



VCI: .NET-API

Software Version 4

SOFTWARE DESIGN GUIDE

4.02.0250.10021 1.1 DEUTSCH

Wichtige Benutzerinformation

Haftung

Dieses Dokument wurde mit größter Sorgfalt erstellt. Bitte informieren Sie HMS Industrial Networks über Ungenauigkeiten oder Versäumnisse. Die Daten und Illustrationen in diesem Dokument sind nicht verbindlich. Wir, HMS Industrial Networks, behalten uns das Recht vor, unsere Produkte gemäß unseres Grundsatzes der kontinuierlichen Produktentwicklung zu modifizieren. Die Informationen in diesem Dokument können ohne Vorankündigung geändert werden und sollten als nicht bindend für HMS Industrial Networks angesehen werden. HMS Industrial Networks übernimmt keine Verantwortung für mögliche Fehler in diesem Dokument.

Es gibt viele Anwendungsmöglichkeiten für dieses Produkt. Die für die Anwendung des Produkts Verantwortlichen müssen sicherstellen, dass alle notwendigen Schritte getroffen wurden, um sicherzustellen, dass die Anwendung alle Anforderungen bezüglich Durchführung und Sicherheit, einschließlich aller zutreffenden Gesetze, Vorschriften, Normen und Standards entspricht.

HMS Industrial Networks übernimmt in keinem Fall die Haftung oder Verantwortung für Probleme, die entstehen könnten, durch die Nutzung undokumentierter Funktionen, zeitlichen Ablauf, oder durch funktionelle Nebeneffekte außerhalb des dokumentierten Umfangs dieses Produkts. Die Effekte, verursacht durch jegliche direkte oder indirekte Verwendung solcher Aspekte des Produkts sind nicht definiert und könnten beispielsweise Kompatibilitätsprobleme und Stabilitätsprobleme beinhalten.

Die Beispiele und Illustrationen in diesem Dokument sind ausschließlich zum Zweck der Veranschaulichung enthalten. Aufgrund der vielen Variablen und Anforderungen, die mit jeder einzelnen Implementierung verbunden sind, kann HMS Industrial Networks keine Verantwortung übernehmen für die tatsächliche Verwendung basierend auf diesen Beispielen und Illustrationen.

Schutz- und Urheberrechte

HMS Industrial Networks besitzt die Schutz- und Urheberrechte für die Technologie, die in dem, in diesem Dokument beschriebenen, Produkt integriert ist. Diese Schutz- und Urheberrechte können Patente und schwebende Patentanmeldungen in den USA und anderen Ländern beinhalten.

Inhaltsverzeichnis

Seite

1 Benutzerführung	3
1.1 Mitgeltende Dokumente.....	3
1.2 Dokumenthistorie.....	3
1.3 Eingetragene Warenzeichen.....	3
1.4 Konventionen	3
1.5 Glossar	4
2 Systemübersicht	5
2.1 Komponenten des VCI .NET-Adapters.....	6
2.2 Legacy-Schnittstellen	7
2.2.1 VCI V3	7
2.2.2 VCI V2	7
2.3 Teilkomponenten und .NET Interfaces/Klassen	8
2.4 Programmierbeispiele	8
3 .NET API einbinden	9
3.1 Manuell in eigene Projekte einbinden	9
3.2 Via NuGet eine eigene Projekte einbinden	9
3.3 Applikationen portieren.....	9
4 Geräteverwaltung und Gerätezugriff	11
4.1 Verfügbare Geräte auflisten	12
4.2 Auf einzelne Geräte zugreifen.....	13
5 Kommunikationskomponenten.....	14
5.1 First-In/First-Out-Speicher (FIFO)	14
5.1.1 Funktionsweise Empfangs-FIFO	17
5.1.2 Funktionsweise Sende-FIFO.....	18
6 Auf Busanschlüsse zugreifen.....	20
6.1 CAN-Controller	22
6.1.1 Socket-Schnittstelle	23
6.1.2 Nachrichtenkanäle	23
6.1.3 Steuereinheit	30
Nachrichtenfilter	33
6.1.4 Zyklische Sendeliste	38
6.2 LIN-Anschluss	41
6.2.1 Socket-Schnittstelle	42
6.2.2 Nachrichtenmonitore	42
6.2.3 Steuereinheit	45

7 Schnittstellenbeschreibung	49
---	-----------

1 Benutzerführung

Bitte lesen Sie das Handbuch sorgfältig. Verwenden Sie das Produkt erst, wenn Sie das Handbuch verstanden haben.

1.1 Mitgeltende Dokumente

Dokument	Autor
VCI: C++ Software Version 4 Software Design Guide	HMS

1.2 Dokumenthistorie

Version	Datum	Beschreibung
1.0	Juli 2016	Erste Version
1.1	Januar 2018	Informationen zu Kapitel 3.2 Via NuGet in eigene Projekte einbinden und Pfad zu Beispielen hinzugefügt. Systemübersicht angepasst

1.3 Eingetragene Warenzeichen

IXXAT® ist ein registriertes Warenzeichen von HMS Industrial Networks. Alle anderen erwähnten Warenzeichen sind Eigentum der jeweiligen Inhaber.

1.4 Konventionen

Handlungsaufforderungen und Resultate sind wie folgt dargestellt:

- ▶ Handlungsaufforderung 1
- ▶ Handlungsaufforderung 2
 - ➔ Ergebnis 1
 - ➔ Ergebnis 2

Listen sind wie folgt dargestellt:

- Listenpunkt 1
- Listenpunkt 2

Fette Schriftart wird verwendet, um interaktive Teile darzustellen, wie Anschlüsse und Schalter der Hardware oder Menüs und Buttons in einer grafischen Benutzeroberfläche.

Diese Schriftart wird verwendet, um Programmcode und andere Arten von Dateninput und -output wie Konfigurationsskripte darzustellen.

Dies ist ein Querverweis innerhalb dieses Dokuments: [Konventionen, S. 3](#)

Dies ist ein externer Link (URL): www.hms-networks.com



Dies ist eine zusätzliche Information, die Installation oder Betrieb vereinfachen kann.



Diese Anweisung muss befolgt werden, um Gefahr reduzierter Funktionen und/oder Sachbeschädigung oder Netzwerk-Sicherheitsrisiken zu vermeiden.

1.5 Glossar

Abkürzungen

VCI	Virtual Communication Interface
VCI-Server	VCI-System-Service
FIFO	First In/First Out Speicher
BAL	Bus Access Layer
VCIID	Systemweit eindeutige Kennzahl eines Geräts
GUID	Eindeutige Kennzahl der Gerätekategorie
API	Application Programming Interface

2 Systemübersicht

VCI (Virtual Communication Interface) ist ein Treiber, der Applikationen einen einheitlichen Zugriff auf verschiedene Geräte von HMS Industrial Networks ermöglicht.

Der VCI.NET-Adapter basiert auf dem VCI, das eine interface-basierte C++ API bietet. In diesem Handbuch ist die .NET-Programmierschnittstelle Ixxat.Vci4.dll beschrieben.

