertragen wird.

4.4.1.8 *linControlWriteMessage*

Diese Funktion sendet die angegebene Nachricht entweder direkt an den mit dem Controller verbundenen LIN-Bus, oder trägt die Nachricht in die Antworttabelle vom Controller ein. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinCtl

[in] Handle der geöffneten LIN-Steuereinheit.

fSend

[in] Bestimmt, ob die Nachricht direkt auf den LIN-Bus übertragen wird, oder ob sie in die Antworttabelle vom Controller eingetragen wird. Mit TRUE wird die Nachricht direkt gesendet, mit FALSE wird die Nachricht in die Antworttabelle eingetragen.

pLinMsg

[in] Zeiger auf eine initialisierte Struktur vom Typ *LINMSG* mit der zu sendenden LIN-Nachricht.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Ausführliche Informationen zu dieser Funktion finden sich in Kapitel 3.2.2.2

4.4.2 Nachrichtenmonitor

Die Schnittstelle stellt Funktionen zum Einrichten eines Nachrichtenmonitors zwischen einer Applikation und einem LIN-Bus zur Verfügung. Ausführliche Informationen zu Nachrichtenmonitoren finden sich in Kapitel 3.2.3.

4.4.2.1 *linMonitorOpen*

Die Funktion erstellt einen Nachrichtenmonitor für den LIN-Anschluss eines Gerätes bzw. einer Interfacekarte. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hDevice

[in] Handle vom geöffneten Gerät bzw. der Interfacekarte.

dwLinNo

[in] Nummer des LIN-Anschlusses der zu öffnenden Steuereinheit. Der Wert 0 wählt dabei den ersten LIN-Anschluss, der Wert 1 den zweiten LIN-Anschluss, usw. aus.

fExclusive

[in] Bestimmt ob der LIN-Anschluss ausschließlich für den neu zu erzeugenden Monitor verwendet wird. Wird hier der Wert TRUE angegeben, können nach erfolgreichem Aufruf der Funktion keine weiteren Monitore mehr erstellt werden, bis der hiermit erzeugte Monitor wieder freigegeben wurde. Beim Wert FALSE können beliebig viele Nachrichtenmonitore für den LIN-Anschluss erstellt werden.

phLinMon

[out] Zeiger auf eine Variable vom Typ HANDLE. Bei erfolgreicher Ausführung liefert die Funktion den Handle vom geöffneten Monitor in dieser Variable zurück. Im Falle eines Fehlers wird die Variable auf den Wert *NULL* gesetzt.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciformatError*.

Bemerkungen:

Wird im Parameter *fExclusive* der Wert TRUE angegeben, können nach erfolgreichem Aufruf der Funktion keine weiteren Nachrichtenmonitore mehr geöffnet werden. D.h. das Programm, das die Funktion als erstes mit dem Wert TRUE im Parameter *fExclusive* aufruft, besitzt exklusive Kontrolle über den Nachrichtenempfang vom LIN-Anschluss.

Wird im Parameter *dwLinNo* der Wert 0xFFFFFFFF angegeben, zeigt die Funktion ein Dialogfenster zur Auswahl eines Gerätes bzw. einer Interfacekarte und eines LIN-Anschlusses auf dem Bildschirm an. In diesem Fall erwartet die Funktion im Parameter *hDevice* nicht das Handle vom Gerät, sondern das Handle eines übergeordneten Fensters oder den Wert NULL, falls kein übergeordnetes Fenster verfügbar ist.

Wird der Nachrichtenmonitor nicht mehr benötigt, sollte das in *phLinMon* zurück gelieferte Handle mittels Funktion *linMonitorClose* wieder freigegeben werden.

4.4.2.2 *linMonitorClose*

Die Funktion schließt einen geöffneten Nachrichtenmonitor für den LIN-Anschluss. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom zu schließenden Nachrichtenmonitor. Der hier angegebene Handle muss von einem Aufruf der Funktion *linMonitorOpen* stammen.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Nach Aufruf der Funktion ist der in *hLinMon* angegebene Handle nicht mehr gültig und darf nicht länger verwendet werden.

4.4.2.3 *linMonitorGetCaps*

Die Funktion ermittelt die Eigenschaften vom LIN-Anschluss des angegebenen Nachrichtenmonitors. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom geöffneten LIN-Nachrichtenmonitor.

pLinCaps

[out] Zeiger auf eine Struktur vom Typ **LINCAPABILITIES**. Bei erfolgreicher Ausführung speichert die Funktion die Eigenschaften vom LIN-Anschluss im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur *LINCAPABILITIES* in Kapitel 5.3.1.

4.4.2.4 linMonitorGetStatus

Die Funktion ermittelt den aktuellen Zustand eines Nachrichtenmonitors, sowie die aktuellen Einstellungen und den momentanen Zustand vom Controller, der mit dem Nachrichtenmonitor verbunden ist. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom geöffneten LIN-Nachrichtenmonitor.

pStatus

[out] Zeiger auf eine Struktur vom Typ *LINMONITORSTATUS*. Bei erfolgreicher Ausführung speichert die Funktion den aktuellen Zustand von Monitor und Controller im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Weitere Informationen über die von der Funktion gelieferten Daten finden sich bei der Beschreibung der Datenstruktur *LINMONITORSTATUS* in Kapitel 5.3.3.

4.4.2.5 *linMonitorInitialize*

Die Funktion initialisiert den Empfangspuffer eines Nachrichtenmonitors. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom geöffneten LIN-Nachrichtenmonitor.

wFifoSize

[in] Grösse vom Empfangspuffer in Anzahl LIN-Nachrichten.

wThreshold

[in] Schwellwert für das Empfangs-Event. Das Event wird ausgelöst, wenn die Anzahl Nachrichten im Empfangspuffer die hier angegebene Anzahl erreicht bzw. überschreitet.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Für die Größe vom Empfangspuffer muss ein Wert größer 0 angegeben werden, andernfalls liefert die Funktion einen Fehlercode entsprechend „Invalid Parameter“ zurück.

Der im Parameter *wFifoSize* angegebene Wert legt die untere Grenze für die Größe der Puffer fest. Die tatsächliche Größe des Puffers ist unter Umständen größer als der angegebene Wert, da der hierfür verwendete Speicher Seitenweise reserviert wird.

