

VCI - Virtual CAN Interface

.Net-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	5
2	Geräteverwaltung und Gerätezugriff	8
2.1	Übersicht.....	9
2.2	Auflisten der verfügbaren IXXAT CAN-Interfacekarten	10
2.3	Zugriff auf eine IXXAT CAN-Interfacekarte	12
3	Kommunikationskomponenten	14
3.1	First-In- First-Out-Speicher (FIFO).....	14
3.1.1	Funktionsweise von Empfangs-FIFOs.....	15
3.1.2	Funktionsweise von Sende-FIFOs.....	16
4	Zugriff auf den Feldbus	18
4.1	Übersicht.....	18
4.2	CAN Anschluss	20
4.2.1	Übersicht.....	20
4.2.2	Socket-Schnittstelle.....	21
4.2.3	Nachrichtenkanäle	22
4.2.3.1	Empfang von CAN-Nachrichten	24
4.2.3.2	Senden von CAN-Nachrichten	25
4.2.3.3	Verzögertes Senden von CAN-Nachrichten.....	26
4.2.4	Steuereinheit	27
4.2.4.1	Kontrollierzustände.....	28
4.2.4.2	Nachrichtenfilter	30
4.2.5	Zyklische Sendeliste	32
4.3	LIN Anschluss	36
4.3.1	Übersicht.....	36
4.3.2	Socket-Schnittstelle.....	37
4.3.3	Nachrichtenmonitore.....	37
4.3.3.1	Empfang von LIN-Nachrichten	39
4.3.4	Steuereinheit	40
4.3.4.1	Kontrollierzustände.....	41
4.3.4.2	Senden von LIN-Nachrichten.....	42
5	Schnittstellenbeschreibung	44

1 Systemübersicht

Das VCI (Virtual Card Interface) ist ein Treiber dessen Aufgabe darin besteht Applikationen einen einheitlichen Zugriff auf unterschiedliche IXXAT CAN-Interfacekarten zu ermöglichen. Nachfolgende Abbildung zeigt den prinzipiellen Aufbau des Systems und dessen Komponenten.

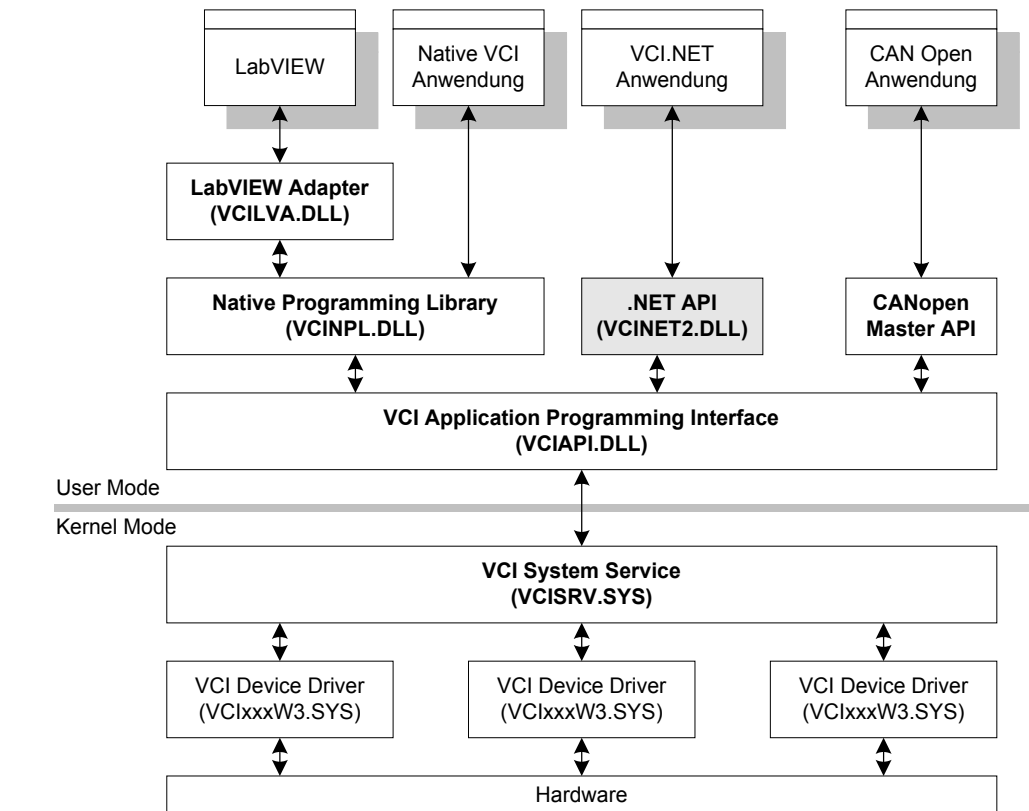


Bild 1-1: Systemkomponenten

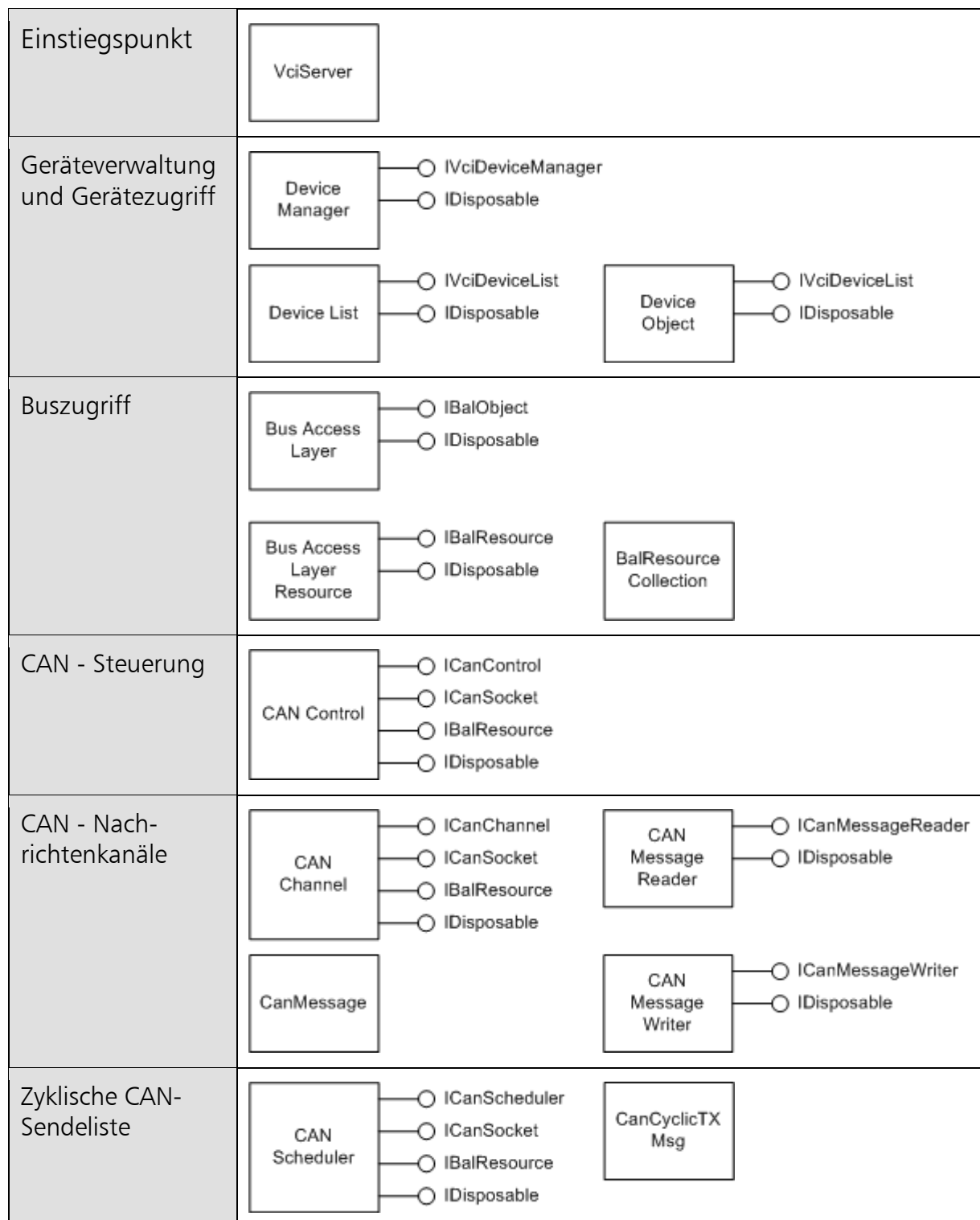
Das VCI besteht im wesentlichen aus den folgenden Komponenten:

- Native VCI-Programmierschnittstelle (VCINPL.DLL)
- VCI .NET 2.0 Programmierschnittstelle (VCINET2.DLL)
- VCI System Service API (VCI-API.DLL)
- VCI-System-Service (VCISRV.SYS)
- Ein oder mehrere VCI-Gerätetreiber (VCIxxxW3.SYS)

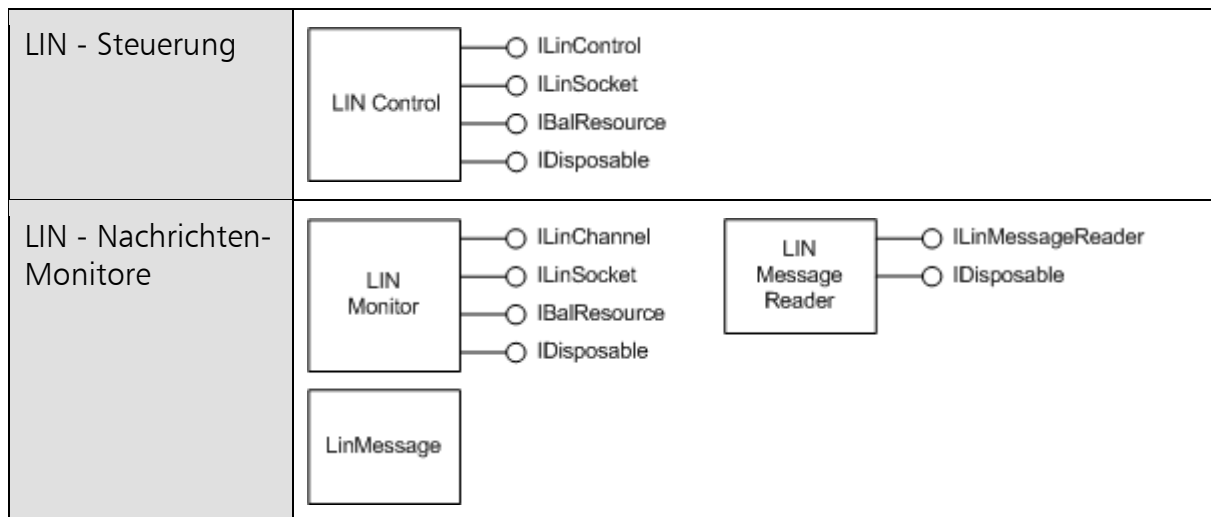
Die Programmierschnittstellen stellen die Verbindung zwischen dem VCI-System-Service oder kurz VCI-Server und den Applikationsprogrammen über einen Satz vordefinierter Schnittstellen und Funktionen her. Die .NET 2.0 API dient ausschließlich der Anpassung an die COM-basierte Programmierschnittstelle.

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

Die im folgenden betrachtete Programmierschnittstelle besteht aus folgenden Teilkomponenten und .NET Interfaces/Klassen:



Systemübersicht



2 Geräteverwaltung und Gerätezugriff

2.1 Übersicht

Die Komponenten der Geräteverwaltung erlauben das Auflisten und den Zugriff auf die beim VCI-Server angemeldeten Gerätetreiber und CAN-Interfacekarten.