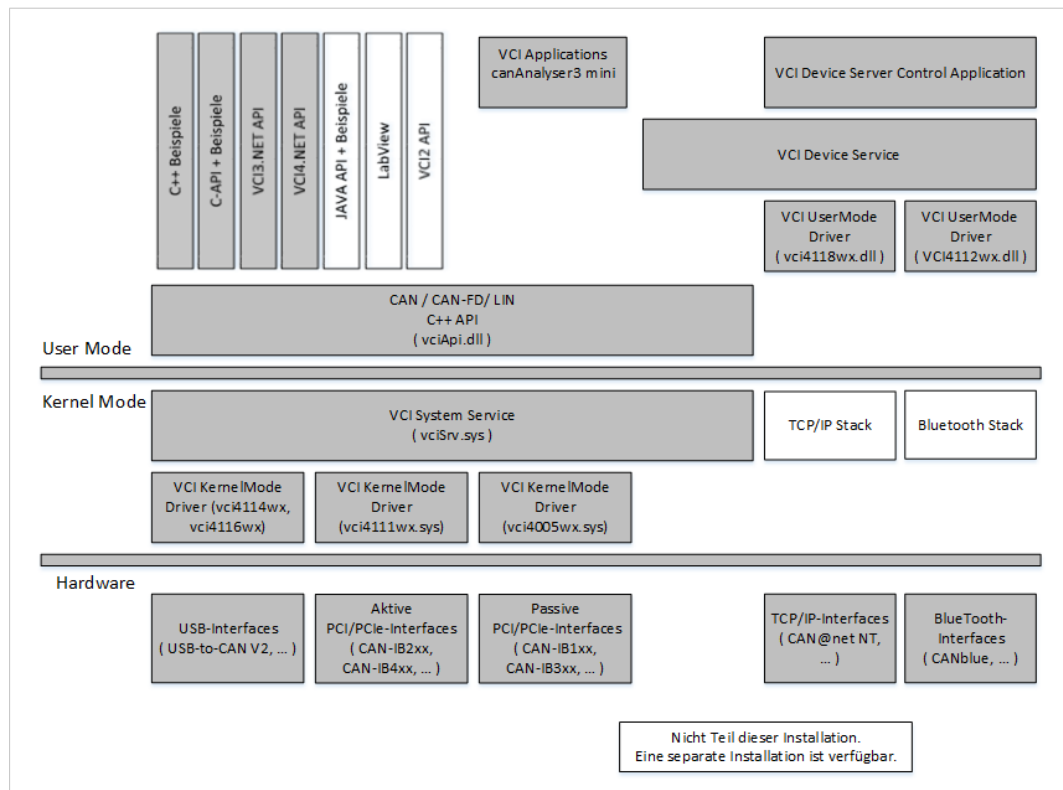


Fig. 1 Systemkomponenten

2.1 Komponenten des VCI .NET-Adapters

Die VCI .NET-Adapter enthält je einen Satz .NET-Assemblies für .NET 3.5 und für .NET 4.0 und höher, abgelegt in den jeweiligen Unterverzeichnissen NET35 und NET40. Beim Einbinden via NuGet wird die richtige Version ins Projektverzeichnis kopiert. Die Assemblies sind bis auf die Abhängigkeiten zu den jeweiligen System-Assemblies funktionsgleich. Der Adapter läuft auf VCI3- und auf VCI4-Installationen.

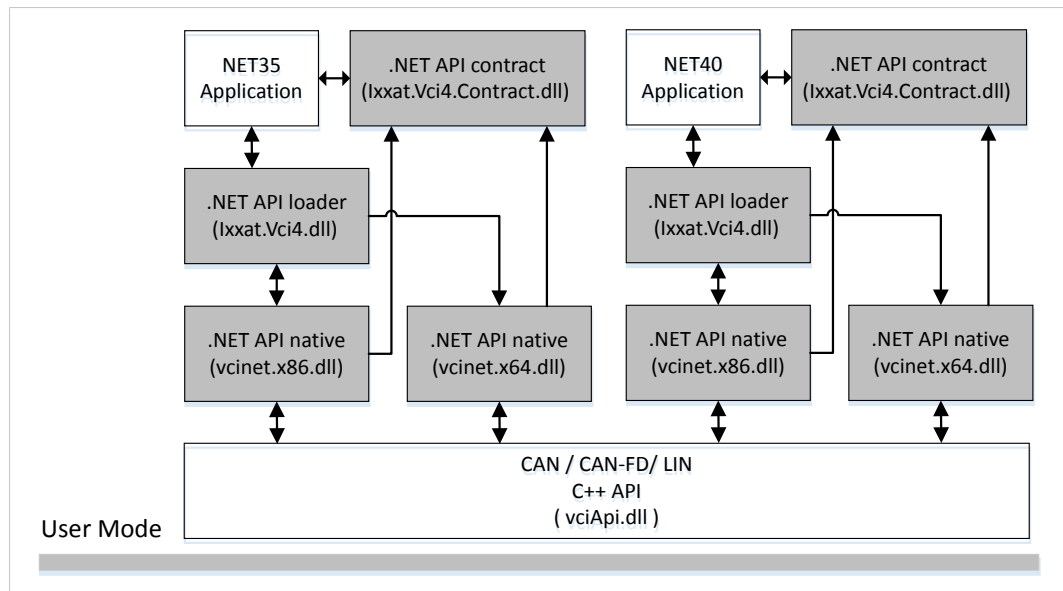


Fig. 2 VCI3 .NET-Adapter

- *Ixxat.Vci4.Contract.dll*: Enthält grundlegende Klassen und Schnittstellendefinitionen, definiert Schnittstelle (Contract) zwischen VCI .NET-Adapter und Applikation.
- *Ixxat.Vci4.dll*: Enthält minimalen Lader, der je nach verwendeter Prozessor-Architektur die entsprechende Native-Komponente (*vcinet.x86.dll* oder *vcinet.x64.dll*) lädt. Vereinfacht Deployment von Applikationen, die architekturunabhängig (AnyCPU) kompiliert sind.
- *vcinet.x86.dll*: Native-Komponente für x86-Systeme
- *vcinet.x64.dll*: Native-Komponente für x64-Systeme

Unterschiede zur VCI .NET API Version 3:

- vereinfachtes Deployment und reduzierte Abhängigkeiten zwischen verschiedenen Applikationen, da keine Installation im GAC
- geänderte Deklarationen durch Auslagern der Schnittstellen in *Ixxat.Vci4.Contract.dll* und Implementierung des Laders *Ixxat.Vci4.dll*
- zusätzliche Schnittstellen *ICanChannel2*, *ICanSocket2*, *ICanScheduler2*, *ICanMessage2* und Value-Typen *CanBitrate2*, *CanFdBitrate* und *CanLineStatus2* zur CAN-FD-Unterstützung

2.2 Legacy-Schnittstellen

2.2.1 VCI V3

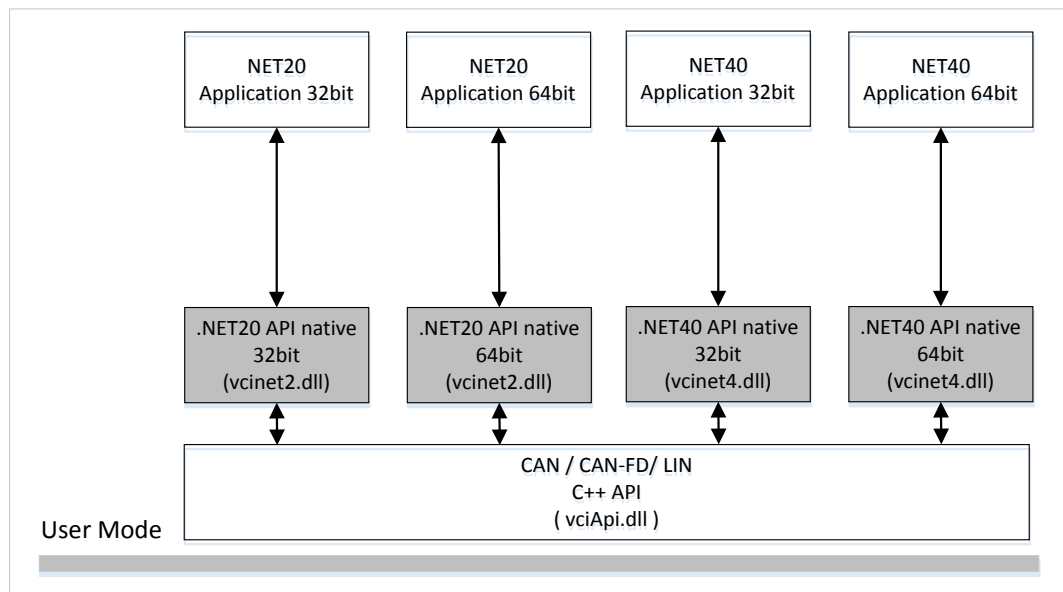


Fig. 3 VCI V3-Schnittstellen

Aus Kompatibilitätsgründen werden die mit der VCI V3 verwendeten Schnittstellen auch mit der VCI V4 installiert. HMS Industrial Networks empfiehlt für neue Entwicklungen ausschließlich die VCI.NET API Version 4 zu verwenden. Wenn der integrierte VCI-V3-Adapter für eine existierende VCI-V3-Applikation verwendet wird siehe Kapitel [Applikationen portieren, S. 9](#) für weitere Informationen.

2.2.2 VCI V2

Um eine bestehende VCI-V2-basierte Applikation mit der VCI V4 zu verwenden, muss der VCI-V2-Adapter installiert werden. Für weitere Informationen *ReadMe*-Datei in VCI-V2-Installationsordner beachten.