Wird die Funktion für einen bereits initialisierten Monitor aufgerufen, deaktiviert die Funktion zunächst den Monitor, gibt anschließend den vorhandenen Puffer frei und erzeugt einen neuen Puffer der angeforderten Größe.

4.4.2.6 *linMonitorActivate*

Die Funktion aktiviert oder deaktiviert einen Nachrichtenmonitor. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom geöffneten LIN-Nachrichtenmonitor.

fEnable

[in] Beim Wert TRUE aktiviert die Funktion den Nachrichtenempfang zwischen LIN-Controller und dem Nachrichtenmonitor, beim Wert FALSE deaktiviert die Funktion den Nachrichtenfluss.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, andernfalls einen Fehlercode ungleich **VCI_OK** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Standardmäßig ist der Nachrichtenmonitor nach dem Öffnen, bzw. Initialisieren deaktiviert. Damit der Monitor Nachrichten vom Bus empfängt, muss dieser aktiviert werden. Gleichzeitig muss sich der LIN-Controller im Zustand „online“ befinden. Weitere Informationen hierzu befinden sich in Kapiteln 3.2.2 und 3.2.3.

Nach erfolgreicher Aktivierung des Monitors können Nachrichten mittels den Funktionen *linMonitorPeekMessage* und *linMonitorReadMessage* aus dem Empfangspuffer gelesen werden.

4.4.2.7 *linMonitorPeekMessage*

Die Funktion liest die nächste LIN-Nachricht aus dem Empfangspuffer eines Monitors. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom geöffneten LIN-Nachrichtenmonitor.

pLinMsg

[out] Zeiger auf eine Struktur vom Typ *LINMSG*. Bei erfolgreicher Ausführung speichert die Funktion die gelesene LIN-Nachrichten im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK** zurück. Ist bei Aufruf der Funktion keine LIN-Nachricht im Empfangspuffer verfügbar, liefert die Funktion den Wert **VCI_E_RXQUEUE_EMPTY** zurück. Schlägt der Funktionsaufruf aus anderen Gründen fehl, liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_RXQUEUE_EMPTY** zurück. Weitere Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Die Funktion kehrt sofort zum aufrufenden Programm zurück, falls keine Nachricht zum Lesen bereitsteht. Wird im Parameter *pLinMsg* der Wert NULL angegeben, entfernt die Funktion die nächste LIN-Nachricht aus dem Empfangspuffer.

4.4.2.8 *linMonitorWaitRxEvent*

Die Funktion wartet bis das Empfangs- Event eingetroffen ist, oder eine bestimmte Wartezeit vergangen ist. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom geöffneten LIN-Nachrichtenmonitor.

dwMsTimeout

[in] Maximale Wartezeit in Millisekunden. Die Funktion kehrt mit dem Fehlercode **VCI_E_TIMEOUT** zum Aufrufer zurück, wenn das Empfangs-Event innerhalb der hier angegebenen Zeit nicht eingetroffen ist. Beim Wert INFINITE (0xFFFFFFFF) wartet die Funktion so lange, bis das Empfangs-Event eingetreten ist.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, zurück. Ist die im Parameter *dwMsTimeout* angegebene Zeitspanne verstrichen, ohne dass das Empfangs-Event eingetroffen ist, liefert die Funktion den Fehlercode **VCI_E_TIMEOUT** zurück. Im Fehlerfall liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TIMEOUT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Das Empfangs- Event wird ausgelöst, sobald die Anzahl Nachrichten im Empfangspuffer die eingestellte Schwelle erreicht. Siehe hierzu die Beschreibung der Funktion *linMonitorInitialize*.

Um zu überprüfen, ob das Empfangs- Event bereits eingetroffen ist, ohne das aufrufenden Programm zu blockieren, kann bei Aufruf der Funktion im Parameter *dwMsTimeout* der Wert 0 angegeben werden.

Falls der in *hLinMon* angegebene Handle von einem anderen Thread aus geschlossen wird, beendet die Funktion den momentanen Funktionsaufruf und kehrt mit einem Rückgabewert ungleich **VCI_OK** zurück.

4.4.2.9 linMonitorReadMessage

Die Funktion liest die nächste LIN-Nachricht aus dem Empfangspuffer eines Monitors. Die vollständige Syntax der Funktion lautet:

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

Parameter:

hLinMon

[in] Handle vom geöffneten LIN-Nachrichtenmonitor.

dwMsTimeout

[in] Maximale Wartezeit in Millisekunden. Die Funktion kehrt mit dem Fehlercode **VCI_E_TIMEOUT** zum Aufrufer zurück, wenn innerhalb der angegebene Zeit keine Nachricht gelesen bzw. empfangen wurde. Beim Wert INFINITE (0xFFFFFFFF) wartet die Funktion so lange, bis eine Nachricht gelesen wurde.

pLinMsg

[out] Zeiger auf eine Struktur vom Typ *LINMSG*. Bei erfolgreicher Ausführung speichert die Funktion die gelesene LIN-Nachricht im hier angegebenen Speicherbereich.

Rückgabewert:

Bei erfolgreicher Ausführung liefert die Funktion den Wert **VCI_OK**, zurück. Ist die im Parameter *dwMsTimeout* angegebene Zeitspanne verstrichen, ohne dass eine Nachricht empfangen wurde, liefert die Funktion den Fehlercode **VCI_E_TIMEOUT** zurück. Im Fehlerfall liefert die Funktion einen Fehlercode ungleich **VCI_OK** oder **VCI_E_TIMEOUT** zurück. Weitergehende Informationen über den Fehlercode liefert die Funktion *vciFormatError*.

Bemerkungen:

Wird im Parameter *pLinMsg* der Wert NULL angegeben, entfernt die Funktion die nächste LIN-Nachricht aus dem Empfangspuffer.

Falls der in *hLinMon* angegebene Handle von einem anderen Thread aus geschlossen wird, beendet die Funktion den momentanen Funktionsaufruf und kehrt mit einem Rückgabewert ungleich **VCI_OK** zurück.

5 Typen und Strukturen

5.1 VCI spezifische Datentypen

Die Deklarationen aller VCI spezifischen Datentypen und Konstanten befinden sich in der Datei `<vcitype.h>`.