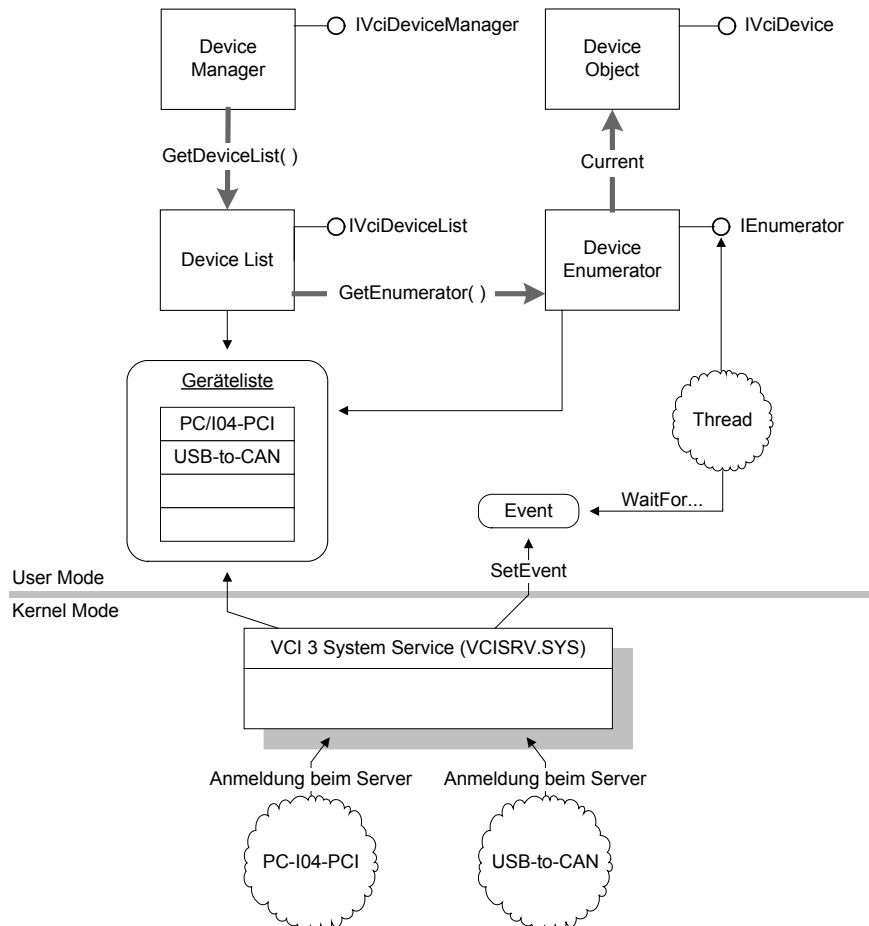


Bild 2-1: Komponenten der Geräteverwaltung

Der VCI-Server verwaltet alle IXXAT CAN-Interfacekarten in einer systemweiten globalen Liste, die nachfolgend kurz als Geräteliste bezeichnet wird.

Die Anmeldung einer CAN-Interfacekarte beim Server erfolgt automatisch beim Start des Computers, bzw. wenn eine Verbindung zwischen Computer und CAN-Interfacekarte hergestellt wird. Ist eine CAN-Interfacekarte nicht mehr verfügbar, weil z. B. die Verbindung unterbrochen wurde, wird dieser automatisch aus der Geräteliste entfernt.

Der Zugriff auf alle verfügbaren IXXAT CAN-Interfacekarten erfolgt über den Gerätemanager, bzw. dessen Schnittstelle *IVciDeviceManager*. Eine Referenz auf diese Schnittstelle liefert die statische Methode *VciServer.GetDeviceManager*.

2.2 Auflisten der verfügbaren IXXAT CAN-Interfacekarten

Zugriff auf die globale Geräteliste erhält man durch Aufruf der Methode *IVciDeviceManager.GetDeviceList*. Die Methode liefert bei erfolgreicher Ausführung eine Referenz auf die Schnittstelle *IVciDeviceList* der Geräteliste zurück. Mit ihr können Änderungen an der Geräteliste überwacht und Enumeratoren für die Geräteliste angefordert werden.

Die Methode *IVciDeviceList.GetEnumerator* liefert das *IEnumerator* Interface eines neuen Enumeratorobjekts für die Geräteliste. Das Property *IEnumerator.Current* liefert bei jedem Abruf ein neues Geräteobjekt mit Informationen zu einer CAN-Interfacekarte. Für den Zugriff auf diese Informationen muss die von Property *Current* des Standard-Interfaces *IEnumerator* gelieferte pure Objektreferenz in den Typ *IVciDevice* umgewandelt werden. Die Methode *IEnumerator.MoveNext* erhöht einen internen Index, so dass *IEnumerator.Current* ein Geräteobjekt für die nächste CAN-Interfacekarte liefern kann. Nachfolgend sind die wichtigsten, von *IVciDevice* bereitgestellten Informationen über eine CAN-Interfacekarte zusammengestellt:

- *Description*: String mit der Bezeichnung der CAN-Interfacekarte, z. B. „USB-to-CAN compact“.
- *VciObjectId*: Eindeutige Kennzahl der CAN-Interfacekarte. Jeder CAN-Interfacekarte wird bei dessen Anmeldung eine systemweit eindeutige Kennzahl zugewiesen.
- *DeviceClass*: Geräteklasse. Jeder Gerätetreiber kennzeichnet seine unterstützte CAN-Interfacekarte-Klasse mit einer eindeutigen Kennzahl (*GUID*). Unterschiedliche CAN-Interfacekarten gehören unterschiedlichen Geräteklassen an. So hat z. B. die IPC-I165/PCI eine andere Klassen, als die PC-I04/PCI.
- *UniqueHardwareId*: Hardwarekennung. Jede CAN-Interfacekarte hat eine eindeutige und einmalige Hardwarekennung. Die Hardwarekennung kann z. B. dazu verwendet werden, um zwischen zwei PC-I04/PCI Karten zu unterscheiden oder um nach einer CAN-Interfacekarte mit einer bestimmten Hardwarekennung zu suchen.
- *DriverVersion*: Versionsnummer des Treibers.
- *HardwareVersion*: Versionsnummer der CAN-Interfacekarte.
- *Equipment*: Technische Ausstattung der IXXAT CAN-Interfacekarte. Die in *Equipment* enthaltene Tabelle von *VciCtrlInfo*-Strukturen gibt Auskunft über die Anzahl und Art der auf einer CAN-Interfacekarte vorhandenen Busanschlüsse. Nachfolgende Abbildung zeigt eine CAN-Interfacekarte mit zwei Anschlüssen.

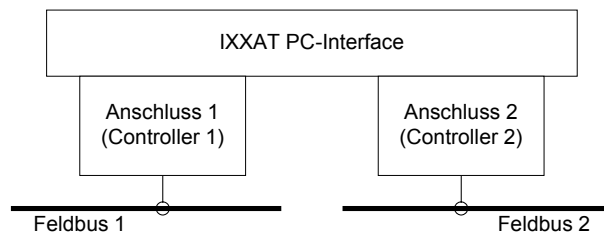


Bild 2-2: CAN-Interfacekarte mit zwei Busanschlüssen.

Der Tabelleneintrag 0 beschreibt Busanschluss 1, der Tabelleneintrag 1 den Busanschluss 2, usw.

Die Liste ist vollständig durchlaufen, wenn die Methode *IEnumerator.MoveNext* den Wert **false** zurückliefert.

Der interne Index lässt sich mit der Methode *IEnumerator.Reset* wieder auf den Anfang zurückstellen, so dass ein späterer Aufruf von *IEnumerator.MoveNext* die Enumeratorposition wieder auf die erste CAN-Interfacekarte setzt.

IXXAT CAN-Interfacekarten die sich während des Betriebs hinzufügen oder entfernen lassen, wie z. B. USB-to-CAN compact melden sich nach dem Einstecken beim VCI-Server an bzw. wieder ab, wenn die CAN-Interfacekarte ausgesteckt wird.

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

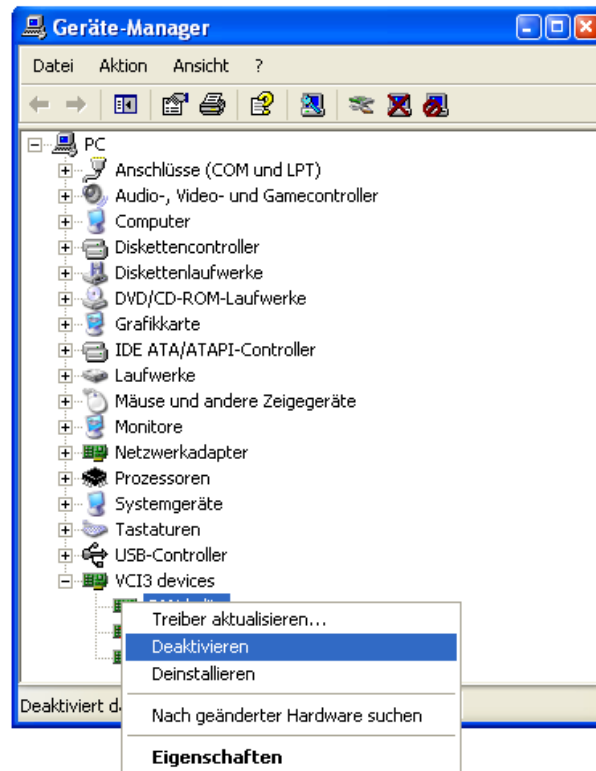


Bild 2-3: Windows Gerätemanager

Applikationen können Änderungen an der Geräteliste überwachen, indem sie ein *AutoResetEvent*- oder ein *ManualResetEvent*-Objekt erzeugen und dieses mittels *IVciDeviceList.AssignEvent* der Liste zuteilen. Sinnvollerweise sollte hierbei ein *AutoResetEvent* verwendet werden. Meldet sich nach Aufruf der Methode ein Gerät beim VCI Server an oder ab, wird das Event in den signalisierten Zustand gesetzt.

2.3 Zugriff auf eine IXXAT CAN-Interfacekarte

Alle IXXAT CAN-Interfacekarten stellen ein oder mehrere Komponenten bzw. Zugriffsebenen für unterschiedliche Anwendungsbereiche bereit. Interessant ist hierbei jedoch nur der Bus Access Layer (BAL). Dieser erlaubt die Steuerung der Controller und ermöglicht die Kommunikation mit dem Feldbus. Der BAL lässt sich über die Methode *IVciDevice.OpenBusAccessLayer* öffnen.

Die unterschiedlichen Zugriffsebenen einer CAN-Interfacekarte können nicht gleichzeitig geöffnet werden. Öffnet z. B. eine Anwendung den Bus Access Layer, so kann die von der CANopen Master API verwendete Zugriffsebene erst wieder geöffnet werden, nachdem der BAL freigegeben, bzw. geschlossen wurde.

Bestimmte Zugriffsebenen sind zusätzlich gegen mehrfaches Öffnen gesichert. So ist es z. B. nicht möglich, dass zwei CANopen-Applikationen eine CAN-Interfacekarte gleichzeitig verwenden. Diese Einschränkung gilt jedoch nicht für den BAL.

Der BAL kann von mehreren Programmen gleichzeitig geöffnet werden. Damit besteht die Möglichkeit, dass unterschiedliche Applikationen gleichzeitig auf die verschiedenen Busanschlüsse zugreifen. Weitere Informationen zum BAL finden sich in Kapitel 4.

3 Kommunikationskomponenten

3.1 First-In- First-Out-Speicher (FIFO)

Das VCI enthält eine Implementierung für so genannte First-In- First-Out-Speicher (FIFO).

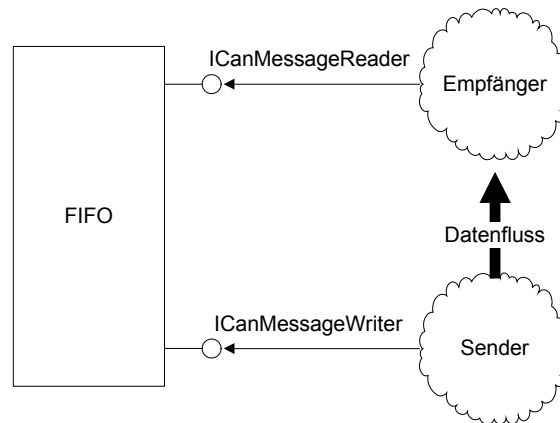


Bild 3-1: FIFO Komponente und Datenfluss

FIFOs werden verwendet, um Daten von einem Sender zu einem Empfänger zu übertragen. Der Sender trägt hierzu Daten über ein Writer-Schnittstelle (z. B. *ICanMessageWriter*) in den FIFO ein. Ein Empfänger kann diese Daten zu einem späteren Zeitpunkt über eine Reader-Schnittstelle (z. B. *ICanMessageReader*) wieder auslesen.

Schreib- und Lesezugriffe auf einen FIFO sind dabei gleichzeitig möglich, d.h. der Empfänger kann Daten lesen, während der Sender neue Daten schreibt. Im Gegensatz dazu ist es jedoch nicht erlaubt, dass mehrere Sender bzw. Empfänger gleichzeitig auf einen FIFO zugreifen.