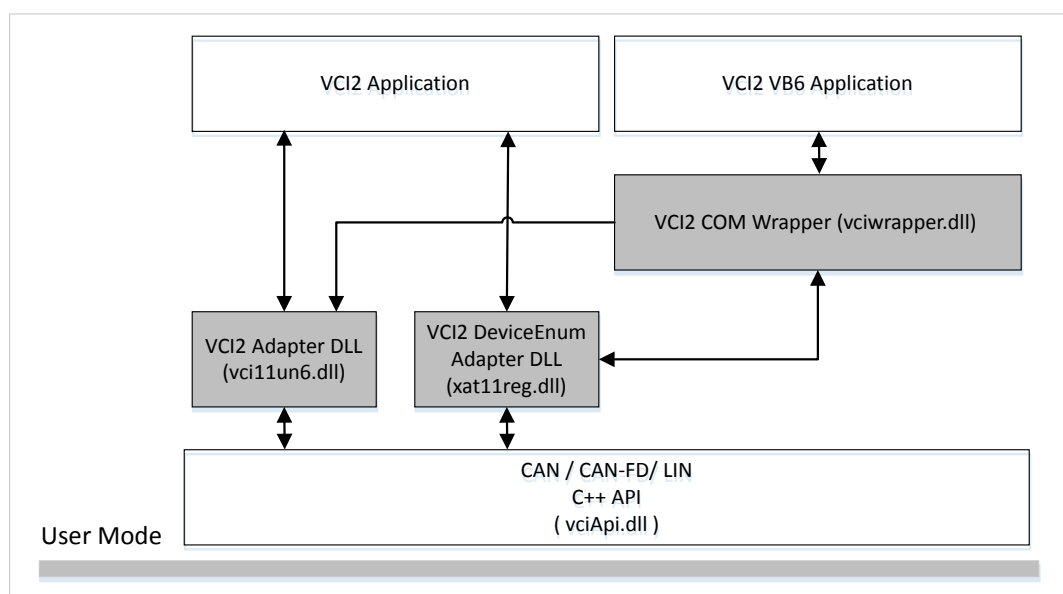
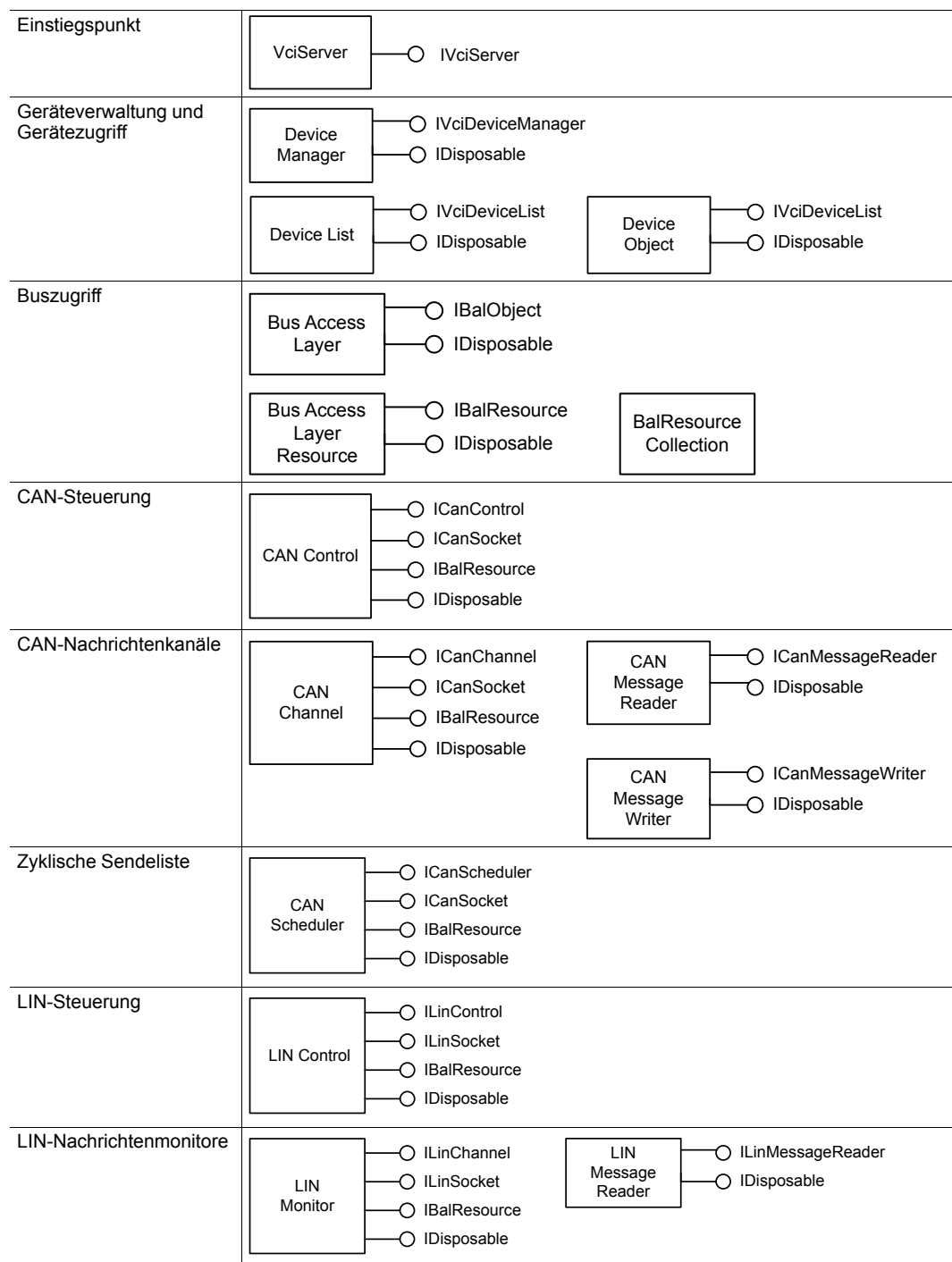


Fig. 4 VCI-V2-Adapter

2.3 Teilkomponenten und .NET Interfaces/Klassen



2.4 Programmierbeispiele

Bei der Installation des VCI-Treibers, werden automatisch Programmierbeispiele in `c:\Users\Public\Documents\HMS\IXXAT VCI 4.0\Samples\dotnet` installiert.

3 .NET API einbinden

3.1 Manuell in eigene Projekte einbinden

- ▶ Abhängigkeiten (Dependencies) zu Projekt hinzufügen.
Ixxat.Vci4.Contact.dll und Lader *Ixxat.Vci4.dll* sind notwendig.
- ▶ Native-Komponenten (*vcinet.x86.dll* und *vcinet.x64.dll*) ins bin-Verzeichnis kopieren.

3.2 Via NuGet eine eigene Projekte einbinden

Das Einbinden via NuGet automatisiert die beim manuellen Einbinden notwendigen Schritte. Via NuGet sind das Strongly-Named-Paket *Ixxat.Vci4StrongName* und das Paket ohne zugeordneten Strong-Name *Ixxat.Vci4* erhältlich.

- ▶ Paket *Ixxat.Vci4StrongName* für das Projekt installieren.
- ▶ Weitere Informationen in Handbüchern (in Paket *Ixxat.Vci4.Manuals*) und auf www.nuget.org beachten.

Verwendung von älteren VisualStudio-Versionen (VS2012 und früher)

Ein Bug in älteren VisualStudio-Versionen (VS2012 und früher) verliert manchmal den Copy-Task während des Builds, der die nativen Komponenten in das bin-Verzeichnis (*vcinet.x86.dll* und *vcinet.x64.dll*) kopiert.

- ▶ Batch-Build-Befehl als Workaround verwenden.
- ▶ Wenn während des Startups Exceptions auftreten, Exception-Text auf Hinweise prüfen und prüfen, ob alle benötigten Komponenten in das Output-Verzeichnis installiert sind.

3.3 Applikationen portieren

Die VCI-API.DLL der VCI 4 ist kompatibel zur VCI 3. Bei der Installation der VCI.NET-API-Version 4 wird Version 3 mitinstalliert.