5.1.1 VCIID

Der Datentyp beschreibt eine VCI spezifische, lokal eindeutige Kennzahl (VCI Locally Unique ID). Der Datentyp hat folgenden Aufbau:

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

- *AsLuid*:
Kennzahl in Form einer LUID. Der Datentyp LUID ist in Windows definiert.
- *AsInt64*:
Kennzahl als vorzeichenbehafteter 64-Bit Integer.

5.1.2 VCIDEVICEINFO

Die Struktur beschreibt die Informationen zu einem Gerät bzw. einer Interfacekarte. Die Struktur hat folgenden Aufbau:

```
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;
```


- *VciObjectId*:
[out] Eindeutige Kennzahl (*VCIID*) des Gerätes bzw. der Interfacekarte. Der VCI System Service weist jedem Gerät eine systemweit eindeutige und einmalige Kennzahl zu. Diese Kennzahl dient als Schlüssel für spätere Zugriffe auf das Gerät.
- *DeviceClass*:
[out] ID der Geräteklasse. Jeder Gerätetreiber kennzeichnet seine Geräteklasse in Form einer Globally Unique ID (GUID). Unterschiedliche Geräte bzw. Interfacekarten gehören unterschiedlichen Geräteklassen an. Applikationen können die Geräteklasse verwenden um z.B. zwischen einer IPC-I165/PCI und einer PC-I04/PCI zu unterscheiden.
- *DriverMajorVersion*:
[out] Hauptversionsnummer des Gerätetreibers.
- *DriverMinorVersion*:
[out] Nebenversionsnummer des Gerätetreibers.
- *DriverBuildVersion*:
[out] Buildversionsnummer des Gerätetreibers.
- *DriveReleaseVersion*:
[out] Release nummer (bzw. Revision nummer) des Gerätetreibers.
- *HardwareBranchVersion*:
[out] Branchversionsnummer der Hardware.
- *HardwareMajorVersion*:
[out] Hauptversionsnummer der Hardware.
- *HardwareMinorVersion*:
[out] Nebenversionsnummer der Hardware.
- *HardwareBuildVersion*:
[out] Buildversionsnummer der Hardware.
- *UniqueHardwareId*:
[out] Eindeutige Kennung des Gerätes bzw. der Interfacekarte. Jedes Gerät hat eine eindeutige Kennung die z.B. zur Unterscheidung von zwei PC-I04/PCI Karten verwendet werden kann. Der Wert kann dabei entweder als GUID oder als Zeichenkette interpretiert werden. Enthalten die ersten beiden Bytes die Zeichen „HW“, handelt es sich um eine ASCII-Zeichenkette nach ISO-8859-1 (Latin-1) mit der Seriennummer der Gerätes bzw. der Interfacekarte.
- *Description*:
[out] Beschreibung des Gerätes bzw. der Interfacekarte als 0-terminierte ASCII-Zeichenkette nach ISO-8859-1 (Latin-1).
- *Manufacturer*:
[out] Hersteller des Gerätes bzw. der Interfacekarte als 0-terminierte ASCII-Zeichenkette nach ISO-8859-1 (Latin-1).

5.1.3 VCIDEVICECAPS

Die Struktur beschreibt die Hardwareausstattung eines Gerätes bzw. einer Interfacekarte. Die Struktur hat folgenden Aufbau:

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

- *BusCtrlCount*:
[out] Anzahl der verfügbaren Busanschlüsse bzw. Controller.
- *BusCtrlTypes*:
[out] Tabelle mit bis zu **VCI_MAX_BUSCTRL** 16-Bit Werten, die den Typ des jeweiligen Anschlusses beschreiben. Gültige Einträge der Tabelle liegen im Bereich von 0 bis *BusCtrlCount*-1. Die oberen 8 Bit eines jeden Wertes der Tabelle definieren dabei den Typ vom unterstützten Feldbus, die unteren 8 Bit den Typ des verwendeten Controllers. Mit dem in *vcitype.h* definierten Makros **VCI_BUS_TYPE** bzw. **VCI_CTL_TYPE** kann der Bustyp bzw. der Controllertyp aus den Tabellenwerten extrahiert werden. Die Datei *vcitype.h* enthält außerdem vordefinierte Konstanten für die möglichen Bus- und Controller- Typen.

5.2 CAN spezifische Datentypen

Die Deklarationen aller CAN spezifischen Datentypen und Konstanten befinden sich in der Datei `<cantype.h>`.

5.2.1 CANCAPABILITIES

Der Datentyp beschreibt die Eigenschaften eines CAN-Anschlusses. Die Struktur hat folgenden Aufbau:

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

- *wCtrlType*:
[out] Typ des CAN-Controllers. Der Wert des Feldes entspricht einer der in `<cantype.h>` definierten **CAN_TYPE_** Konstanten.
- *wBusCoupling*:
[out] Art der Busankopplung. Für die Busankopplung sind folgende Werte definiert:
CAN_BUSC_LOWSPEED:
 Der CAN-Controller verfügt über eine Low-Speed Ankopplung.
CAN_BUSC_HIGHSPEED:
 Der CAN-Controller verfügt über eine High-Speed Ankopplung.
- *dwFeatures*:
[out] Unterstützte Eigenschaften. Der Wert ist eine Kombination aus einer oder mehreren der folgenden Konstanten:
CAN_FEATURE_STDOREXT:
 Der CAN-Controller unterstützt 11-Bit oder 29- Bit Nachrichten, jedoch nicht beide Formate gleichzeitig.
CAN_FEATURE_STDANDEXT:
 Der CAN-Controller unterstützt 11- und 29- Bit Nachrichten gleichzeitig.
CAN_FEATURE_RMTFRAME:
 Der CAN-Controller unterstützt Remote Transmission Request Nachrichten.
CAN_FEATURE_ERRFRAME:
 Der CAN-Controller liefert Fehlnachrichten.
CAN_FEATURE_BUSLOAD:
 Der CAN-Controller unterstützt die Berechnung der Buslast.
CAN_FEATURE_IDFILTER:
 Der CAN-Controller erlaubt die exakte Filterung von Nachrichten.
CAN_FEATURE_LISTONLY:
 Der CAN-Controller unterstützt die Betriebsart „Listen Only“.