Ein gleichzeitiger Zugriff auf einen FIFO durch mehrere Empfänger bzw. Sender wird vom FIFO verhindert, indem die jeweilige Reader- und Writer-Schnittstellen (z. B. *ICanMessageReader* / *ICanMessageWriter*) jeweils nur ein einziges mal geöffnet werden können. Erst wenn eine Schnittstelle mittels *IDisposable.Dispose* freigegeben wurde, kann diese erneut geöffnet werden. Damit ist jedoch nicht ausgeschlossen, dass unterschiedliche Threads einer Applikation gleichzeitig auf eine Schnittstelle zugreifen.

Der Programmierer muss darauf achten, dass die Funktionen einer Schnittstelle nicht von mehreren Threads gleichzeitig aufgerufen werden. Andernfalls ist er selbst dafür verantwortlich diese Aufruf gegeneinander zu verriegeln. In der Regel ist es aber die bessere Alternative, für den zweiten Thread einen separaten Nachrichtenkanal anzulegen.

3.1.1 Funktionsweise von Empfangs-FIFOs

Nachfolgende Abbildung zeigt die Funktionsweise von Empfangs-FIFOs. Die dicken Pfeile zeigen dabei den Datenfluss vom Sender zum Empfänger an.

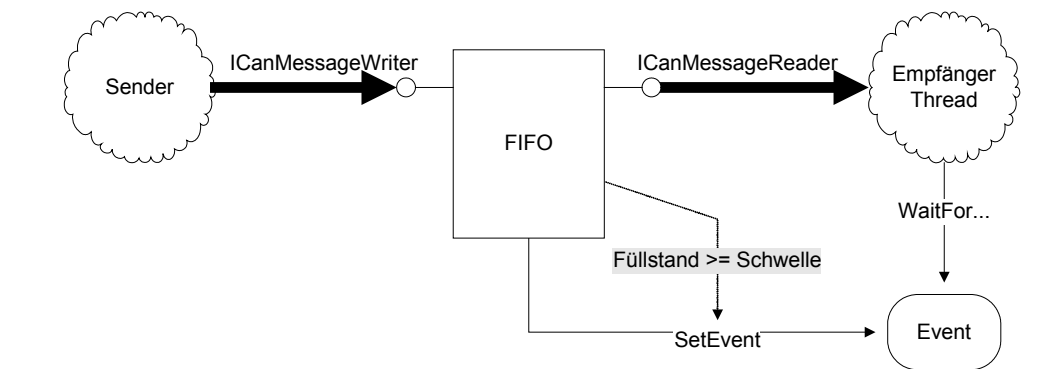


Bild 3-2: Funktionsweise von Empfangs-FIFOs

Empfangs-FIFOs werden über eine Reader-Schnittstelle (hier *ICanMessageReader*) angesprochen. Der Zugriff auf einzelne zu lesende Nachrichten erfolgt dabei über die Methode *GetMessage*. Methode *GetMessages* liefert gleich mehrere FIFO-Einträge über einen Aufruf.

Damit ein Empfänger nicht ständig überprüfen muss, ob neue Daten verfügbar sind, kann dem FIFO ein Event-Objekt zugeordnet werden, das immer dann in den signalisierten Zustand gesetzt wird, wenn ein gewisser Füllstand erreicht ist.

Hierzu wird ein *AutoResetEvent* oder ein *ManualResetEvent* erzeugt und anschließend mit der Methode *AssignEvent* an den FIFO übergeben. Die Schwelle, bzw. der Füllstand bei dem das Event ausgelöst wird, kann mit dem Property *Threshold* eingestellt werden.

Die Applikation kann mit *WaitOne* oder *WaitAll* des Eventobjekts auf das Eintreffen des Events warten. Anschließend können die Daten aus dem FIFO gelesen werden. Folgendes Sequenzdiagramm zeigt den zeitlichen Ablauf beim ereignisgesteuerten Lesen von Daten aus dem FIFO.

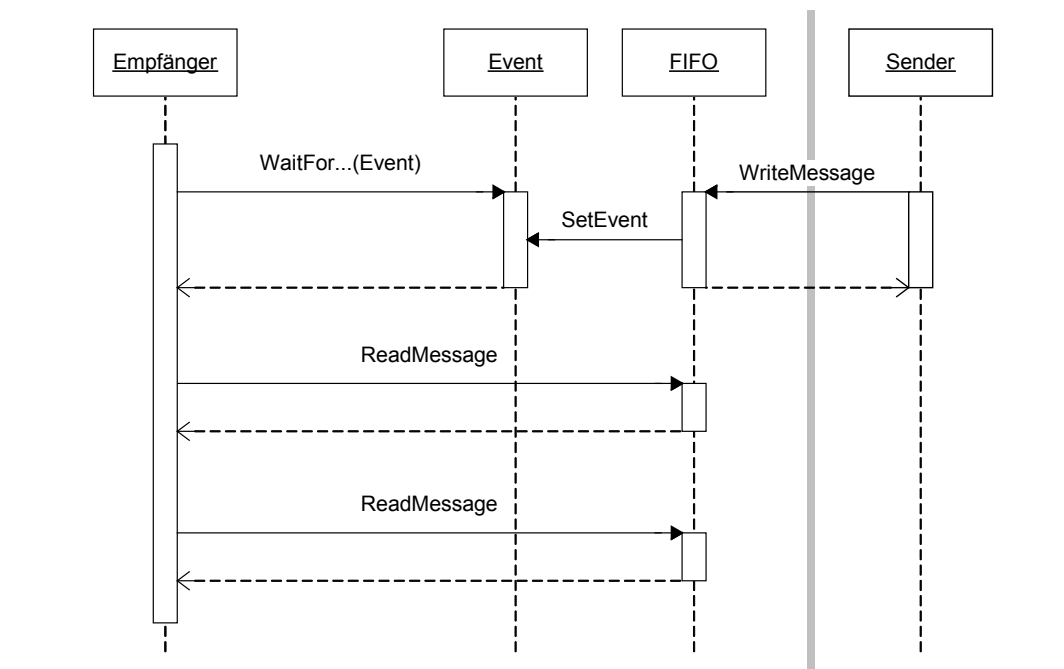


Bild 3-3: Empfangssequenz

3.1.2 Funktionsweise von Sende-FIFOs

Folgende Abbildung zeigt die Funktionsweise von Sende-FIFOs. Die dicken Pfeile zeigen den Datenfluss vom Sender zum Empfänger an.

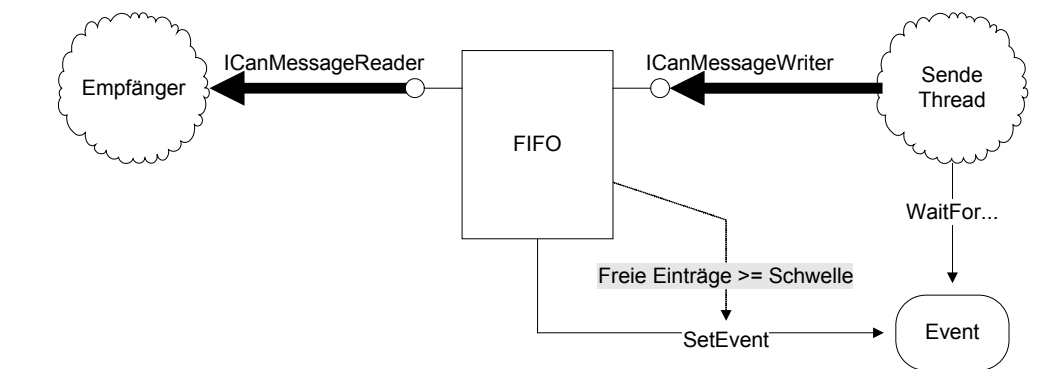


Bild 3-4: Funktionsweise von Sende – FIFOs

Sende-FIFOs werden über eine Writer-Schnittstelle (hier *ICanMessageWriter*) angesprochen. Einzelne zu sendende Nachrichten werden dabei über die Methode *WriteMessage* in den FIFO eingetragen. Die Methode *WriteMessage* schreibt gleich mehrere Nachrichten in einem Aufruf in den FIFO.

Damit ein Sender nicht ständig überprüfen muss, ob freie Elemente verfügbar sind, kann dem FIFO ein Event-Objekt zugeordnet werden, das immer dann in den signalisierten Zustand gesetzt wird, wenn die Anzahl freier Elemente einen gewissen Wert überschreitet.

Hierzu wird ein *AutoResetEvent* oder ein *ManualResetEvent* erzeugt und anschließend mit der Methode *AssignEvent* an den FIFO übergeben. Die Schwelle, bzw. die Zahl freier Elemente bei dem das Event ausgelöst wird, kann mit dem Property *Threshold* eingestellt werden.

Die Applikation kann mit *WaitOne* oder *WaitAll* des Eventobjekts auf das Eintreffen des Events warten. Anschließend können Daten in den FIFO geschrieben werden. Folgendes Sequenzdiagramm zeigt den zeitlichen Ablauf beim ereignis-gesteuerten Schreiben von Daten in den FIFO.

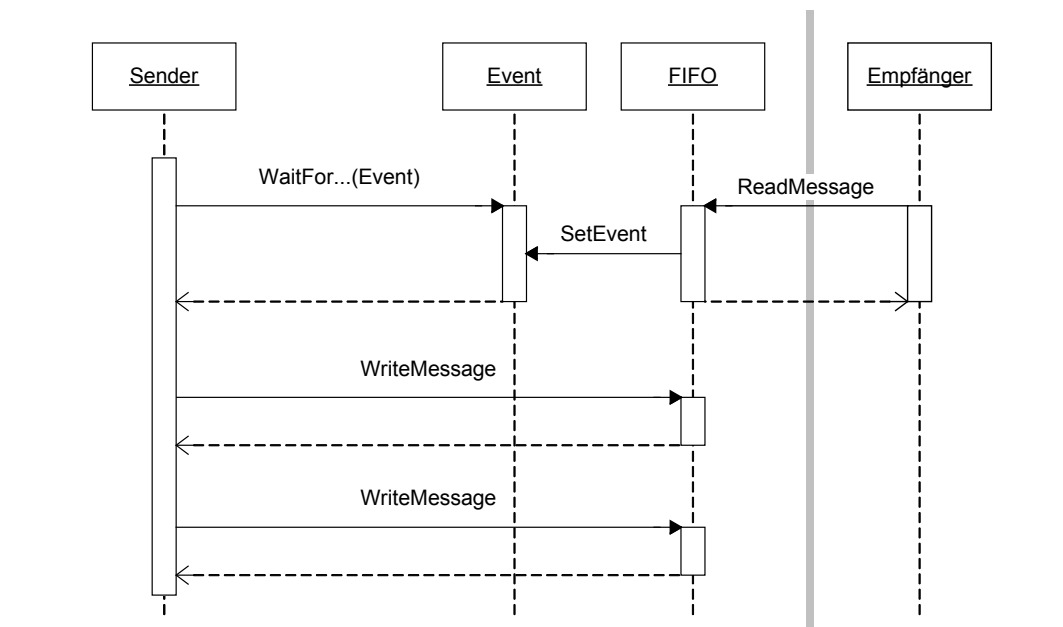


Bild 3-5: Sendesequenz

4 Zugriff auf den Feldbus

4.1 Übersicht

Der Zugriff auf die mit der CAN-Interfacekarte verbundenen Feldbusse erfolgt über den Bus Access Layer (BAL). Folgende Abbildung zeigt alle für den Zugriff notwendigen bzw. zur Verfügung gestellten Komponenten sowie die Funktionen, um den BAL einer CAN-Interfacekarte zu öffnen.