Um die Applikationen der VCI3-.NET-API auf den aktuellen VCI4-.NET-Adapter zu portieren, werden folgende Sourcen geändert:

- Using-Anweisungen
- Zugriff auf Device Manager
- Verwendung von CAN/LIN-Nachrichten
- Abfrage Channel Status

Using-Anweisungen

```
// Version3
using Ixxat.Vci3;
// Version4
using Ixxat.Vci4;
```

Zugriff auf Device Manager

```
// Version3
deviceManager = VciServer.GetDeviceManager();
// Version4
deviceManager = VciServer.Instance().DeviceManager;
```

Verwendung von CAN/LIN-Nachrichten (Transmit)

Durch die Abstraktion von Nachrichten über Interfaces ist die Verwendung einer Factory-Klasse notwendig:

```
// Version3
CanMessage canMsg = new CanMessage();
// Version4
IMessageFactory factory = VciServer.Instance().MsgFactory;
ICanMessage canMsg = (ICanMessage)factory.CreateMsg(typeof(ICanMessage));
```

Verwendung von CAN/LIN-Nachrichten (Receive)

Ausschließlich die Deklaration ist betroffen.

```
// Version3
CanMessage canMessage;
// Version4
ICanMessage canMessage;
```

Abfrage Channel Status

Die Änderung der Implementierung des LineStatus (um uninitialisierte Status zu unterscheiden) macht eventuell Anpassungen beim Zugriff auf diese Objekte notwendig.

4 Geräteverwaltung und Gerätezugriff

Die Geräteverwaltung ermöglicht das Auflisten und den Zugriff auf die beim VCI-Server angemeldeten Geräte.

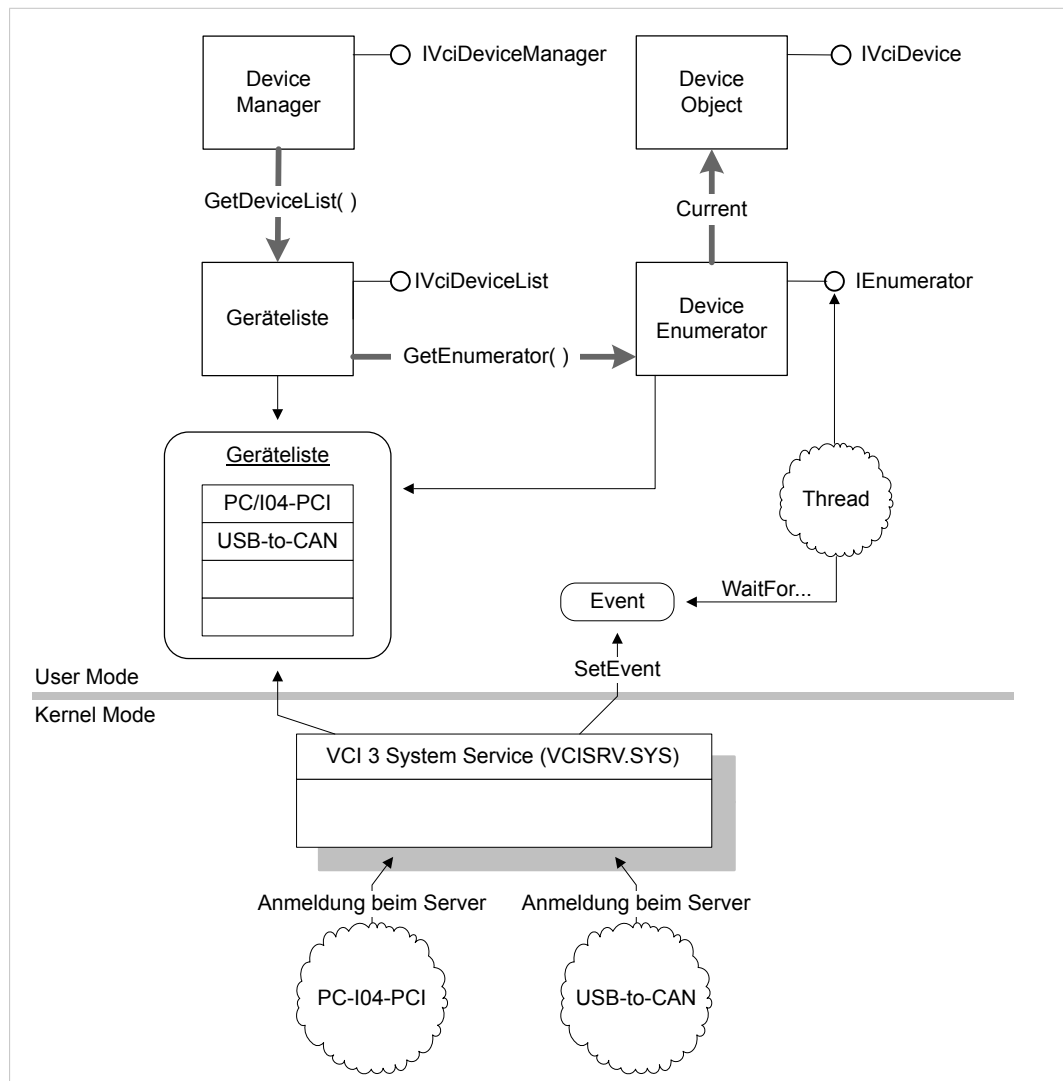


Fig. 5 Komponenten der Geräteverwaltung

Der VCI-Server verwaltet alle Geräte in einer systemweiten globalen Geräteliste. Beim Start des Computers oder wenn eine Verbindung zwischen Gerät und Computer hergestellt wird, wird das Gerät automatisch beim Server angemeldet. Ist ein Gerät nicht mehr verfügbar, weil z.B. die Verbindung unterbrochen ist, wird das Gerät automatisch aus der Geräteliste entfernt.

Auf die angemeldeten Geräte wird mit dem VCI-Gerätemanager oder dessen Schnittstelle `IVciDeviceManager` zugegriffen. Eine Referenz auf diese Schnittstelle liefert das Property `VciServer.DeviceManager`.

Wichtigste Geräteinformationen

Schnittstelle	Typ	Beschreibung
Beschreibung	String mit Bezeichnung des Interface	Zum Beispiel USB-to-CAN compact
<i>VciObjectId</i>	Eindeutige ID des Geräts	Der Server weist jedem Gerät bei der Anmeldung eine systemweit eindeutige ID (VCIID) zu. Diese ID wird für spätere Zugriffe auf das Gerät benötigt.
<i>DeviceClass</i>	Gerätekategorie	Alle Gerätetreiber kennzeichnen ihre unterstützte Geräte-Klasse mit einer weltweit eindeutigen und einmaligen Kennzahl (GUID). Unterschiedliche Geräte gehören unterschiedlichen Geräteklassen an, z. B. hat das USB-to-CAN eine andere Geräteklasse, als die PC-I04/PCI.
<i>UniqueHardwareId</i>	Hardware-ID	Jedes Gerät hat eine eindeutige Hardware-ID. Die ID kann verwendet werden, um zwischen zwei Interfaces zu unterscheiden oder um nach einem Gerät mit bestimmter ID zu suchen. Bleibt auch bei Neustart des Systems erhalten. Kann daher in Konfigurationsdateien gespeichert werden und ermöglicht automatische Konfiguration der Anwendersoftware nach Programmstart und Systemstart.
<i>DriverVersion</i>	Versionsnummer des Treibers	
<i>HardwareVersion</i>	Versionsnummer des Interface	
<i>Equipment</i>	Technische Ausstattung des Interface	Enthaltene Tabelle von VciCtrlInfo-Strukturen gibt Auskunft über Anzahl und Art der auf dem Interface vorhandenen Busanschlüsse. Tabelleneintrag 0 beschreibt Busanschluss 1, Tabelleneintrag 1 den Busanschluss 2 usw.

4.1 Verfügbare Geräte auflisten

- Um auf globale Geräteliste zuzugreifen, Methode `IVciDeviceManager.GetDeviceList` aufrufen.
 - ➡ Liefert Zeiger auf Schnittstelle `IVciDeviceList` der Geräteliste zurück.

Änderungen an der Geräteliste können überwacht und Enumeratoren für die Geräteliste angefordert werden. Es gibt verschiedene Möglichkeiten durch die Geräteliste zu navigieren.

Enumeratoren anfordern

Methode `IVciDeviceList.GetEnumerator` liefert `IEnumerator` Interface eines neuen Enumeratorobjekts für die Geräteliste.