`CAN_FEATURE_SCHEDULER:`

Zyklische Sendeliste vorhanden.

`CAN_FEATURE_GENERRFRM:`

Der CAN-Controller unterstützt die Generierung von Error- Frames.

`CAN_FEATURE_DELAYEDTX:`

Der CAN-Controller unterstützt verzögertes Senden von Nachrichten.

- *dwClockFreq:*
[out] Frequenz des primären Timer in Hz.
- *dwTscDivisor:*
[out] Teilerfaktor für den Time Stamp Counter. Der Time Stamp Counter liefert die Zeitstempel für CAN-Nachrichten. Die Frequenz des Time Stamp Counters berechnet sich aus der Frequenz des primären Timers geteilt durch den hier angegebenen Wert.
- *dwCmsDivisor:*
[out] Teilerfaktor für den Timer der zyklischen Sendeliste. Die Frequenz des Timers berechnet sich aus der Frequenz des primären Timers geteilt durch den hier angegebenen Wert. Ist keine zyklische Sendeliste vorhanden enthält das Feld den Wert 0.
- *dwCmsMaxTicks:*
[out] Maximale Zykluszeit der zyklischen Sendeliste in Timer Ticks. Ist keine zyklische Sendeliste vorhanden enthält das Feld den Wert 0.
- *dwDtxDivisor:*
[out] Teilerfaktor für den Timer der zum verzögerten Senden von Nachrichten verwendet wird. Die Frequenz des Timers berechnet sich aus der Frequenz des primären Timers geteilt durch den hier angegebenen Wert. Wird das verzögerte Senden von der CAN-Interfacekarte nicht unterstützt, enthält das Feld den Wert 0.
- *dwDtxMaxTicks:*
[out] Maximaler Verzögerungszeit für zu sendende Nachrichten in Timer Ticks. Wird das verzögerte Senden von der CAN-Interfacekarte nicht unterstützt, enthält das Feld den Wert 0.

5.2.2 CANLINESTATUS

Der Datentyp beschreibt den aktuellen Status eines CAN-Controllers. Die Struktur hat folgenden Aufbau:

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

- *bOpMode*:
[out] Aktuelle Betriebsart vom CAN-Controller. Der Wert ist eine Kombination aus einer oder mehreren **CAN_OPMODE_** Konstanten aus *<cantype.h>* und enthält den bei Aufruf der Funktion *canControllInitialize* im Parameter *bMode* angegebenen Wert.
- *bBtReg0*:
[out] Aktueller Wert vom Bus Timing Register 0. Der Wert entspricht dem BTR0 Register vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz. Weitere Informationen hierzu finden sich im Datenblatt zum SJA 1000.
- *bBtReg1*:
[out] Aktueller Wert vom Bus Timing Register 1. Der Wert entspricht dem BTR1 Register vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz. Weitere Informationen hierzu finden sich im Datenblatt zum SJA 1000.
- *bBusLoad*:
[out] Aktuelle Busbelastung in Prozent (0 bis 100). Der Wert ist nur gültig wenn die Buslastberechnung vom Controller unterstützt wird. Weitere Informationen finden sich im Kapitel über die **CANCAPABILITIES**.
- *dwStatus*:
[out] Aktueller Status vom CAN-Controller. Der Wert ist eine Kombination aus ein oder mehreren der folgenden Konstanten:
 - CAN_STATUS_TXPEND**:
Der CAN-Controller sendet momentan eine Nachricht auf den Bus.
 - CAN_STATUS_OVRRUN**:
Ein Datenüberlauf im Empfangspuffer vom CAN-Controller hat stattgefunden. Dieser Status kann nur über einen Controller Reset (siehe §4.3.1.7) zurückgesetzt werden.
 - CAN_STATUS_ERRLIM**:
Der Überlauf eines Fehlerzähler vom CAN-Controller hat stattgefunden.
 - CAN_STATUS_BUSOFF**:
Der CAN-Controller befindet sich im Zustand „BUS-OFF“.
 - CAN_STATUS_ININIT**:
Der CAN-Controller befindet sich im gestoppten Zustand.

5.2.3 CANCHANSTATUS

Der Datentyp beschreibt den aktuellen Zustand eines CAN-Nachrichtenkanals. Die Struktur hat folgenden Aufbau:

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

- *sLineStatus*:
[out] Aktueller Status vom CAN-Controller. Weitere Informationen finden sich bei der Beschreibung der Datenstruktur *CANLINESTATUS*.
- *fActivated*:
[out] Zeigt an, ob der Nachrichtenkanal momentan aktiv (TRUE) oder inaktiv (FALSE) ist.
- *fRxOverrun*:
[out] Zeigt mit dem Wert TRUE an, ob ein Überlauf im Empfangspuffer vom CAN-Controller stattgefunden hat. (siehe §5.2.2 *CAN_STATUS_OVRRUN*).
- *bRxFifoLoad*:
[out] Aktueller Füllstand vom Empfangspuffer in Prozent.
- *bTxFifoLoad*:
[out] Aktueller Füllstand vom Sendepuffer in Prozent.

5.2.4 CANSCHEDULERSTATUS

Der Datentyp beschreibt den aktuellen Status einer zyklischen Sendeliste. Die Struktur hat folgenden Aufbau:

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

- *bTaskStat*:
[out] Momentaner Zustand der Sendetask.
CAN_CTXTSK_STAT_STOPPED:
Die Sendetask ist momentan gestoppt, bzw. deaktiviert.
CAN_CTXTSK_STAT_RUNNING:
Die Sendetask wird ausgeführt, bzw. ist aktiv.

- *abMsgStat*:
Tabelle mit dem Status aller 16 Sendeobjekte. Jeder Tabelleneintrag kann dabei einen der folgenden Werte annehmen:
CAN_CTXMSG_STAT_EMPTY:
Dem Eintrag ist kein Sendeobjekt zugeordnet, bzw. der Eintrag wird momentan nicht verwendet.
CAN_CTXMSG_STAT_BUSY:
Das Sendeobjekt wird momentan bearbeitet.
CAN_CTXMSG_STAT_DONE:
Die Bearbeitung des Sendeobjekts ist abgeschlossen.