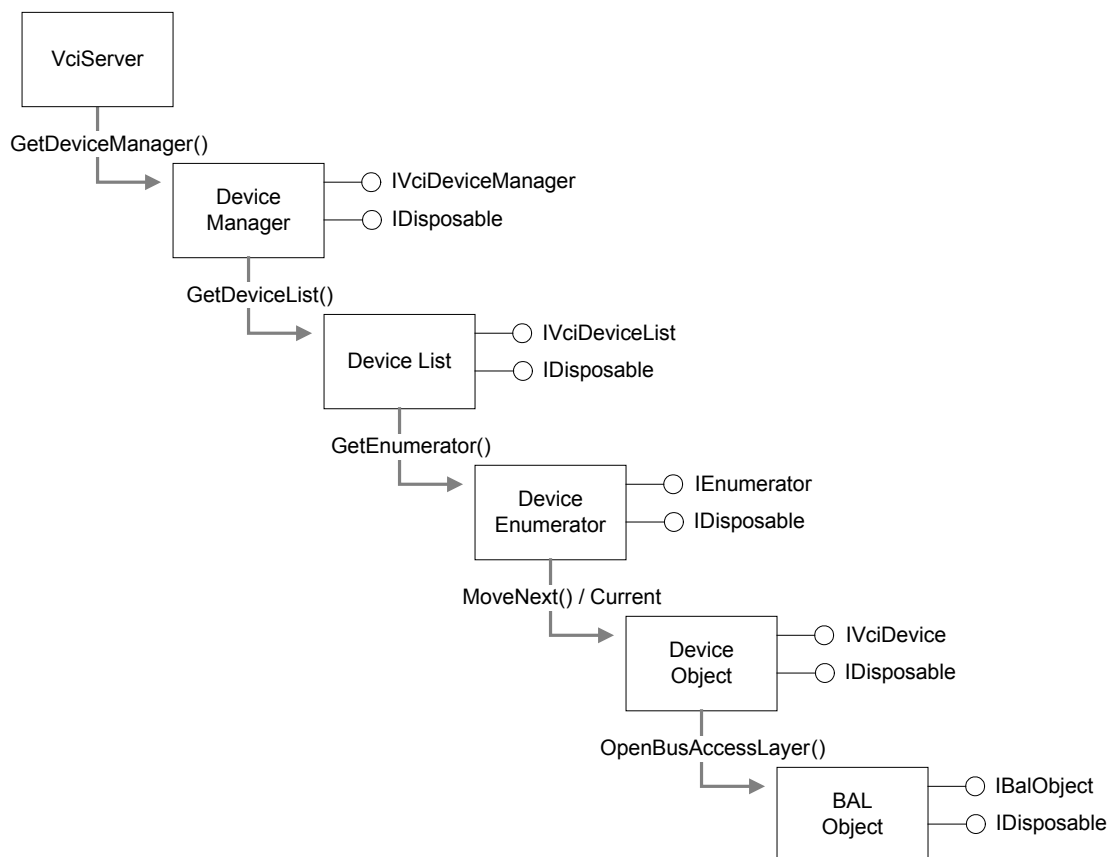


Bild 4-1: Komponenten für den Buszugriff

Im ersten Schritt wird die gewünschte CAN-Interfacekarte in der Geräteliste gesucht (siehe Kapitel 2.2). Danach wird der BAL durch Aufruf der Methode *IVciDevice.OpenBusAccessLayer* geöffnet.

Nachdem der BAL geöffnet ist, werden die Referenzen auf den Gerätemanager, die Geräteliste, den Geräteenumerator und das Geräteobjekt nicht mehr benötigt und können mittels der Methode *IDisposable.Dispose* freigegeben werden. Für die weitere Arbeit ist nur noch das BAL Objekt, bzw. die Schnittstelle *IBalObject* erforderlich.

Der BAL einer CAN-Interfacekarte kann von mehreren Programmen gleichzeitig geöffnet werden. Außerdem unterstützt er prinzipiell mehrere, auch unterschiedliche Arten von Busanschlüssen. Nachfolgende Abbildung zeigt eine CAN-Interfacekarte mit zwei Anschlüssen.

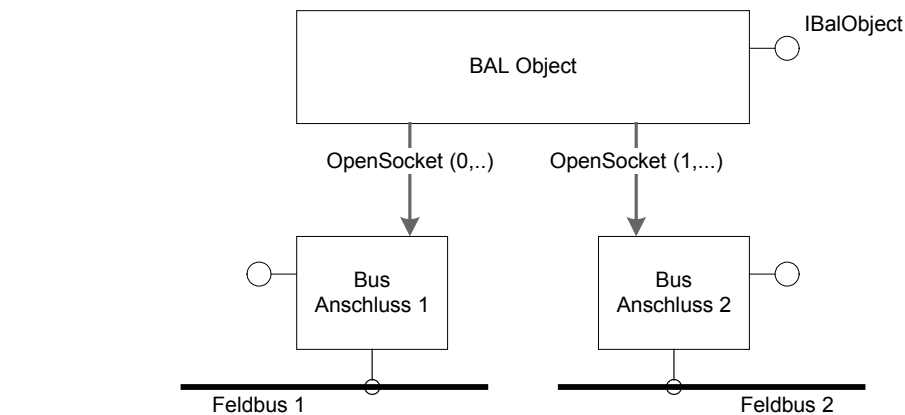


Bild 4-2: BAL mit zwei Busanschlüssen

Die Anzahl und die Art der zur Verfügung gestellten Anschlüsse lässt sich mittels Property *IBalObject.Resources* feststellen. Die Informationen werden vom Property in Form einer *BalResourceCollection* geliefert, welche für jeden vorhandenen Busanschluss ein BAL Resourceobjekt enthält.

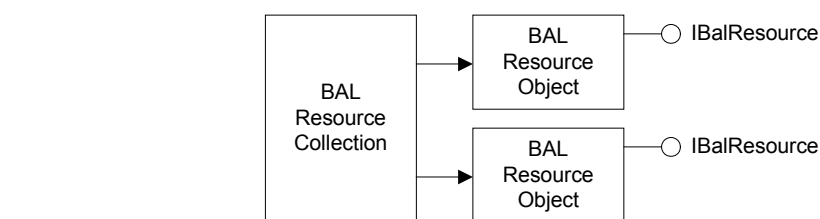


Bild 4-3: BalResourceCollection mit zwei Busanschlüssen.

Die Versionsnummer der Geräte-Firmware stellt der BAL über das Property *IBalObject.FirmwareVersion* bereit.

Zugriff auf einen Anschluss, bzw. auf eine Schnittstelle des Anschlusses erhält man mit der Methode *IBalObject.OpenSocket*. Die Methode erwartet im ersten Parameter die Nummer des zu öffnenden Anschlusses, wobei der Wert im Bereich 0 bis *IBalObject.Resources.Count-1* liegen muss. Zum Öffnen von Anschluss 1 wird der Wert 0, für Anschluss 2 der Wert 1, usw. angegeben. Im zweiten Parameter erwartet die Methode den Typ der Schnittstelle über die auf den Anschluss zugegriffen werden soll. Bei erfolgreicher Ausführung liefert die Methode eine Referenz auf die gewünschte Schnittstelle zurück.

Welche Möglichkeiten bzw. welche Schnittstellen ein Anschluss bietet hängt vom unterstützten Feldbus ab. Abschließend sei noch erwähnt, dass auf bestimmte Schnittstellen eines Anschlusses jeweils nur ein einziges Programm zugreifen kann, wohingegen auf andere beliebig viele Programme gleichzeitig zugreifen können. Die Regeln für den Zugriff auf die einzelnen Schnittstellen hängen ebenfalls von der Art des Anschlusses ab und werden in den folgenden Kapiteln genauer beschrieben.

4.2 CAN Anschluss

4.2.1 Übersicht

Jeder CAN-Anschluss setzt sich aus den in der folgenden Abbildung gezeigten Teilkomponenten zusammen.

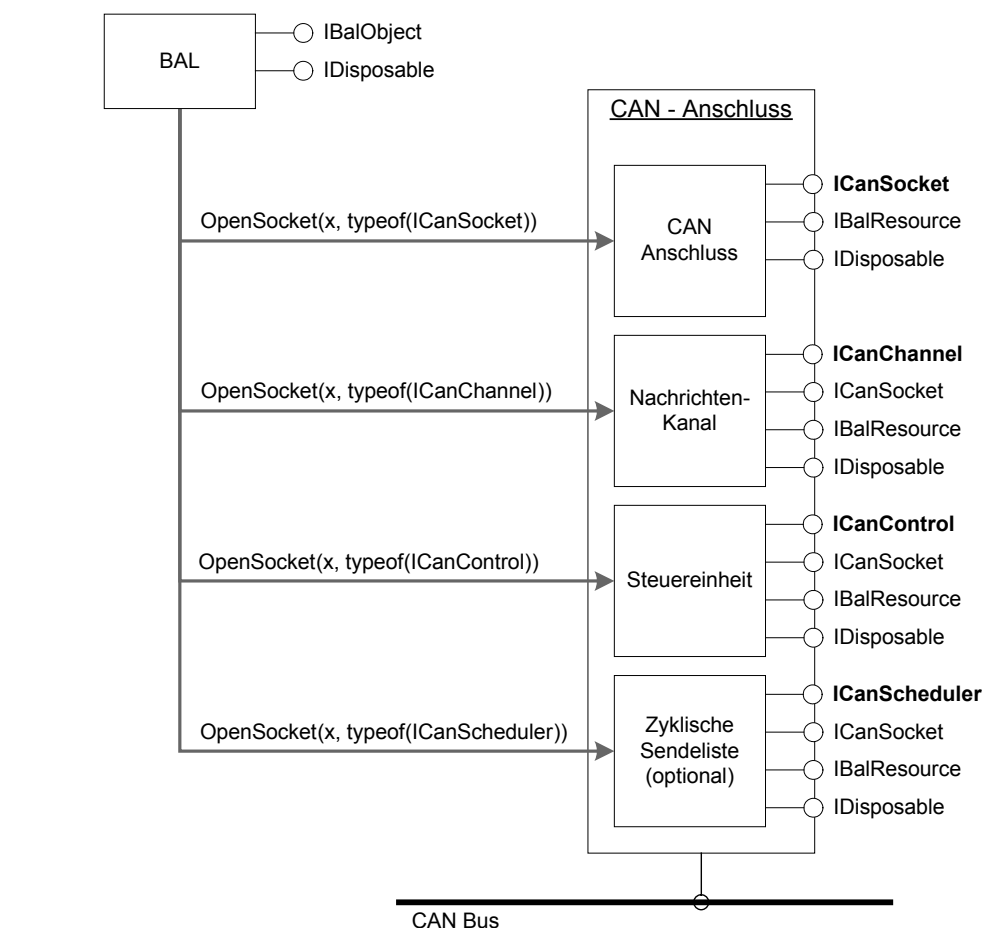


Bild 4-4: Komponenten eines CAN-Anschlusses

Der Zugriff auf die einzelnen Teilkomponenten eines CAN-Anschlusses findet über die Schnittstellen *ICanSocket*, *ICanChannel*, oder *ICanControl* statt. Die optionale zyklische Sendeliste mit der Schnittstelle *ICanScheduler* ist normalerweise nur bei CAN-Interfacekarten verfügbar, welche über einen eigenen Mikroprozessor verfügen.

Die Schnittstelle *ICanSocket* stellt Funktionen zur Abfrage der Eigenschaften vom CAN-Controller, sowie des aktuellen Controllerzustandes bereit.

Die Schnittstelle *ICanChannel* repräsentiert einen Nachrichtenkanal. Es lassen sich ein oder mehrere Nachrichtenkanäle für den selben CAN-Anschluss einrichten. Das Versenden und der Empfang von CAN-Nachrichten erfolgt ausschließlich über diese Nachrichtenkanäle.

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

Mit der optional vorhandenen zyklischen Sendeliste können über die Schnittstelle *ICanScheduler* pro Anschluss bis zu 16 Nachrichtenobjekte zyklisch, d. h. wiederkehrend in bestimmten Zeitintervallen gesendet werden.

Zugang zu den einzelnen Komponenten erhält man, wie bereits in Kapitel 4.1 beschrieben, über die Methode *IBalObject.OpenSocket*. Bild 4-4 zeigt die dabei zu verwendenden Schnittstellentypen.

4.2.2 Socket-Schnittstelle

Die Socket-Schnittstelle lässt sich mittels der Methode *IBalObject.OpenSocket* öffnen. Im Parameter *socketType* ist dabei der Typ *ICanSocket* anzugeben. Die Schnittstelle unterliegt keinerlei Zugriffsbeschränkungen und kann beliebig oft und von verschiedenen Programmen gleichzeitig geöffnet werden.

Die Schnittstelle *ICanSocket* stellt Funktionen zur Abfrage der Eigenschaften vom CAN-Controller, sowie des aktuellen Controllerzustandes bereit. Die Steuerung des Anschlusses ist jedoch nicht möglich.

Die Eigenschaften eines CAN-Anschlusses, wie der Typ des CAN-Controllers, die Art der Busankopplung und die unterstützten Features werden über zahlreiche Properties bereitgestellt.

Die aktuelle Betriebsart sowie der momentane Zustand vom CAN-Controller lässt über Property *LineStatus* ermitteln.

4.2.3 Nachrichtenkanäle

Erzeugt, bzw. geöffnet wird ein Nachrichtenkanal mittels der Methode *IBalObject.OpenSocket*. Im Parameter *socketType* ist dabei der Typ *ICanChannel* anzugeben. Jeder Nachrichtenkanal muss vor seiner Verwendung über die Methode *ICanChannel.Initialize* initialisiert werden. Der Parameter *exclusive* bestimmt dabei, ob der Anschluss exklusiv verwendet werden soll. Wird hier der Wert *true* angegeben, können nach erfolgreicher Ausführung der Methode keine weiteren Kanäle mehr geöffnet werden. Wird der CAN-Anschluss nicht exklusiv verwendet, lassen sich prinzipiell beliebig viele Nachrichtenkanäle einrichten.