- Methode `IEnumerator.Current` aufrufen.
 - ➡ Liefert bei jedem Aufruf ein neues Geräteobjekt mit Informationen zum Interface.
- Für Zugriff auf Informationen vom Property `Current` des Standard-Interfaces `IEnumerator` gelieferte pure Objektreferenz in Typ `IVciDevice` umwandeln.
- Um internen Index zu erhöhen, Methode `IEnumerator.MoveNext` aufrufen.
 - ➡ `IEnumerator.Current` liefert Geräteobjekt für nächstes Interface.

Liste ist vollständig durchlaufen wenn die Methode `IEnumerator.MoveNext` den Wert `FALSE` zurückliefert.

Internen Listenindex zurücksetzen

- ▶ Methode `IEnumerator.Reset` aufrufen.
 - ➔ Folgender Aufruf der Methode `IEnumerator.MoveNext` liefert wieder Informationen zum ersten Gerät in Geräteliste.

Geräte die sich während des laufenden Betriebs hinzufügen oder entfernen lassen, wie z. B. USB-Geräte melden sich nach dem Einstecken beim Server an und mit Ausstecken wieder ab. Die Geräte werden auch angemeldet oder abgemeldet, wenn beim Gerätemanager vom Betriebssystem ein Gerätetreiber aktiviert oder deaktiviert wird.

Änderungen an Geräteliste überwachen

- ▶ `AutoResetEvent`-Objekt oder `ManualResetEvent`-Objekt erzeugen.
- ▶ Objekt mit `IVciDeviceList.AssignEvent` der Liste zuteilen.



`AutoResetEvent` verwenden, damit Event in den signalisierten Zustand gesetzt wird, wenn sich nach Aufruf der Methode ein Gerät beim VCI-Server anmeldet oder abmeldet.

4.2 Auf einzelne Geräte zugreifen

Alle IXXAT-Interfaces bieten ein oder mehrere Komponenten bzw. Zugriffsebenen für unterschiedliche Anwendungsbereiche. Relevant ist hier der Bus Access Layer (BAL). Der BAL erlaubt die Steuerung der Controller und ermöglicht die Kommunikation mit dem Feldbus.

Die unterschiedlichen Zugriffsebenen eines IXXAT-Interfaces können nicht gleichzeitig geöffnet werden. Öffnet z. B. eine Anwendung den BAL, kann die von der CANopen Master API verwendete Zugriffsebene erst wieder geöffnet werden, nachdem der BAL freigegeben oder geschlossen wurde.

Bestimmte Zugriffsebenen sind zusätzlich gegen mehrfaches Öffnen geschützt, z. B. können zwei CANopen-Applikationen ein IXXAT-Interface nicht gleichzeitig verwenden.

Der BAL kann von mehreren Programmen gleichzeitig geöffnet werden, um zu ermöglichen, dass unterschiedliche Applikationen gleichzeitig auf verschiedene Busanschlüsse zugreifen können (weitere Informationen siehe [Auf Busanschlüsse zugreifen, S. 20](#)).

5 Kommunikationskomponenten

5.1 First-In/First-Out-Speicher (FIFO)

Das VCI enthält eine Implementierung für First-In/First-Out-Speicherobjekte.

Merkmale FIFO

- Zweitortspeicher, bei dem Daten auf Eingabeseite hineingeschrieben und auf Ausgabeseite wieder ausgelesen werden.
- Zeitliche Reihenfolge bleibt erhalten, d. h. Daten die zuerst in den FIFO geschrieben werden, werden auch wieder als erstes ausgelesen.
- Ähnelt der Funktionsweise einer Rohrverbindung und wird daher auch als Pipe bezeichnet.
- Verwendet, um Daten von Sender zu parallel laufendem Empfänger zu übertragen. Einigung über eine Art Sperrmechanismus, wer zu einem bestimmten Zeitpunkt Zugriff auf den gemeinsamen Speicherbereich hat, ist nicht notwendig.
- Arretierungsfrei, kann überlaufen, wenn Empfänger mit Auslesen der Daten nicht nachkommt.
- Sender schreibt die zu sendenden Daten mit Writer-Schnittstelle in den FIFO. Empfänger liest Daten parallel dazu mit Reader-Schnittstelle.

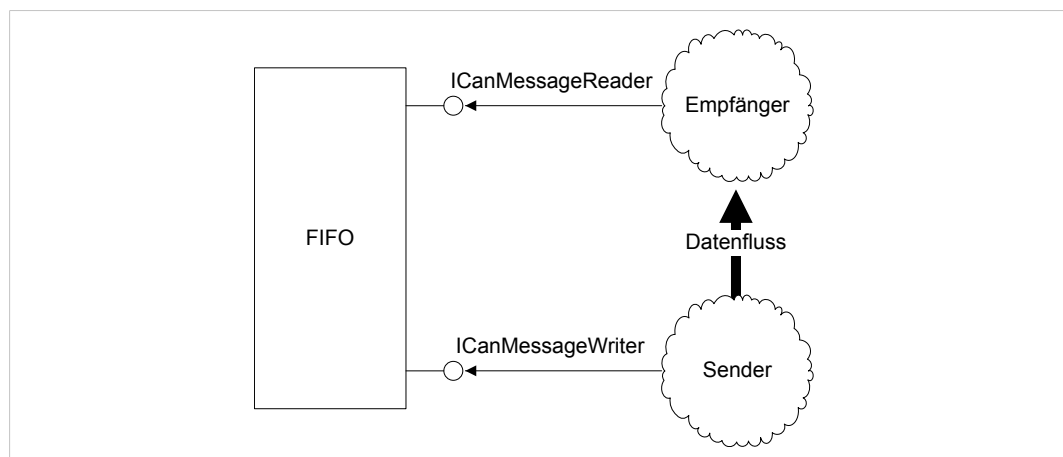


Fig. 6 FIFO-Datenfluss

Zugriff

- Schreib- und Lesezugriffe auf ein FIFO ist gleichzeitig möglich, ein Empfänger kann Daten lesen während ein Sender neue Daten in den FIFO schreibt.
- Gleichzeitiger Zugriff mehrerer Sender bzw. Empfänger auf den FIFO ist nicht möglich.
- Mehrfacher Zugriff auf die Schnittstellen `ICanMessageReader` und `ICanMessageWriter` wird verhindert, da die entsprechende Schnittstelle vom FIFO jeweils ausschließlich ein einziges Mal geöffnet werden kann, d. h. erst wenn die Schnittstelle mit `IDisposable.Dispose` freigegeben ist, kann sie erneut geöffnet werden.

- Um gleichzeitigen Zugriff unterschiedlicher Threads einer Anwendung auf eine Schnittstelle zu verhindern:
 - ▶ Sicherstellen, dass Methoden einer Schnittstelle ausschließlich von einem Thread der Anwendung aufgerufen werden können (z. B. für zweiten Thread separaten Nachrichtenkanal anlegen).
 - oder
 - ▶ Zugriff auf Schnittstelle mit entsprechenden Threads synchronisieren: Jeweils vor dem eigentlichen Zugriff auf den FIFO Funktion `Lock` und nach Abschluss des Zugriffs Funktion `Unlock` der entsprechenden Schnittstelle aufrufen.