5.2.5 CANMSGINFO

Der Datentyp fasst verschiedene Informationen über CAN-Nachrichten in einem 32-Bit Wert zusammen. Der Wert kann dabei entweder byteweise oder über einzelne Bitfelder angesprochen werden.

```
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;
```

Die Informationen einer CAN-Nachricht können über das Element *Bytes* byteweise angesprochen werden. Hierfür sind folgende Felder definiert:

- *Bytes.bType*:
[in/out] Typ der Nachricht. Siehe auch *Bits.type*.
- *Bytes.bReserved*:
Reserviert. Siehe auch *Bits.res*.
- *Bytes.bFlags*:
[in/out] Verschiedene Flags. Siehe auch *Bits.dlc*, *Bits.ovr*, *Bits.srr*, *Bits.rtr* und *Bits.ext*.
- *Bytes.bAccept*:
[out] Zeigt bei Empfangsnachrichten an, welcher Filter die Nachricht akzeptiert hat. Siehe auch *Bits.afc*.

Mit dem Element *Bits* kann bitweise auf die Informationen einer CAN-Nachricht zugegriffen werden. Es sind folgende Bitfelder definiert:

- *Bits.type*:

[in/out] Typ der Nachricht. Für Empfangsnachrichten sind die im folgenden aufgeführten Typen definiert. Für Sendenachrichten ist momentan nur der Nachrichtentyp **CAN_MSGTYPE_DATA** definiert, andere Werte sind hier nicht erlaubt.

CAN_MSGTYPE_DATA:

Normale Nachricht. Alle regulären Empfangsnachrichten sind von diesem Typ. Im Feld *CANMSG.dwMsgId* befindet sich die ID der Nachricht, im Feld *CANMSG.dwTime* der Empfangszeitpunkt. Die Felder *CANMSG.abData* enthalten je nach Länge (siehe *Bits.dlc*) die Datenbytes der Nachricht. Bei Sendenachrichten ist im Feld *CANMSG.dwMsgId* die ID und in den Feldern *CANMSG.abData* je nach Länge die Datenbytes anzugeben. Das Feld *CANMSG.dwTime* wird normalerweise auf 0 gesetzt, es sei denn die Nachricht soll verzögert gesendet werden. In diesem Fall ist hier die Verzögerungszeit in Ticks anzugeben. Siehe hierzu auch die Beschreibung zur Struktur **CANMSG**, bzw. Kapitel 3.1.3.3.

CAN_MSGTYPE_INFO:

Informationsnachricht. Dieser Nachrichtentyp wird bei bestimmten Ereignissen, bzw. bei Änderungen am Zustand des Controllers in die Empfangspuffer aller aktivierten Nachrichtenkanäle eingetragen. Das Feld *CANMSG.dwMsgId* der Nachricht hat dabei immer den Wert 0xFFFFFFFF. Das Feld *CANMSG.abData[0]* enthält einen der folgenden Werte:

Konstante	Bedeutung
CAN_INFO_START	Der CAN-Controller wurde gestartet. Das Feld <i>CANMSG.dwTime</i> enthält den relativen Startzeitpunkt (normalerweise 0).
CAN_INFO_STOP	Der CAN-Controller wurde gestoppt. Das Feld <i>CANMSG.dwTime</i> enthält den Wert 0.
CAN_INFO_RESET	Der CAN-Controller wurde zurückgesetzt. Das Feld <i>CANMSG.dwTime</i> enthält den Wert 0.

CAN_MSGTYPE_ERROR:

Fehlernachricht. Dieser Nachrichtentyp wird beim Auftreten von Busfehlern in die Empfangspuffer aller aktivierten Nachrichtenkanäle eingetragen, sofern bei Aufruf der Funktion *canControllInitialize* das Flag **CAN_OPMODE_ERRFRAME** im Parameter *CANMSG.bMode* angegeben wurde. Das Feld *CANMSG.dwMsgId* der Nachricht hat dabei immer den Wert 0xFFFFFFFF. Der Zeitpunkt des Ereignisses ist im Feld *CANMSG.dwTime* der Nachricht vermerkt. Das Feld *CANMSG.abData[0]* enthält einen der folgenden Werte:

Konstante	Bedeutung
CAN_ERROR_STUFF	Bit Stuff Fehler
CAN_ERROR_FORM	Format Fehler
CAN_ERROR_ACK	Acknowledge Fehler
CAN_ERROR_BIT	Bit Fehler
CAN_ERROR_CRC	CRC Fehler
CAN_ERROR_OTHER	Anderer unspezifizierter Fehler

Das Feld *CANMSG.abData[1]* der Nachricht enthält das niederwertige Byte vom aktuellen CAN-Status (siehe auch *CANLINESTATUS.dwStatus*). Der Inhalt der anderen Datenfelder ist undefiniert.

CAN_MSGTYPE_STATUS:

Statusnachricht. Dieser Nachrichtentyp wird bei Änderungen vom Controllerstatus in die Empfangspuffer aller aktivierten Nachrichtenkanäle eingetragen. Das Feld *CANMSG.dwMsgId* der Nachricht hat dabei immer den Wert 0xFFFFFFFF. Der Zeitpunkt des Ereignisses ist im Feld *CANMSG.dwTime* der Nachricht vermerkt. Das Feld *CANMSG.abData[0]* enthält das niederwertige Byte vom aktuellen CAN-Status (siehe auch *CANLINESTATUS.dwStatus*). Der Inhalt der anderen Datenfelder ist undefiniert.

CAN_MSGTYPE_WAKEUP:

Wird momentan nicht verwendet, bzw. ist zukünftige Erweiterungen reserviert.

CAN_MSGTYPE_TIMEOVR:

Zählerüberlauf. Nachrichten dieses Typs werden bei einem Überlauf des 32 Bit Zeitstempels von CAN-Nachrichten generiert. Im Feld *CANMSG.dwTime* der Nachricht befindet sich der Zeitpunkt des Ereignisses (normalerweise 0) und im Feld *CANMSG.dwMsgId* die Anzahl von Timerüberläufen nach dem Empfang der letzten Zählerüberlauf-Nachricht. Der Inhalt der Datenfelder *CANMSG.abData* ist undefiniert.