Ein Nachrichtenkanal besteht aus jeweils einem Empfangs- und einem Sende-FIFO wie sie in Kapitel 3.1 beschrieben sind.

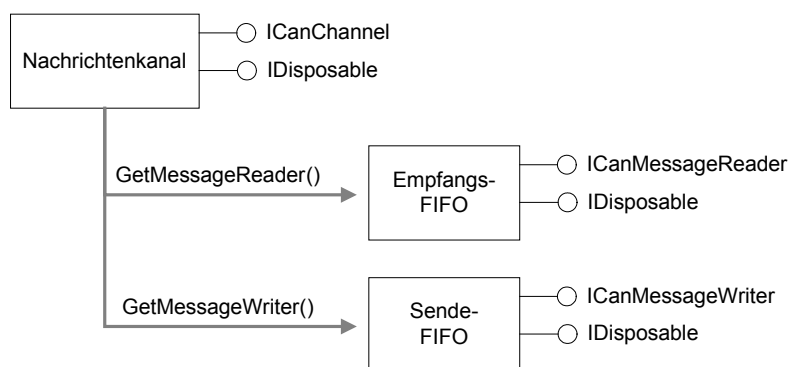


Bild 4-5: CAN-Nachrichtenkanal

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

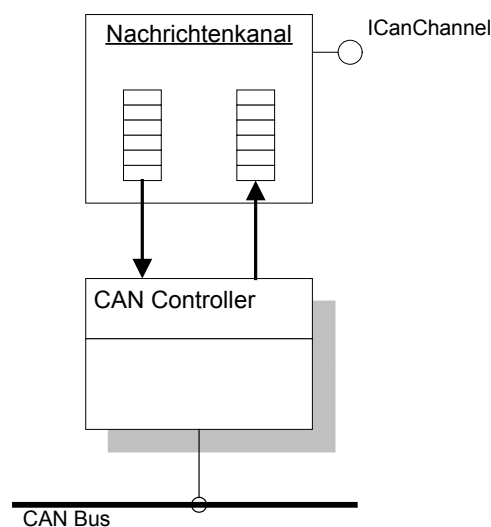


Bild 4-6: Exklusive Verwendung eines CAN-Nachrichtenkanals

Bei nicht exklusiver Verwendung des Anschlusses (*exclusive* = 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.

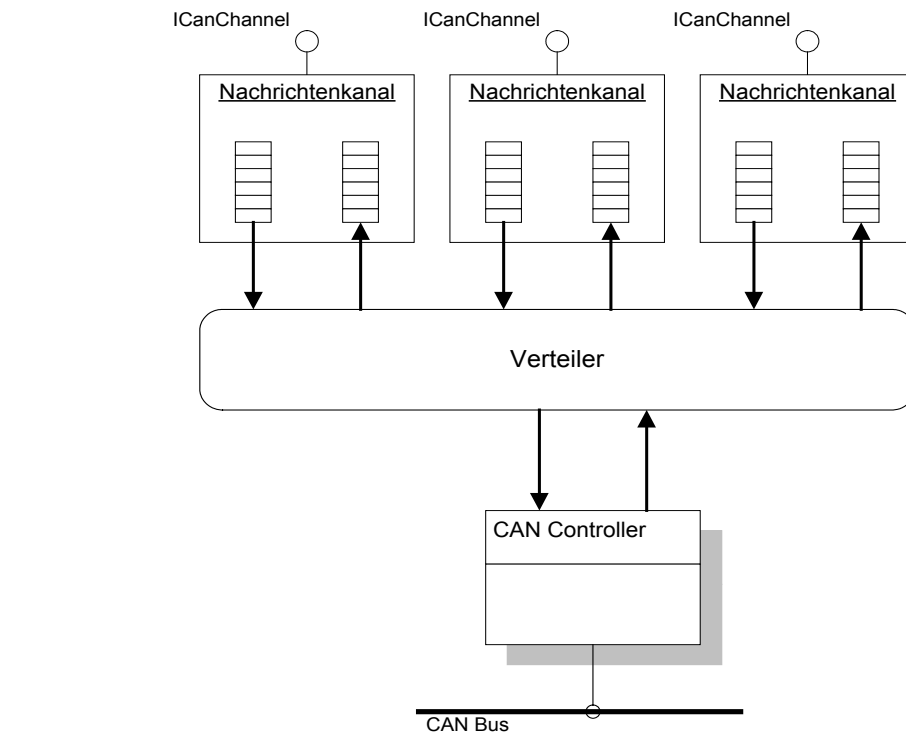


Bild 4-7: CAN-Nachrichtenverteiler

Ein neu erzeugter Nachrichtenkanal besitzt zunächst keinen Empfangs- und Sende-FIFO. Diese müssen erst durch einen Aufruf der Methode *ICanChannel.Initialize* erzeugt werden. Als Eingabeparameter erwartet die Methode die Größe des jeweiligen FIFOs in Anzahl CAN-Nachrichten.

Ist der Kanal eingerichtet, kann er mittels der Methode *ICanChannel.Activate* aktiviert und mittels der Methode *ICanChannel.Deactivate* wieder deaktiviert werden. Standardmäßig ist ein Nachrichtenkanal nach dem Öffnen deaktiviert.

Nachrichten werden nur dann vom Bus empfangen, bzw. an diesen gesendet, wenn der Kanal aktiv und der CAN-Controller gestartet ist. Einfluss auf die empfangenen Nachrichten hat auch der Nachrichtenfilter vom CAN-Controller. Weitere Informationen zum CAN-Controller finden sich in Kapitel 4.2.4.

4.2.3.1 Empfang von CAN-Nachrichten

Die empfangenen und vom Filter akzeptierten Nachrichten werden in den Empfangs-FIFO eines Nachrichtenkanals eingetragen. Zum Lesen der Nachrichten aus dem FIFO ist die Schnittstelle *ICanMessageReader* erforderlich. Diese kann mittels der Methode *ICanChannel.GetMessageReader* angefordert werden.

Die einfachste Art empfangene Nachrichten aus dem Empfangs-FIFO zu lesen ist ein Aufruf der Methode *ReadMessage*. Folgendes Codefragment zeigt eine mögliche Verwendung der Methode.

```
void DoMessages( ICanMessageReader reader )
{
    CanMessage message;
    while( reader.ReadMessage(out message) )
    {
        // Verarbeitung der Nachricht
    }
}
```

Eine weitere, mehr auf Datendurchsatz optimierte Möglichkeit Nachrichten aus dem Empfangs-FIFO zu lesen besteht in der Verwendung der Methode *ReadMessages*. Die Methode wird verwendet um mehrere CAN-Nachrichten über einen Methodenaufruf auszulesen. Der Benutzer legt ein Feld von CAN-Nachrichten an und übergibt dieses der Methode *ReadMessages*, welche versucht dieses mit empfangenen Nachrichten zu füllen. Die Anzahl tatsächlich gelesener Nachrichten signalisiert die Methode über ihren Rückgabewert.

Folgendes Codefragment zeigt eine mögliche Verwendung der Funktionen.

```
void DoMessages( ICanMessageReader reader )
{
    CanMessage[] messages = new CanMessage[10];
    int readCount = reader.ReadMessages(messages);
    for( int i = 0; i < readCount; i++ )
    {
        // Verarbeitung der Nachricht
    }
}
```

Eine ausführliche Beschreibung der FIFOs findet sich in Kapitel 3.1. Die Funktionsweise von Empfangs-FIFOs kann in Kapitel 3.1.1 nachgelesen werden.

4.2.3.2 Senden von CAN-Nachrichten

Zum Senden von Nachrichten wird die Schnittstelle *ICanMessageWriter* des Sende-FIFOs benötigt.

Diese kann mittels der Methode *ICanChannel.GetMessageWriter* vom Nachrichtenkanal angefordert werden.

Die einfachste Art eine Nachricht zu senden besteht in einem Aufruf der Methode *SendMessage*. Hierbei muss der Methode im Parameter *message* die zu sendende Nachricht vom Typ *CanMessage* übergeben werden. Folgendes Codefragment zeigt eine mögliche Verwendung der Methode.

```
bool SendByte( ICanMessageWriter writer, UInt32 id, Byte data )
{
    CanMessage message = new CanMessage();

    // CAN Nachricht initialisieren.
    message.TimeStamp      = 0;      // kein verzögertes Senden
    message.Identifizier   = id;     // Nachrichten ID (CAN-ID)
    message.FrameType      = CanMsgFrameType.Data;
    message.SelfReceptionRequest = false; // kein Self-Reception
    message.ExtendedFrameFormat = false; // Standard Frame
    message.DataLength     = 1;      // nur 1 Datenbyte
    message[0]             = data;

    // Nachricht senden
    return writer.SendMessage(message);
}
```

Beachten Sie, dass nur Nachrichten vom Typ *CanMsgFrameType.Data* versendet werden können. Andere Nachrichtentypen sind nicht erlaubt, bzw. werden vom CAN-Controller stillschweigend verworfen.

Wird in *TimeStamp* ein Wert ungleich 0 angegeben, wird die Nachricht verzögert auf den Bus gesendet. Weitere Informationen zum verzögerten Senden finden Sie in Kapitel 4.2.3.3.

Eine andere Möglichkeit Nachrichten zu senden besteht in der Verwendung der Methode *WriteMessages*. Über diese Methode kann eine vordefinierten Abfolge von Nachrichten über einen Methodenaufruf gesendet werden. Im Parameter *messages* wird die Nachrichtenabfolge als Feld von CAN-Nachrichten übergeben. Der Methodenrückgabewert quittiert die Anzahl Nachrichten, die tatsächlich in den Sende-FIFO eingetragen werden konnten.

Folgendes Codefragment zeigt eine mögliche Verwendung der Funktionen.

```
bool Send( ICanMessageWriter writer)
{
    CanMessage[] messages = new CanMessage[3];

    // CAN Nachrichten initialisieren.
    message[0].Identifier = 0x100;
    message[0].DataLength = 0;

    message[1].Identifier = 0x200;
    message[1].DataLength = 3;
    message[1].RemoteTransmissionRequest = true;

    message[2].Identifier = 0x300;
    message[2].DataLength = 2;
    message[2][0]         = 0x01;
    message[2][1]         = 0xAF;

    return ( message.Length == writer.SendMessages(messages) );
}
```

Eine ausführliche Beschreibung der FIFOs findet sich in Kapitel 3.1. Die Funktionsweise von Sende-FIFOs kann in Kapitel 3.1.2 nachgelesen werden.

4.2.3.3 Verzögertes Senden von CAN-Nachrichten

Anschlüsse bei denen das Flag *ICanSocket.SupportsDelayedTransmission* gesetzt ist unterstützen das verzögerte Senden von CAN-Nachrichten.

Durch ein verzögertes Senden von Nachrichten 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 *CanMessage.TimeStamp* 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 *ICanSocket.MaxDelayedTXTicks* entnehmen. Die Auflösung eines Ticks in Sekunden berechnet sich aus den Werten in den Feldern *ICanSocket.ClockFrequency* und *ICanSocket.DelayedTXTimerDivisor* nach folgender Formel:

$$\text{Auflösung [s]} = \text{DelayedTXTimerDivisor} / \text{ClockFrequency}$$

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

4.2.4 Steuereinheit

Die Steuereinheit, bzw. die Schnittstelle *ICanControl* stellt Methoden zur Konfiguration des CAN-Controllers, dessen Übertragungseigenschaften sowie Funktionen zum Konfigurieren von CAN-Nachrichtenfiltern und 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 Schnittstelle durch unterschiedliche Programme ist nicht möglich. Dadurch lassen sich Situationen vermeiden, bei denen z. B. eine Applikation den CAN-Controller starten, eine andere aber stoppen möchte.

Geöffnet wird die Schnittstelle mittels der Methode *IBalObject.OpenSocket*. Im Parameter *socketType* ist dabei der Typ *ICanControl* anzugeben. Endet der Methodenaufruf in einer Exception, wird die Komponente bereits von einem anderen Programm verwendet.

Mittels der Methode *IDisposable.Dispose* kann eine geöffnete Steuereinheit geschlossen und damit für andere Applikationen freigegeben werden. Sind beim Schließen der Steuereinheit noch andere Schnittstellen des Anschlusses offen, bleiben die momentanen Controller-Einstellungen erhalten.