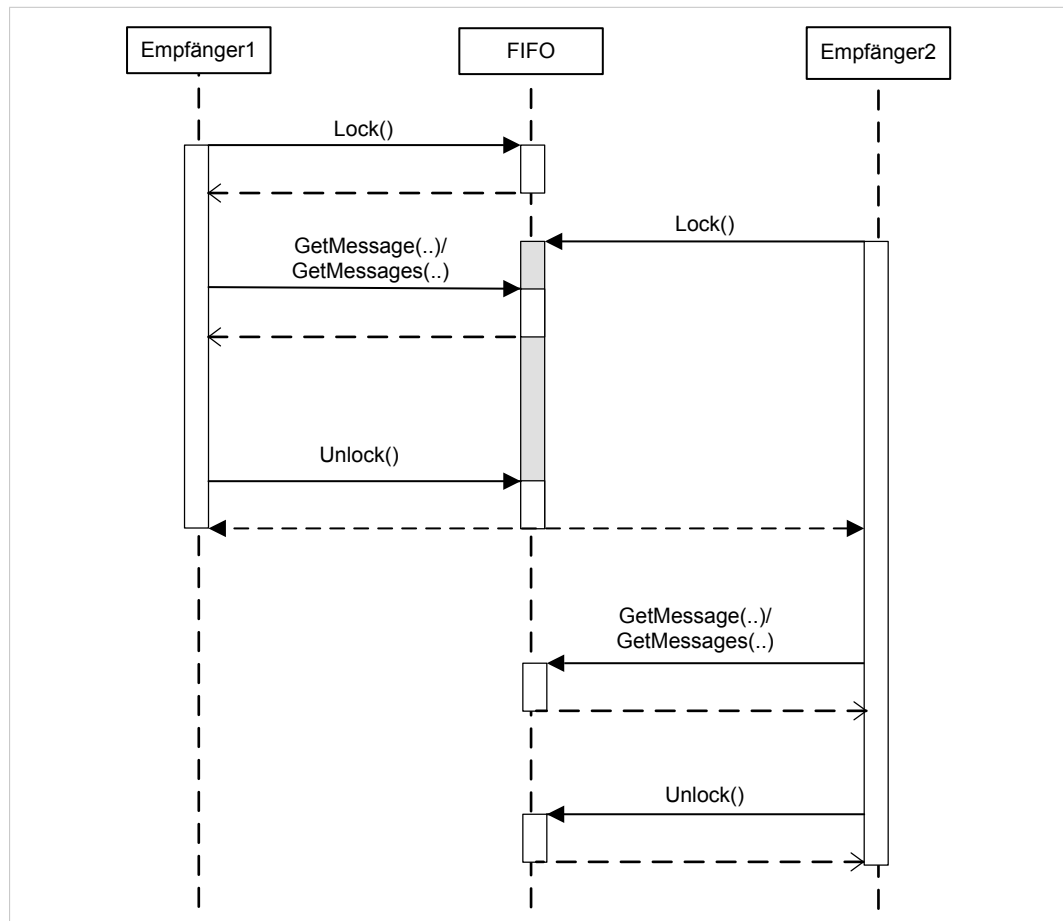


Fig. 7 Sperrmechanismus bei FIFOs

Empfänger 1 ruft die Methode `Lock` auf und erhält Zugriff auf den FIFO. Der anschließende Aufruf von `Lock` durch Empfänger 2 blockiert so lange, bis Empfänger 1 durch Aufruf der Methode `Unlock` den FIFO wieder freigibt. Erst jetzt kann Empfänger 2 mit der Bearbeitung beginnen. Auf gleiche Weise können zwei Sender synchronisiert werden, die über die Schnittstelle `ICanMessageWriter` auf einen FIFO zugreifen.

Die vom VCI zur Verfügung gestellten FIFOs erlauben den Austausch von Daten auch zwischen zwei Prozessen, d. h. über Prozessgrenzen hinweg.

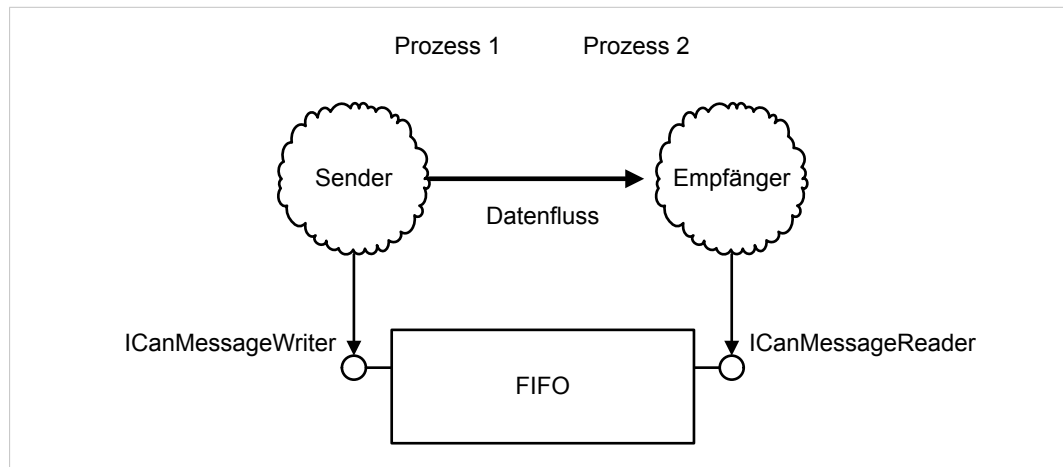


Fig. 8 FIFO für Datenaustausch zwischen zwei Prozessen

FIFOs werden auch zum Austausch von Daten zwischen im Kernel-Mode laufenden Komponenten und im User-Mode laufenden Programmen verwendet.

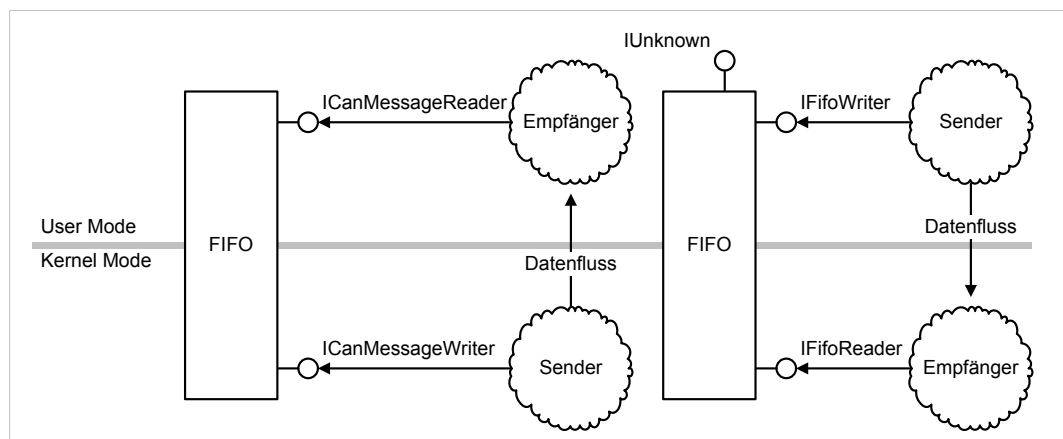


Fig. 9 Mögliche Kombinationen eines FIFO für den Datenaustausch zwischen User- und Kernel-Mode

5.1.1 Funktionsweise Empfangs-FIFO

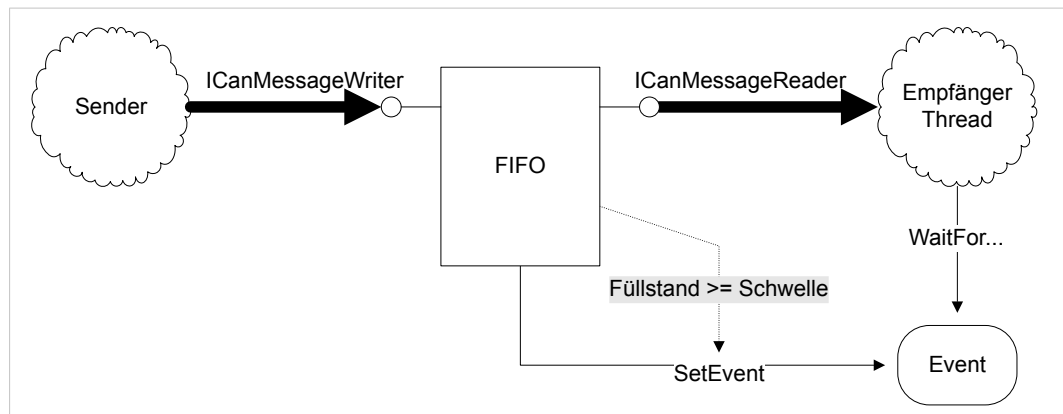


Fig. 10 Funktionsweise Empfangs-FIFO

Empfangsseitig werden FIFOs über die Schnittstelle `ICanMessageReader` angesprochen.

Auf zu lesende Dateien zugreifen:

- ▶ Um einzelne Nachricht zu lesen, Methode `GetMessage` aufrufen.
oder
- ▶ Um mehrere Nachrichten zu lesen, Funktion `GetMessages` aufrufen.
- ▶ Um ein oder mehrere ausgelesene oder verarbeitete Elemente freizugeben, Funktion `IDisposable.Dispose` aufrufen.

Eventobjekt

Dem FIFO kann ein Eventobjekt zugeordnet werden, um zu verhindern, dass der Empfänger nachfragen muss, ob neue Daten zum Lesen bereit stehen. Das Eventobjekt wird in signalisierten Zustand versetzt, wenn ein gewisser Füllstand erreicht ist.