CAN_MSGTYPE_TIMERST:

Wird momentan nicht verwendet, bzw. ist zukünftige Erweiterungen reserviert..

- *Bits.res*:
Reserviert für zukünftige Erweiterung. Aus Kompatibilitätsgründen sollte das Feld immer auf 0 gesetzt werden.
- *Bits.dlc*:
[in/out] Data Length Code. Legt die Anzahl gültiger Datenbytes in den Feldern *CANMSG.abData* der Nachricht fest.
- *Bits.ovr*:
[out] Data Overrun. Das Bit wird auf 1 gesetzt, wenn der Empfangspuffer nach dem Eintragen dieser Nachricht voll ist.

- *Bits.srr*:
[in/out] Self Reception Request. Wird das Bit bei Sendenachrichten gesetzt, wird die Nachricht in den Empfangspuffer eingetragen, sobald diese auf dem Bus gesendet wurde. Bei Empfangsnachrichten deutet ein gesetztes Bit darauf hin, dass es sich um eine empfangene Self- Reception Nachricht handelt.
- *Bits.rtr*:
[in/out] Remote Transmission Request.
- *Bits.ext*:
[in/out] Nachricht mit erweiterter 29- Bit ID.
- *Bits.afc*:
[out] Acceptance Filter Code. Bei Empfangsnachrichten gibt dieses Feld den Filter an, der die Nachricht passieren ließ.

5.2.6 CANMSG

Der Datentyp beschreibt den Aufbau eines CAN-Nachrichtentelegramms.

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

- *dwTime*:
[in/out] Bei Empfangsnachrichten enthält dieses Feld den relativen Empfangszeitpunkt der Nachricht in Ticks. Die Auflösung eines Ticks kann hierbei aus den Feldern *dwClockFreq* und *dwTscDivisor* der Struktur **CANCAPABILITIES** nach folgender Formel berechnet werden:

$$\text{Auflösung [s]} = \text{dwTscDivisor} / \text{dwClockFreq}$$

Der Überlauf der 32-Bit *dwTime* Variable wird durch eine Timer Overrun message übertragen. (CAN_MSGTYPE_TIMEOVR §5.2.5)

Bei Sendenachrichten legt das Feld fest um wie viele Ticks die Nachricht verzögert auf den Bus gesendet werden soll. Die Verzögerungszeit zwischen der zuletzt gesendeten Nachricht und der neuen Nachricht kann dabei mittels der Felder *dwClockFreq* und *dwDtxDivisor* der Struktur **CANCAPABILITIES** nach folgender Formel berechnet werden.

$$\text{Verzögerungszeit [s]} = (\text{dwDtxDivisor} / \text{dwClockFreq}) * \text{dwTime}$$

Die maximal mögliche Verzögerungszeit wird durch das Feld *dwDtxMaxTicks* der Struktur **CANCAPABILITIES** bestimmt.

- *dwMsgId*:
[in/out] CAN ID der Nachricht im Intel-Format (rechtsbündig) ohne RTR Bit.
- *uMsgInfo*:
[in/out] Bitfeld mit Informationen über den Nachrichtentyp. Eine Beschreibung des Bitfeldes befindet sich in Kapitel 5.2.5.
- *abData*:
[in/out] Array für bis zu 8 Datenbytes. Die Anzahl gültiger Datenbytes wird durch das Feld *uMsgInfo.Bits.dlc* bestimmt.

5.2.7 CANYCLICTXMSG

Der Datentyp beschreibt den Aufbau einer zyklischen CAN-Sendenachricht.

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

wCycleTime:

[in/out] Zykluszeit der Sendenachricht in Ticks. Die Zykluszeit kann mittels der Felder *dwClockFreq* und *dwCmsDivisor* der Struktur **CANCAPABILITIES** nach folgender Formel berechnet werden.

$$\text{Zykluszeit [s]} = (\text{dwCmsDivisor} / \text{dwClockFreq}) * \text{wCycleTime}$$

Die maximal mögliche Zykluszeit ist auf den Wert im Feld *dwCmsMaxTicks* der Struktur **CANCAPABILITIES** begrenzt.

- *bIncrMode*:
[in/out] Mittels dieses Feldes wird festgelegt, ob ein Teil der zyklischen Sendenachricht nach jedem Sendevorgang automatisch inkrementiert wird.

CAN_CTXMSG_INC_NO:

Es erfolgt kein automatisches Inkrementieren eines Nachrichtenfeldes.

CAN_CTXMSG_INC_ID:

Inkrementiert das Feld *dwMsgId* der Nachricht nach jedem Sendevorgang um 1. Erreicht das Feld den Wert 2048 (11-bit ID) bzw. 536.870.912 (29-bit ID) erfolgt automatisch ein Überlauf auf 0.

CAN_CTXMSG_INC_8:

Inkrementiert ein 8-Bit Wert im Datenfeld *abData* der Nachricht. Das zu inkrementierende Datenfeld wird dabei im Feld *bByteIndex* festgelegt. Beim Überschreiten des Maximalwertes 255 erfolgt automatisch ein Überlauf auf 0.

CAN_CTXMSG_INC_16:

Inkrementiert ein 16-Bit Wert im Datenfeld *abData* der Nachricht. Das niederwertige Byte des zu inkrementierenden 16-Bit Wertes wird dabei im Feld *bByteIndex* festgelegt. Das höherwertige Byte befindet sich bei *abData[bByteIndex+1]*. Beim Überschreiten des Maximalwertes 65535 erfolgt automatisch ein Überlauf auf 0.

- *bByteIndex*:
[in/out] Wird im Feld *bIncrMode* der Wert **CAN_CTXMSG_INC_8** angegeben bestimmt dieses Feld den Index des Datenbytes im Datenfeld *abMsgData*, das nach jedem Sendevorgang automatisch inkrementiert werden soll. Wird für *bIncrMode* der Wert **CAN_CTXMSG_INC_16** angegeben, legt dieses Feld den Index vom niederwertigen Byte (LSB) des 16-Bit Wertes im Datenfeld *abMsgData* fest. Das höherwertige Byte (MSB) des 16-Bit Wertes befindet sich im Datenfeld *abMsgData[bByteIndex+1]*.
- *dwMsgId*:
[in/out] ID der Nachricht (CAN-ID) im Intel-Format (rechtsbündig) ohne RTR-Bit.
- *uMsgInfo*:
[in/out] Bitfeld mit Informationen über den Nachrichtentyp. Eine Beschreibung des Bitfeldes befindet sich in Kapitel 5.2.5.
- *abData*:
[in/out] Array für bis zu 8 Datenbytes. Die Anzahl gültiger Datenbytes wird durch das Feld *uMsgInfo.Bits.dlc* bestimmt.