4.2.4.1 Kontrollerzustände

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

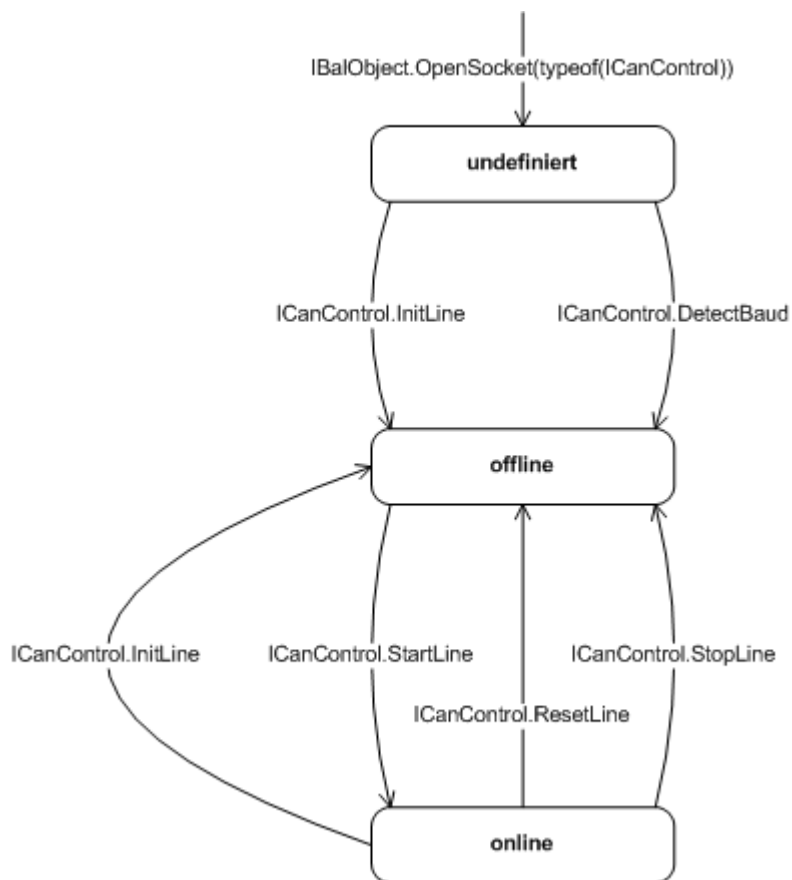


Bild 4-8: Kontrollerzustände

Nach dem Öffnen der Steuereinheit, bzw. der Schnittstelle *ICanControl* befindet sich der Controller normalerweise in einem undefinierten Zustand. Dieser Zustand wird durch Aufruf der Methode *InitLine* oder *DetectBaud* verlassen. Danach befindet sich der Controller im Zustand „offline“.

Mittels *InitLine* wird die Betriebsart und Bitrate vom CAN Controller eingestellt. Hierzu erwartet die Methode Werte für die Parameter *operatingMode* und *bitrate*.

Die Bitrate wird in den Feldern *CanBitrate.Btr0* und *CanBitrate.Btr1* angegeben. Die Werte entsprechen dabei den Werten für die Register BTR0 und BTR1 vom Philips SJA 1000 CAN-Controller bei einer Taktfrequenz von 16 MHz. Weitere Informationen hierzu finden sich im Datenblatt zum SJA 1000 in Kapitel 6.5. Eine Zusammenstellung der Bus Timing Werte mit allen CiA bzw. CANopen konformen Bitraten findet sich in folgender Tabelle.

Bitrate (KBit)	Vordefinierte CiA Bitraten	BTR0	BTR1
10	<i>CanBitrate.Cia10KBit</i>	0x31	0x1C
20	<i>CanBitrate.Cia20KBit</i>	0x18	0x1C
50	<i>CanBitrate.Cia50KBit</i>	0x09	0x1C
125	<i>CanBitrate.Cia125KBit</i>	0x03	0x1C
250	<i>CanBitrate.Cia250KBit</i>	0x01	0x1C
500	<i>CanBitrate.Cia500KBit</i>	0x00	0x1C
800	<i>CanBitrate.Cia800KBit</i>	0x00	0x16
1000	<i>CanBitrate.Cia1000KBit</i>	0x00	0x14
100	<i>CanBitrate.100KBit</i>	0x04	0x1C

Ist der CAN-Anschluss mit einem laufenden System verbunden bei dem die Bitrate unbekannt ist, kann die aktuelle Bitrate des Systems mit der Methode ***DetectBaud*** ermittelt werden. Die von der Methode ermittelten Bus Timing Werte können anschließend an die Methode ***InitLine*** übergeben werden.

Die Methode ***DetectBaud*** benötigt ein Feld mit vordefinierten Bus Timing Werten. Folgendes Beispiel zeigt die Verwendung der Methode zur automatischen Initialisierung eines CAN-Anschlusses an einem CANopen-System.

```
void AutoInitLine( ICanControl control )
{
    // Bitrate ermitteln
    int index = control.DetectBaud(10000, CanBitrate.CiaBitRates);

    if (-1 < index)
    {
        CanOperatingModes mode;
        mode = CanOperatingModes.Standard | CanOperatingModes.ErrFrame;
        control.InitLine(mode, CanBitrate.CiaBitRates[index]);
    }
}
```

Gestartet wird der CAN-Controller durch Aufruf der Methode ***StartLine***. Nach erfolgreicher Ausführung der Methode befindet sich der CAN-Controller im Zustand „online“. In diesem Zustand ist der CAN-Controller aktiv mit dem Bus verbunden. Eingehende CAN-Nachrichten werden hierbei an alle geöffneten und aktiven Nachrichtenkanäle weitergeleitet, bzw. Sendenachrichten von diesen an den Bus ausgegeben.

Die Methode ***StopLine*** schaltet den CAN-Controller wieder in den Zustand „off-line“. Dabei wird der Nachrichtentransport unterbrochen und der Controller deaktiviert. Ein Aufruf der Methode ändert die eingestellten Akzeptanzfilter und Filterlisten nicht. Auch bricht die Methode einen laufenden Sendevorgang des Controllers nicht einfach ab, sondern wartet bis die Nachricht vollständig auf den Bus übertragen wurde.

Die Methoden *ResetLine* schaltet den CAN-Controller ebenfalls in den Zustand „offline“. Im Gegensatz zu *StopLine* setzt diese Methode die Controllerhardware zurück und löscht alle Nachrichtenfilter. Beachten Sie, dass das Zurücksetzen der Controllerhardware zu fehlerhaften Nachrichtentelegrammen auf dem Bus führt, wenn bei Aufruf der Methode ein Sendevorgang mitten in der Übertragung abgebrochen wird.

Die Methoden *ResetLine* und *StopLine* löschen nicht den Inhalt der Sende- und Empfangs-FIFOs der Nachrichtenkanäle.

4.2.4.2 Nachrichtenfilter

Jede Steuereinheit verfügt über einen zweistufigen 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.

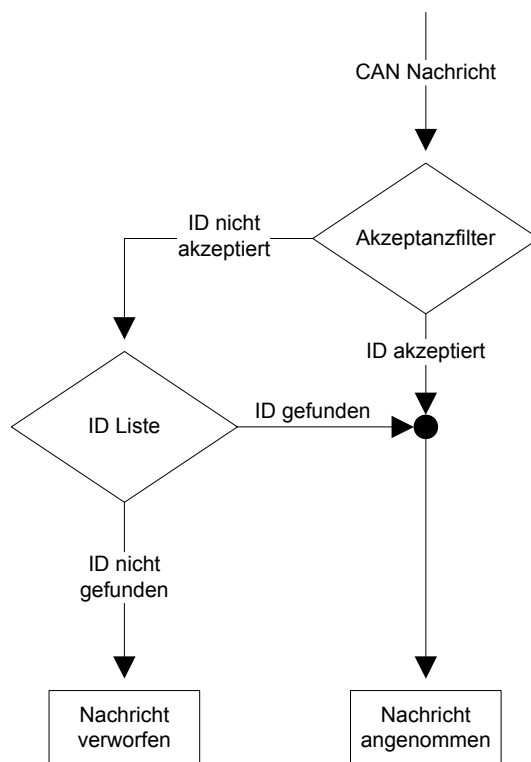


Bild 4-9: Filtermechanismus

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 Methoden *SetAccFilter*, *AddFilterIds*, und *RemFilterIds* ändern. Als Eingabewerte erwarten die Funktionen in den Parametern *code* und *mask* 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 *code* und *mask* bestimmen welche IDs vom Filter durchgelassen werden. Der Wert von *code* bestimmt dabei das Bitmuster der ID, wohingegen *mask* festlegt welche Bits in *code* für den Vergleich herangezogen werden. Hat ein Bit in *mask* den Wert 0, wird das entsprechende Bit in *code* 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 Bits relevant. Beim 29-Bit-Filter werden die Bits 0 bis 29 verwendet. Alle anderen Bits sollten vor Aufruf der Methode auf 0 gesetzt werden.

Nachfolgende Tabellen zeigen den Zusammenhang zwischen den Bits in den Parametern *code* und *mask*, 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 Request Bit (RTR) einer Nachricht.

Folgendes Beispiel zeigt die Werte für die Parameter *code* und *mask*, um nur die Nachrichten im Bereich 100h bis 103h zu akzeptieren, bei denen das RTR- Bit 0 gleichzeitig ist:

<i>code</i> :	001 0000 0000 0
<i>mask</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 Methode **AddFilterIds** können einzelne oder Gruppen von IDs in die Liste eingetragen und mittels der Methode **RemFilterIds** wieder aus der Liste entfernt werden. Die Parameter *code* und *mask* haben dabei das gleiche Format wie beim Akzeptanzfilter.

Wird die Methode **AddFilterIds** z. B. mit den Werten aus dem vorherigen Beispiel aufgerufen, trägt die Methode die IDs 100h bis 103h in die Liste ein. Soll bei einem Aufruf der Methode nur eine einzige ID eingetragen werden, gibt man in *code* die gewünschte ID (einschließlich RTR- Bit) an und setzt *mask* auf den Wert FFFh bzw. 3FFFFFFh.

Der Akzeptanzfilter kann durch einen Aufruf der Methode **SetAccFilter** vollständig gesperrt werden, wenn für *code* der Wert **CanAccCode.None** und für *mask* der Wert **CanAccMask.None** angegeben wird. Die weitere Filterung erfolgt anschließend nur noch anhand der ID-Liste. Ein Aufruf der Methode mit den Werten **CanAccCode.All** und **CanAccMask.All** hingegen öffnet den Akzeptanzfilter vollständig. Die ID-Liste ist in diesem Fall also wirkungslos.

4.2.5 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 CAN-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 Schnittstelle mittels der Methode *IBalObject.OpenSocket*. Im Parameter *socketType* muss dabei der Type *ICanScheduler* angegeben werden. Endet die Methode in einer *VciException* wird die Sendeliste bereits von einem anderen Programm verwendet. Falls der CAN-Anschluss keine zyklische Sendeliste unterstützt, wirft *IBalObject.OpenSocket* eine *NotImplementedException*. Mittels der Methode *IDisposable.Dispose* wird eine geöffnete Sendeliste geschlossen und für andere Applikationen freigegeben.

Mit der Methode *ICanScheduler.AddMessage* wird ein Nachrichtenobjekt der Liste hinzugefügt. Die Methode erwartet eine Referenz auf ein *CanCyclicTXMsg* Objekt, welches das Nachrichtenobjekt spezifiziert, das der Liste hinzugefügt werden soll.

Die Zykluszeit eines Sendeobjekts wird in Anzahl Ticks im Feld *CanCyclicTXMsg.CycleTicks* angegeben. Der Wert in diesem Feld muss größer 0 sein und darf den Wert im Feld *ICanSocket.MaxCyclicMsgTicks* nicht überschreiten.