- ▶ `AutoResetEvent` oder `ManualResetEvent` erzeugen.
 - ➡ Zurückgelieferter Handle wird mit Methode `AssignEvent` an FIFO übergeben.
- ▶ Schwelle bzw. Füllstand, bei dem Event ausgelöst wird, mit Property `Threshold` einstellen.

Im weiteren Verlauf kann die Applikation mit einer der Methoden `WaitOne` oder `WaitAll` des Eventobjekts auf das Eintreffen des Events warten und die empfangenen Daten lesen.

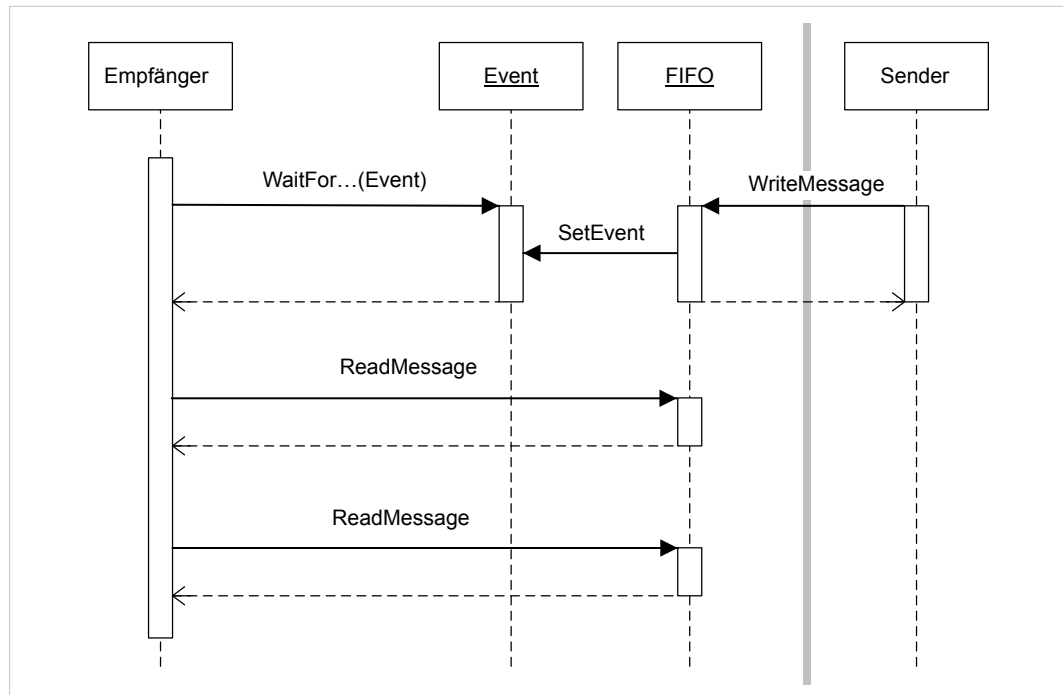


Fig. 11 Empfangssequenz beim ereignisgesteuerten Lesen von Daten aus dem FIFO



Da das Event ausschließlich bei Überschreitung der eingestellten Schwelle ausgelöst wird, sicherstellen, dass beim ereignisgesteuerten Lesen möglichst immer alle Einträge aus dem FIFO gelesen werden. Wenn die Schwelle z. B. auf 1 eingestellt ist und bei Eintreffen des Events bereits 10 Elemente im FIFO liegen und nur eines gelesen wird, dann wird ein weiteres Event erst wieder beim nächsten Schreibzugriff ausgelöst. Erfolgt vom Sender kein weiterer Schreibzugriff, liegen 9 ungelesene Elemente im FIFO, die nicht mehr als Event angezeigt werden.

5.1.2 Funktionsweise Sende-FIFO

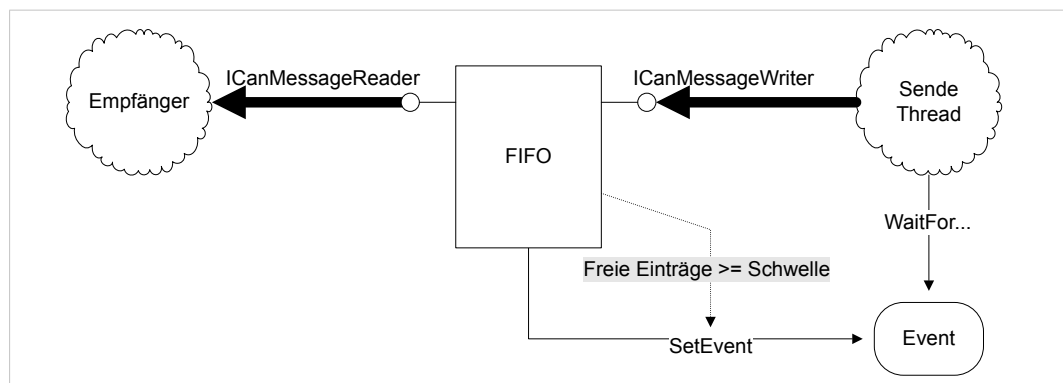


Fig. 12 Funktionsweise Sende-FIFO

Sendeseitig werden FIFOs über die Schnittstelle `ICanMessageWriter` angesprochen.

Zu sendende Daten in FIFO schreiben:

- Um einzelne Nachrichten in FIFO zu schreiben, Methode `WriteMessage` aufrufen.
oder
- Um mehrere Nachrichten in FIFO zu schreiben, Methode `WriteMessages` aufrufen.

Eventobjekt

Dem FIFO kann ein Eventobjekt zugeordnet werden, um zu verhindern, dass Empfänger prüfen muss, ob freie Elemente verfügbar sind. Das Eventobjekt wird in signalisierten Zustand versetzt, wenn die Anzahl freier Elemente einen gewissen Wert überschreitet.

- ▶ `AutoResetEvent` oder `ManualResetEvent` erzeugen.
 - ➡ Zurückgeliefertes Handle wird mit Methode `AssignEvent` an FIFO übergeben.
- ▶ Schwelle bzw. Anzahl freier Element, bei dem der Event ausgelöst wird, mit Property `Threshold` einstellen.

Im weiteren Verlauf kann die Applikation mit einer der Methoden `WaitOne` oder `WaitAll` des Eventobjekts auf das Eintreffen des Events warten und neue Daten in den FIFO schreiben.

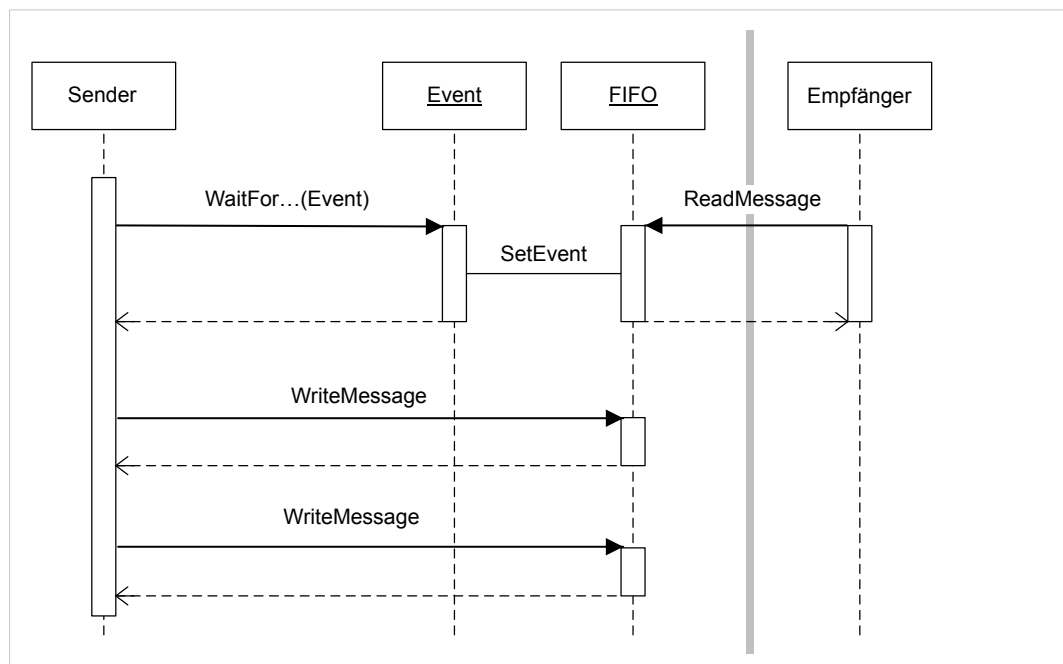


Fig. 13 Sendesequenz beim ereignisgesteuerten Schreiben von Daten in den FIFO

6 Auf Busanschlüsse zugreifen

Über den Bus Access Layer (BAL) wird auf die mit dem CAN-Interface verbundenen Feldbusse zugegriffen.