5.3 LIN spezifische Datentypen

Die Deklarationen aller LIN spezifischen Datentypen und Konstanten findet sich in der Datei *<lintype.h>*.

5.3.1 LINCAPABILITIES

Der Datentyp beschreibt die Eigenschaften eines LIN-Anschlusses. Die Struktur hat folgenden Aufbau:

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

- *dwFeatures*:
[out] Unterstützte Eigenschaften. Der Wert ist eine Kombination aus einer oder mehreren der folgenden Konstanten:
 - LIN_FEATURE_MASTER**:
Der LIN-Controller unterstützt die „Master“ Betriebsart.
 - LIN_FEATURE_AUTORATE**:
Der LIN-Controller unterstützt die automatische Bitratenerkennung.
 - LIN_FEATURE_ERRFRAME**:
Der LIN-Controller liefert Fehlernachrichten.
 - LIN_FEATURE_BUSLOAD**:
Der LIN-Controller unterstützt die Berechnung der Buslast.
- *dwClockFreq*:
[out] Frequenz vom primären Timer in Hz.

- *dwTscDivisor*:
[out] Teilerfaktor vom Time Stamp Counter. Der Time Stamp Counter liefert die Zeitstempel für LIN-Nachrichten. Die Frequenz vom Time Stamp Counter berechnet sich aus der Frequenz vom primären Timer geteilt durch den hier angegebenen Wert.

5.3.2 LINLINESTATUS

Der Datentyp beschreibt den aktuellen Status eines LIN-Controllers. Die Struktur hat folgenden Aufbau:

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

- *bOpMode*:
[out] Aktuelle Betriebsart vom LIN-Controller. Der Wert ist eine Kombination aus einer oder mehreren **LIN_OPMODE_** Konstanten aus *<lintype.h>* und enthält den bei Aufruf der Funktion *linControlInitialize* im Parameter *bMode* angegebenen Wert.
- *bReserved*:
[out] nicht verwendet.
- *wBitrate*:
[out] Aktuelle eingestellte Übertragungsrate in Bits pro Sekunde.
- *dwStatus*:
[out] Aktueller Status vom LIN-Controller. Der Wert ist eine Kombination aus ein oder mehreren der folgenden Konstanten:
LIN_STATUS_OVRRUN:
Es hat ein Datenüberlauf im Empfangspuffer vom Controller stattgefunden.
LIN_STATUS_ININIT:
Der Controller befindet sich im gestoppten Zustand.

5.3.3 LINMONITORSTATUS

Der Datentyp beschreibt den aktuellen Zustand eines LIN-Nachrichtenmonitors. Die Struktur hat folgenden Aufbau:

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

- *sLineStatus*:
[out] Aktueller Status vom LIN-Controller. Weitere Informationen finden sich bei der Beschreibung der Datenstruktur *LINLINESTATUS*.
- *fActivated*:
[out] Zeigt an, ob der Nachrichtenmonitor momentan aktiv (TRUE) oder inaktiv (FALSE) ist.
- *fRxOverrun*:
[out] Zeigt mit dem Wert TRUE an, ob der Empfangspuffer übergelaufen ist.
- *bRxFifoLoad*:
[out] Aktueller Füllstand vom Empfangspuffer in Prozent.

5.3.4 LINMSGINFO

Der Datentyp fasst verschiedene Informationen über LIN-Nachrichten in einem 32-Bit Wert zusammen. Der Wert kann dabei entweder bytewise oder über einzelne Bitfelder angesprochen werden.

```
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;
```

Die Informationen einer LIN-Nachricht können über das Element *Bytes* bytewise angesprochen werden. Hierfür sind folgende Felder definiert:

- *Bytes.bPid*:
[in/out] Protected Identifier. Siehe auch *Bits.pid*.
- *Bytes.bType*:
[in/out] Typ der Nachricht. Siehe auch *Bits.type* und *Bits.ecs*.
- *Bytes.bDlen*:
[in/out] Datenlänge, siehe auch *Bits.dlen*.
- *Bytes.bFlags*:
[in/out] Verschiedene Flags. siehe auch *Bits.ecs*, *Bits.sor*, *Bits.ovr* und *Bits.ido*.

Mit dem Element *Bits* kann bitweise auf die Informationen einer LIN-Nachricht zugegriffen werden. Es sind folgende Bitfelder definiert:

- *Bits.pid*:
[in/out] Protected Identifier der Nachricht.
- *Bits.type*:
[in/out] Typ der Nachricht. Für Empfangsnachrichten sind die im folgenden aufgeführten Typen definiert. Für Sendenachrichten sind momentan nur der Nachrichtentyp **LIN_MSGTYPE_DATA** und **LIN_MSGTYPE_WAKEUP** definiert, andere Werte sind hier nicht erlaubt.

LIN_MSGTYPE_DATA:

Normale Nachricht. Alle regulären Empfangsnachrichten sind von diesem Typ. Im Feld *LINMSG.bPid* findet sich die ID der Nachricht, im Feld *LINMSG.dwTime* der Empfangszeitpunkt. Das Feld *LINMSG.abData* enthält je nach Länge (siehe *Bits.dlen*) die Datenbytes der Nachricht. In der Master-Betriebsart können Nachrichten dieses Typs auch gesendet werden. Dabei muss im Feld *LINMSG.bPid* die ID und im Feld *LINMSG.abData* je nach Länge (*Bits.dlen*) die zu sendenden Daten angegeben werden. Das Feld *LINMSG.dwTime* wird auf 0 gesetzt. Um nur die ID ohne Daten zu senden, wird *Bits.ido* auf 1 gesetzt.