Die Dauer eines Ticks, bzw. die Zykluszeit t_z der Sendeliste kann mittels der Felder *ICanSocket.ClockFrequency* und *ICanSocket.CyclicMessageTimeDivisor* nach folgender Formel berechnet werden.

$$t_z [s] = (\text{CyclicMessageTimeDivisor} / \text{ClockFrequency})$$

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

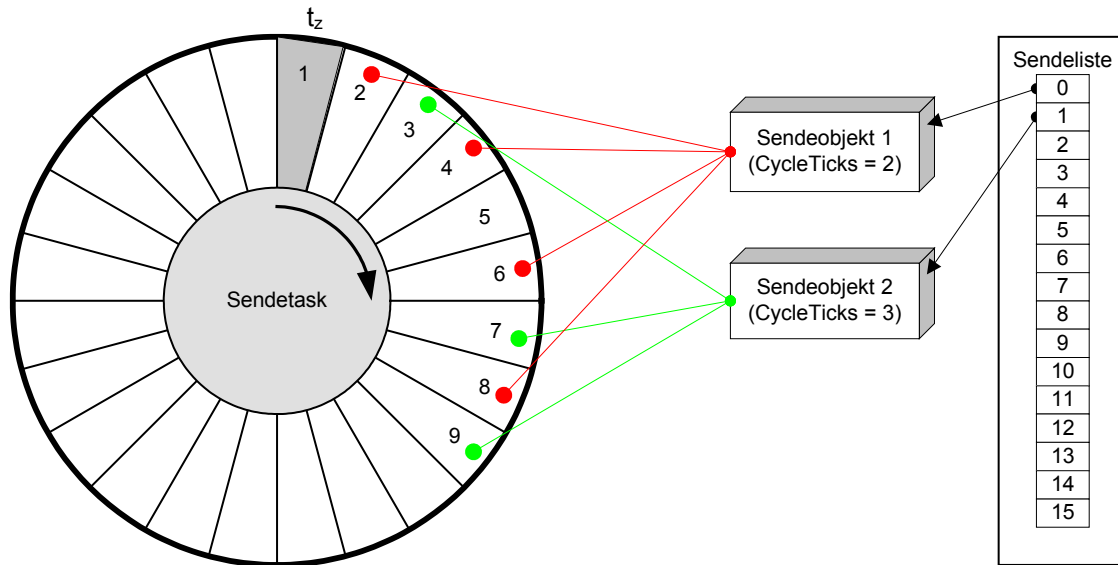


Bild 4-10: Sendetask der zyklischen Sendeliste

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 Zeitschlitz 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 $1/\text{CycleTime}$ 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 4-10 zeigt zwei Sendeobjekte mit den Zykluszeiten 2 und 3. Beim Einrichten von Sendeobjekt 1 werden die Zeitschlitz 2, 4, 6, 8, usw. belegt. Beim anschließenden Einrichten des zweiten Objekts (Zykluszeit = 3) kommt es in den Zeitschlitz 6, 12, 18, usw. zu Kollisionen, da diese Zeitschlitz 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 Zeitschlitz 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 *CanCyclicTXMsg.AutoIncrementMode* bestimmt, ob ein Teil der Nachricht nach jedem Sendevorgang automatisch inkrementiert wird. Wird hier der Wert *CanCyclicTXIncMode.NoInc* angegeben, bleibt der Inhalt unverändert. Beim Wert *CanCyclicTXIncMode.Incl1* wird das Feld *Identifier* der Nachricht nach jedem Sendevorgang automatisch um 1 erhöht. Erreicht das Feld *Identifier* den Wert 2048 (11-bit ID) bzw. 536.870.912 (29-bit ID) erfolgt automatisch ein Überlauf auf 0.

Beim Wert *CanCyclicTXIncMode.Inc8* oder *CanCyclicTXIncMode.Inc16* im Feld *CanCyclicTXMsg.AutoIncrementMode*, wird ein einzelner 8- oder 16-Bit Wert im Datenfeld der Nachricht inkrementiert. Das Feld *AutoIncrementIndex* legt dabei den Index des Datenfeldes fest. Bei 16-Bit Werten liegt das niederwertige Byte (LSB) im Datenfeld *Data[AutoIncrementIndex]* und das höherwertige Byte (MSB) im Feld *Data[AutoIncrementIndex + 1]*. Wird der Wert 255 (8-Bit) bzw. 65535 (16-Bit) erreicht, erfolgt ein Überlauf auf 0.

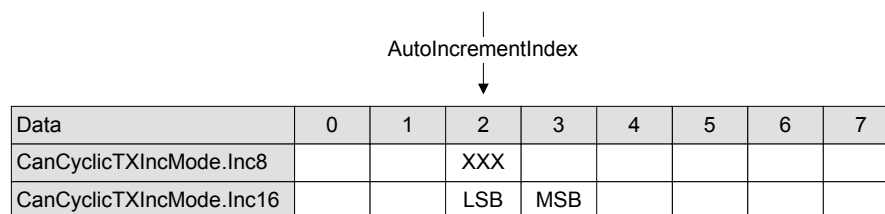


Bild 4-11: Auto-Inkrement von Datenfeldern

Mit der Methode *RemMessage* kann ein Sendeobjekt wieder aus der Liste entfernt werden. Die Methode erwartet hierzu eine Referenz eines über Methode *AddMessage* hinzugefügten Nachrichtenobjekts.

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

Der Zustand eines einzelnen Sendeobjekts kann über dessen *Status* Property abgefragt werden. Die Aktualisierung der Sendeobjektstatis muss allerdings manuell über die Methode *UpdateStatus* der zugehörigen Sendeliste angestossen werden.

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 Methode *Resume*.

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

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

4.3 LIN Anschluss

4.3.1 Übersicht

Jeder LIN-Anschluss setzt sich aus den in der folgenden Abbildung gezeigten Teilkomponenten zusammen.

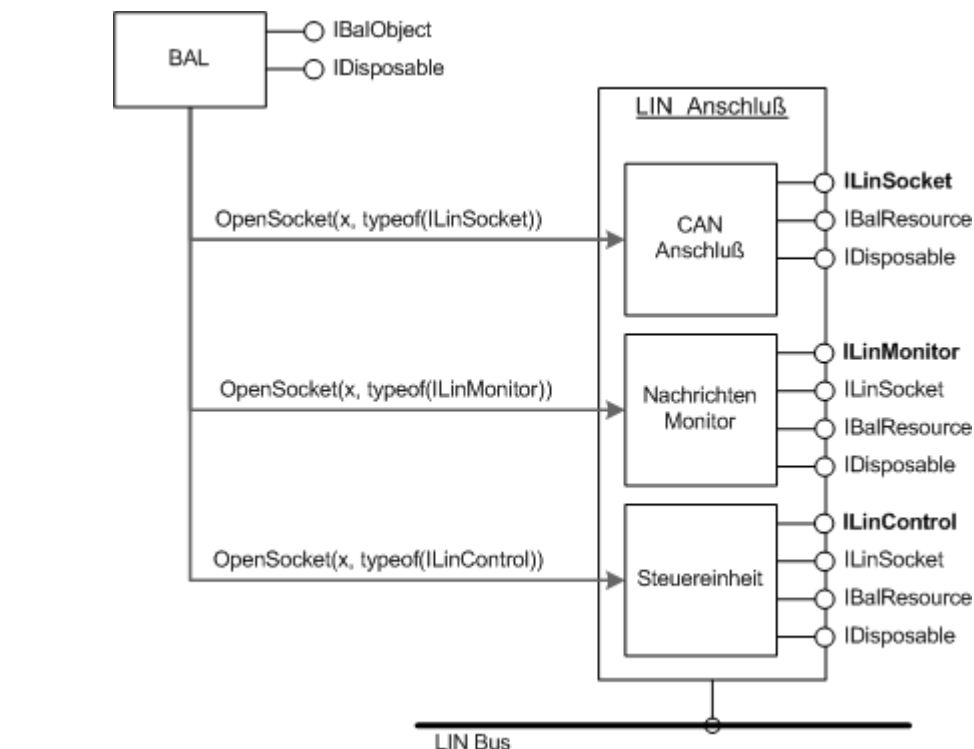


Bild 4-12: Komponenten eines LIN-Anschlusses

Der Zugriff auf die einzelnen Teilkomponenten eines LIN-Anschlusses findet über die Schnittstellen **ILinSocket**, **ILinMonitor**, oder **ILinControl** statt.

Die Schnittstelle **ILinSocket** stellt Funktionen zur Abfrage der Eigenschaften vom LIN-Controller, sowie des aktuellen Controllerzustandes bereit.

Die Schnittstelle **ILinMonitor** repräsentiert einen Nachrichtenmonitor. Es lassen sich ein oder mehrere Nachrichtenmonitore für denselben LIN-Anschluss einrichten. Der Empfang von LIN-Nachrichten erfolgt ausschließlich über diese Nachrichtenmonitore.

Die Steuereinheit, bzw. die Schnittstelle *ILinControl* stellt Funktionen zur Konfiguration des LIN-Controllers, dessen Übertragungseigenschaften sowie Funktionen zur Abfrage des aktuellen Controllerzustandes bereit.

Zugang zu den einzelnen Komponenten erhält man, wie bereits in Kapitel 4.1 beschrieben, über die Methode *IBalObject.OpenSocket*. Bild 4-12 zeigt die dabei zu verwendenden Schnittstellentypen.

4.3.2 Socket-Schnittstelle

Die Socket-Schnittstelle lässt sich mittels der Methode *IBalObject.OpenSocket* öffnen. Im Parameter *socketType* ist dabei der Typ *ILinSocket* anzugeben. Die Schnittstelle unterliegt keinerlei Zugriffsbeschränkungen und kann beliebig oft und von verschiedenen Programmen gleichzeitig geöffnet werden.

Die Schnittstelle *ILinSocket* stellt Funktionen zur Abfrage der Eigenschaften vom LIN-Controller, sowie des aktuellen Controllerzustandes bereit. Die Steuerung des Anschlusses ist jedoch nicht möglich.

Die Eigenschaften eines LIN-Anschlusses, wie die unterstützten Features werden über Properties bereitgestellt.

Die aktuelle Betriebsart sowie der momentane Zustand des LIN-Controllers lassen sich über das Property *LineStatus* ermitteln.

4.3.3 Nachrichtenmonitore

Erzeugt, bzw. geöffnet wird ein Nachrichtenmonitor mittels der Methode *IBalObject.OpenSocket*. Im Parameter *socketType* ist dabei der Typ *ILinMonitor* anzugeben. Jeder Nachrichtenkanal muss vor seiner Verwendung über die Methode *ILinMonitor.Initialize* initialisiert werden. Der Parameter *exclusive* bestimmt dabei, ob der Anschluss exklusiv verwendet werden soll. Wird hier der Wert *true* angegeben, können nach erfolgreicher Ausführung der Methode keine weiteren Monitore mehr geöffnet werden. Wird der LIN-Anschluss nicht exklusiv verwendet, lassen sich prinzipiell beliebig viele Nachrichtenmonitore einrichten.

Ein Nachrichtenmonitor besteht aus einem Empfangs-FIFO wie er in Kapitel 3.1 beispielhaft für CAN-Nachrichten beschrieben ist.

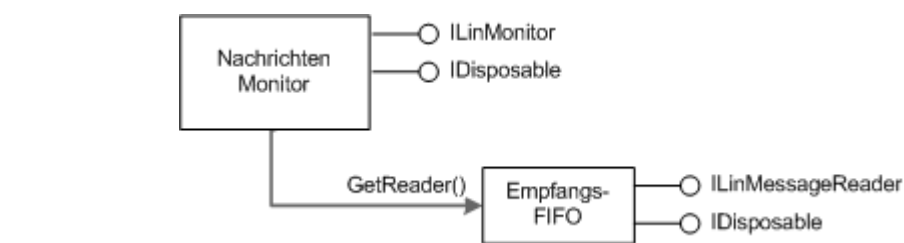


Bild 4-13: LIN-Nachrichtenmonitor

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

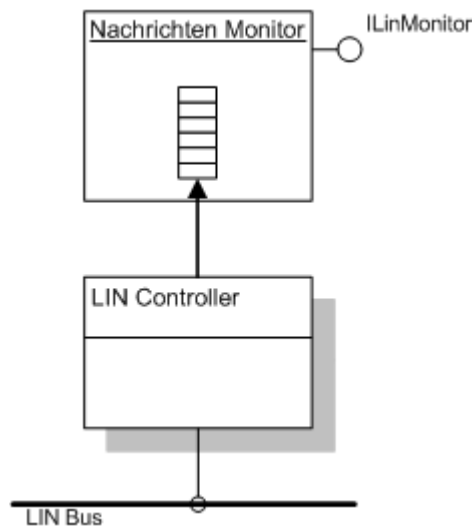


Bild 4-14: Exklusive Verwendung eines LIN-Nachrichtenmonitors

Bei nicht exklusiver Verwendung des Anschlusses (*exclusive* = false) wird ein Verteiler zwischen Controller und die Nachrichtenmonitore geschaltet. Der Verteiler leitet eingehende Nachrichten vom LIN-Controller an alle Nachrichtenmonitore 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.

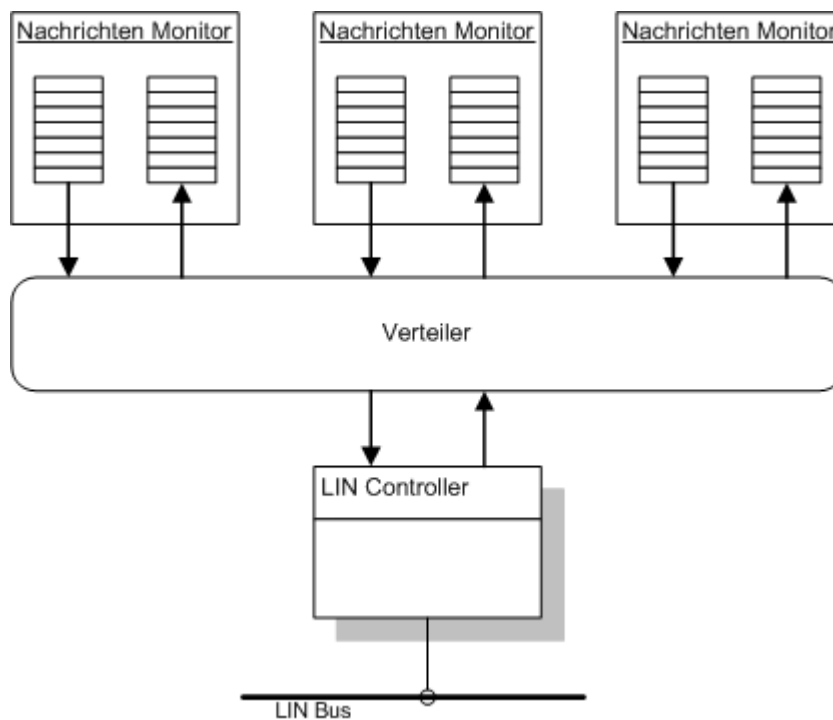


Bild 4-15: LIN-Nachrichtenverteiler

Ein neu erzeugter Nachrichtenkanal besitzt zunächst keinen Empfangs-FIFO. Dieser muss erst durch einen Aufruf der Methode *ILinMonitor.Initialize* erzeugt werden. Als Eingabeparameter erwartet die Methode die Größe des Empfangs-FIFOs in Anzahl LIN-Nachrichten.

Ist der Monitor eingerichtet, kann er mittels der Methode *ILinMonitor.Activate* aktiviert und mittels der Methode *ILinMonitor.Deactivate* wieder deaktiviert werden. Standardmäßig ist ein Nachrichtenmonitor nach dem Öffnen deaktiviert.

Nachrichten werden nur dann vom Bus empfangen, wenn der Monitor aktiv und der LIN-Controller gestartet ist. Weitere Informationen zum LIN-Controller finden sich in Kapitel 4.3.4.

4.3.3.1 Empfang von LIN-Nachrichten

Die empfangenen Nachrichten werden in den Empfangs-FIFO eines Nachrichtenmonitors eingetragen. Zum Lesen der Nachrichten aus dem FIFO ist die Schnittstelle *ILinMessageReader* erforderlich. Diese kann mittels der Methode *ILinMonitor.GetMessageReader* angefordert werden.

Die einfachste Art empfangene Nachrichten aus dem Empfangs-FIFO zu lesen ist ein Aufruf der Methode *ReadMessage*. Folgendes Codefragment zeigt eine mögliche Verwendung der Methode.

```
void DoMessages( ILinMessageReader reader )
{
    LinMessage message;
    while( reader.ReadMessage(out message) )
    {
        // Verarbeitung der Nachricht
    }
}
```

Eine weitere, mehr auf Datendurchsatz optimierte Möglichkeit Nachrichten aus dem Empfangs-FIFO zu lesen besteht in der Verwendung der Methode *ReadMessages*. Die Methode wird verwendet um mehrere LIN-Nachrichten über einen Methodenaufruf auszulesen. Der Benutzer legt ein Feld von LIN-Nachrichten an und übergibt dieses der Methode *ReadMessages*, welche versucht dieses mit empfangenen Nachrichten zu füllen. Die Anzahl tatsächlich gelesener Nachrichten signalisiert die Methode über ihren Rückgabewert.

Folgendes Codefragment zeigt eine mögliche Verwendung der Funktionen.

```
void DoMessages( ILinMessageReader reader )
{
    LinMessage[] messages = new LinMessage[10];
    int readCount = reader.ReadMessages(messages);
}
```

```
for( int i = 0; i < readCount; i++ )
{
    // Verarbeitung der Nachricht
}
```

Eine ausführliche Beschreibung der FIFOs findet sich in Kapitel 3.1. Die Funktionsweise von Empfangs-FIFOs kann in Kapitel 3.1.1 nachgelesen werden.

4.3.4 Steuereinheit

Die Steuereinheit, bzw. die Schnittstelle *ILinControl* stellt Methoden zur Konfiguration des LIN-Controllers, sowie dessen Übertragungseigenschaften und Properties 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 Schnittstelle 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 Schnittstelle mittels der Methode *IBalObject.OpenSocket*. Im Parameter *socketType* ist dabei der Typ *ILinControl* anzugeben. Endet der Methodenaufruf in einer Exception, wird die Komponente bereits von einem anderen Programm verwendet.

Mittels der Methode *IDisposable.Dispose* kann eine geöffnete Steuereinheit geschlossen und damit für andere Applikationen freigegeben werden. Sind beim Schließen der Steuereinheit noch andere Schnittstellen des Anschlusses offen, bleiben die momentanen Controller-Einstellungen erhalten.

4.3.4.1 Kontrollerzustände

Die folgende Abbildung zeigt die verschiedenen Zustände eines LIN-Controllers.

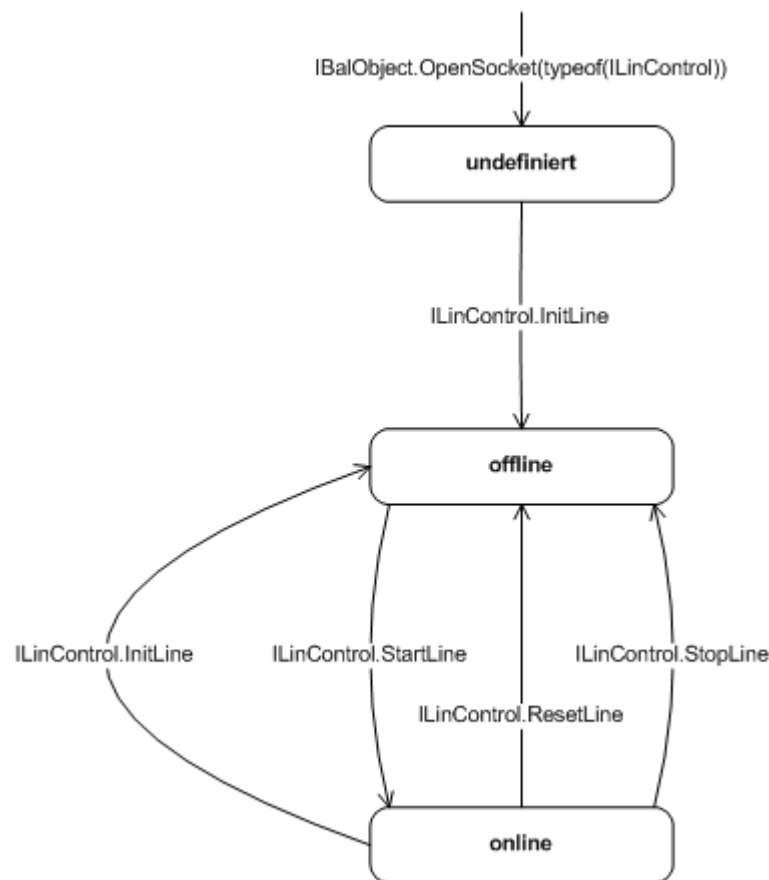


Bild 4-16: Kontrollerzustände

Nach dem Öffnen der Steuereinheit, bzw. der Schnittstelle *ILinControl* befindet sich der Controller normalerweise in einem undefinierten Zustand. Dieser Zustand wird durch Aufruf der Methode *InitLine* verlassen. Danach befindet sich der Controller im Zustand „offline“.

Mittels *InitLine* wird die Betriebsart und Bitrate des LIN Controllers eingestellt. Hierzu erwartet die Methode eine Struktur *LinInitLine* mit Werten für die Betriebsart und die Bitrate.

Die Übertragungsrate in Bit pro Sekunde wird im Feld *LinInitLine.Bitrate* angegeben. Gültige Werte für die Bitrate liegen zwischen 1000 und 20000, bzw. zwischen *LinBitrate.MinBitrate* und *LinBitrate.MaxBitrate*. Unterstützt der Anschluss die automatische Bitratenerkennung, kann diese durch Verwendung von *LinBitrate.AutoRate* aktiviert werden. Nachfolgende Tabelle zeigt einige empfohlene Bitraten:

Slow	Medium	Fast
<i>LinBitrate. Lin2400Bit</i>	<i>LinBitrate. Lin9600Bit</i>	<i>LinBitrate. Lin19200Bit</i>

Gestartet wird der LIN-Controller durch Aufruf der Methode ***StartLine***. Nach erfolgreicher Ausführung der Methode befindet sich der LIN-Controller im Zustand „online“. In diesem Zustand ist der LIN-Controller aktiv mit dem Bus verbunden. Eingehende LIN-Nachrichten werden hierbei an alle geöffneten und aktiven Nachrichtenmonitore weitergeleitet.

Die Methode ***StopLine*** schaltet den LIN-Controller wieder in den Zustand „offline“. Dabei wird der Nachrichtentransport unterbrochen und der Controller deaktiviert. Die Methode bricht einen laufenden Sendevorgang des Controllers nicht einfach ab, sondern wartet bis die Nachricht vollständig auf den Bus übertragen wurde.

Die Methoden ***ResetLine*** schaltet den LIN-Controller ebenfalls in den Zustand „offline“. Im Gegensatz zu ***StopLine*** setzt diese Methode die Controllerhardware zurück. Beachten Sie, dass das Zurücksetzen der Controllerhardware zu fehlerhaften Nachrichtentelegrammen auf dem Bus führt, wenn bei Aufruf der Methode ein Sendevorgang mitten in der Übertragung abgebrochen wird.

Die Methoden ***ResetLine*** und ***StopLine*** löschen nicht den Inhalt des Empfangs-FIFOs der Nachrichtenmonitore.

4.3.4.2 Senden von LIN-Nachrichten

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

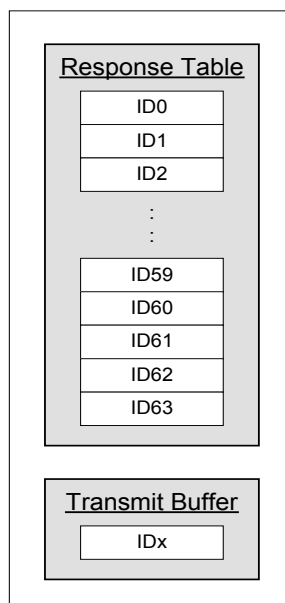


Bild 4-17: Interner Aufbau der Steuereinheit

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 Methode *ILinControl.WriteMessage* geändert, bzw. aktualisiert werden, indem im Parameter *send* der Wert *false* angegeben wird. Die Nachricht mit den Antwortdaten im Datenfeld der Struktur *LinMessage* wird der Funktion dabei im Parameter *message* übergeben. Beachten Sie, dass die Nachricht vom Typ *LinMessageType.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 Methode *ILinControl.ResetLine*.

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

Wird der Anschluss als Master betrieben, können neben den Steuernachrichten *LinMessageType.Sleep* und *LinMessageType.Wakeup* auch Datennachrichten vom Typ *LinMessageType.Data* direkt versendet werden.

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

Nachrichten vom Typ *LinMessageType.Sleep* erzeugen ein „Goto-Sleep“ Frame auf dem Bus, Nachrichten vom Typ *LinMessageType.Wakeup* dagegen einen WAKEUP Frame. Weitere Informationen hierzu finden sich in der LIN-Spezifikation im Kapitel „Network Management“.

In der Master-Betriebsart dient die Methode *ILinControl.WriteMessage* auch zum Aufschalten von IDs. Hierzu wird eine *LinMessageType.Data* Nachricht mit gültiger ID und Datenlänge gesendet, bei der das Flag *IdOnly* gleichzeitig den Wert *true* hat.

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

5 Schnittstellenbeschreibung

Eine detaillierte Beschreibung der VCI V3 .NET 2.0 Schnittstellen und Klassen finden Sie in der mitinstallierten Onlinereferenz **vcinet2.chm** im Unterverzeichnis **net2**.