LIN_MSGTYPE_INFO:

Informationsnachricht. Dieser Nachrichtentyp wird bei bestimmten Ereignissen, bzw. bei Änderungen am Zustand des Controllers in die Empfangspuffer aller aktivierten Nachrichtenmonitore eingetragen. Das Feld *LINMSG.bPid* der Nachricht hat dabei immer den Wert 0xFF. Das Feld *LINMSG.abData[0]* enthält einen der folgenden Werte:

Konstante	Bedeutung
LIN_INFO_START	Der Controller wurde gestartet. Das Feld <i>LINMSG.dwTime</i> enthält den relativen Startzeitpunkt (normalerweise 0).
LIN_INFO_STOP	Der Controller wurde gestoppt. Das Feld <i>LINMSG.dwTime</i> enthält den Wert 0.
LIN_INFO_RESET	Der Controller wurde zurückgesetzt. Das Feld <i>LINMSG.dwTime</i> enthält den Wert 0.

LIN_MSGTYPE_ERROR:

Fehlernachricht. Dieser Nachrichtentyp wird beim Auftreten von Busfehlern in die Empfangspuffer aller aktivierten Nachrichtenmonitore eingetragen, sofern beim Initialisieren des Controllers das Flag **LIN_OPMODE_ERRORS** angegeben wurde. Das Feld *LINMSG.bPid* der Nachricht hat dabei immer den Wert 0xFF. Der Zeitpunkt des Ereignisses ist im Feld *LINMSG.dwTime* der Nachricht vermerkt. Das Feld *LINMSG.abData[0]* enthält einen der folgenden Werte:

Konstante	Bedeutung
LIN_ERROR_BIT	Bitfehler
LIN_ERROR_CHKSUM	Checksummenfehler
LIN_ERROR_PARITY	Paritäts-Fehler des Identifiers
LIN_ERROR_SLNORE	„Slave“ Antwortet nicht
LIN_ERROR_SYNC	Ungültiges Synchronisationsfeld
LIN_ERROR_NOBUS	Keine Busaktivität
LIN_ERROR_OTHER	Anderer, unspezifizierter Fehler

Das Feld *LINMSG.abData[1]* der Nachricht enthält das niederwertige Byte vom aktuellen Status (siehe auch *LINLINESTATUS.dwStatus*). Der Inhalt der anderen Datenfelder ist undefiniert.

LIN_MSGTYPE_STATUS:

Statusnachricht. Dieser Nachrichtentyp wird bei Änderungen vom Controllerstatus in die Empfangspuffer aller aktivierten Nachrichtenkanäle eingetragen. Das Feld *LINMSG.bPid* der Nachricht hat dabei immer den Wert 0xFF. Der Zeitpunkt des Ereignisses ist im Feld *LINMSG.dwTime* der Nachricht vermerkt. Das Feld *LINMSG.abData[0]* enthält das niederwertige Byte vom aktuellen Status (siehe auch *LINLINESTATUS.dwStatus*). Der Inhalt der anderen Datenfelder ist undefiniert.

LIN_MSGTYPE_WAKEUP:

Nur für Sendenachrichten. Nachrichten dieses Typs generieren ein Wake-Up Signal auf dem Bus. Die Felder *LINMSG.dwTime*, *LINMSG.bPid* und *LINMSG.bDlen* sind ohne Bedeutung.

LIN_MSGTYPE_TMOVR:

Zählerüberlauf. Nachrichten dieses Typs werden bei einem Überlauf des 32 Bit Zeitstempels von LIN-Nachrichten generiert. Im Feld *LINMSG.dwTime* der Nachricht findet sich der Zeitpunkt des Ereignisses (normalerweise 0) und im Feld *LINMSG.bDlen* die Anzahl der Timerüberläufe. Der Inhalt der Datenfelder *LINMSG.abData* ist undefiniert, das Feld *LINMSG.bPid* hat immer den Wert 0xFF.

LIN_MSGTYPE_SLEEP:

Goto Sleep Nachricht. Die Felder *LINMSG.dwTime*, *LINMSG.bPid* und *LINMSG.bDlen* sind ohne Bedeutung.

- *Bits.dlen*:
[in/out] Anzahl der gültigen Datenbytes im Feld *LINMSG.abData* der Nachricht.
- *Bits.ecs*:
[in/out] Enhanced Checksum. Das Bit wird auf 1 gesetzt, wenn es sich um eine Nachricht mit erweiterter Checksumme nach LIN 2.0 handelt.

- *Bits.sor*:
[out] Sender of Response. Das Bit wird bei Nachrichten gesetzt, die der LIN-Controller selbst versendet hat, d.h. bei Nachrichten, für die der Controller einen Eintrag in der Antworttabelle unterhält.
- *Bits.ovr*:
[out] Data Overrun. Das Bit wird auf 1 gesetzt, wenn der Empfangs- FIFO nach dem Eintragen dieser Nachricht voll ist.
- *Bits.ido*:
[in] ID Only. Das Bit ist nur bei Nachrichten mit dem Typ `LIN_MSGTYPE_DATA` relevant, die direkt gesendet werden. Wird das Bit bei Sendenachrichten auf 1 gesetzt, wird nur die ID ohne Daten übertragen und dient somit in der Master-Betriebsart zum Umschalten der IDs. Bei allen anderen Nachrichtentypen ist dieses Bit ohne Bedeutung.
- *Bits.res*:
Reserviert für zukünftige Erweiterungen. Diese Feld ist normalerweise 0.

5.3.5 LINMSG

Der Datentyp beschreibt den Aufbau von LIN- Nachrichtentelegrammen.

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

- *dwTime*:
[in/out] Bei Empfangsnachrichten enthält diese Feld den relativen Empfangszeitpunkt der Nachricht in Ticks. Die Auflösung eines Ticks lässt sich aus den Felder *dwClockFreq* und *dwTscDivisor* der Struktur *LINCAPABILITIES* nach folgender Formel berechnen:

$$\text{Auflösung [s]} = \text{dwTscDivisor} / \text{dwClockFreq}$$

- *uMsgInfo*:
[in/out] Bitfeld mit Informationen über die Nachricht. Eine Beschreibung des Bitfeldes findet sich in Kapitel 5.3.4.
- *abData*:
[in/out] Array für bis zu 8 Datenbytes. Die Anzahl gültiger Datenbytes wird durch das Feld *uMsgInfo.Bits.dlen* bestimmt.