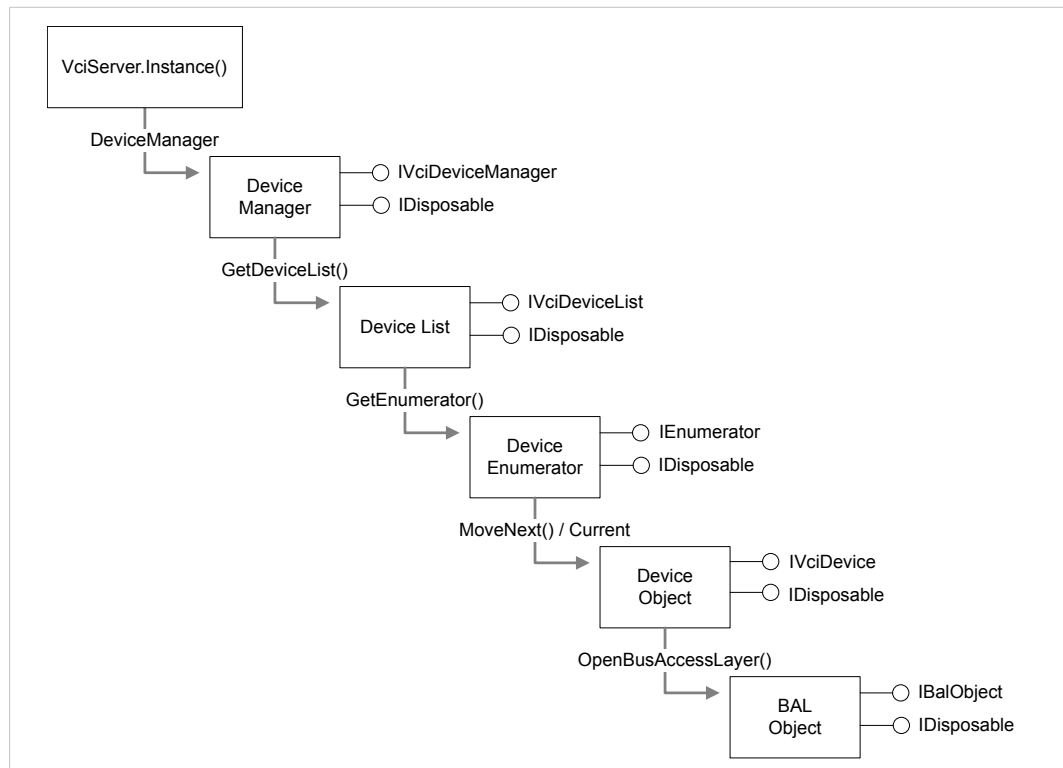


Fig. 14 Komponenten für den Buszugriff

- Adapter in Geräteliste suchen und BAL mit `IVciDeviceManager.OpenBusAccessLayer` öffnen.
- Nach dem Öffnen nicht mehr benötigte Referenzen auf den Gerätemanager, Geräteliste, Geräteenumerator oder das Geräteobjekt mit `IDisposable.Dispose` freigeben.

Für die weitere Arbeit mit dem Adapter ist nur noch das BAL-Objekt bzw. dessen Schnittstelle `IBalObject` erforderlich. Der BAL eines Interfaces kann von mehreren Programmen gleichzeitig geöffnet werden.

Das BAL-Objekt unterstützt mehrere Arten von Busanschlüssen.

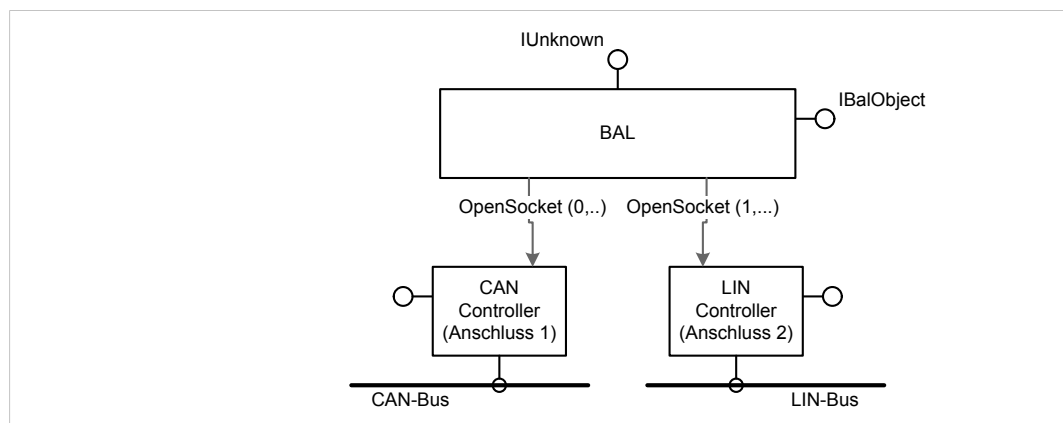


Fig. 15 BAL mit CAN- und LIN-Anschluss

Anzahl und Art der zur Verfügung gestellten Anschlüsse ermitteln

- ▶ Property `IBalObject.Resources` aufrufen.
 - ➡ Liefert Informationen in Form einer `BalResourceCollection`, die für jeden vorhandenen Busanschluss ein BAL-Ressourcenobjekt enthält.
 - ➡ BAL liefert die Versionsnummer der Geräte-Firmware über das Property `IBalObject.FirmwareVersion`.

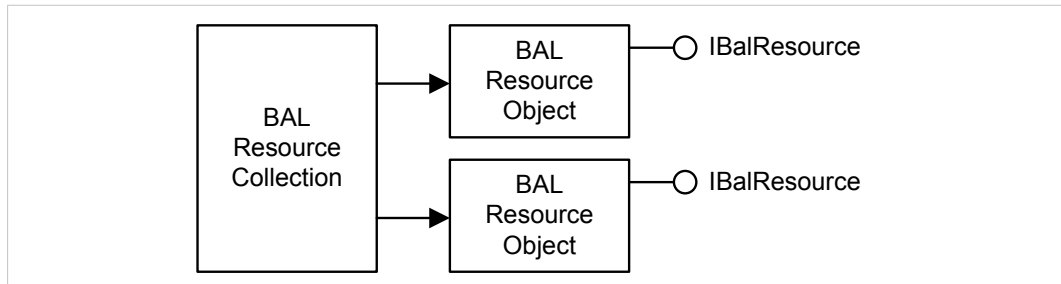


Fig. 16 BalResourceCollection mit zwei Busanschlüssen

Auf Anschluss oder Schnittstelle des Anschlusses zugreifen

Mit Methode `IBalObject.OpenSocket` auf Anschlüsse zugreifen.

- ▶ Im ersten Parameter Nummer des zu öffnenden Anschlusses angeben. Wert muss im Bereich 0 bis `IBalObject.Resources.Count-1` liegen. Zum Öffnen von Anschluss 1 den Wert 0, für Anschluss 2 den Wert 1, usw. eingeben.
- ▶ Im zweiten Parameter ID der Schnittstelle bestimmen, über die auf den Anschluss zugegriffen wird.
- ▶ Methode aufrufen.
 - ➡ Liefert Referenz auf gewünschte Schnittstelle zurück.
 - ➡ Möglichkeiten bzw. Schnittstellen eines Anschlusses sind vom unterstützten Bus abhängig.



Auf bestimmte Schnittstellen eines Anschlusses kann jeweils nur ein Programm zugreifen, auf andere beliebig viele Programme gleichzeitig. Die Regeln für den Zugriff auf die einzelnen Schnittstellen sind vom Anschlusstyp abhängig und sind in den folgenden Kapiteln genauer beschrieben.

6.1 CAN-Controller

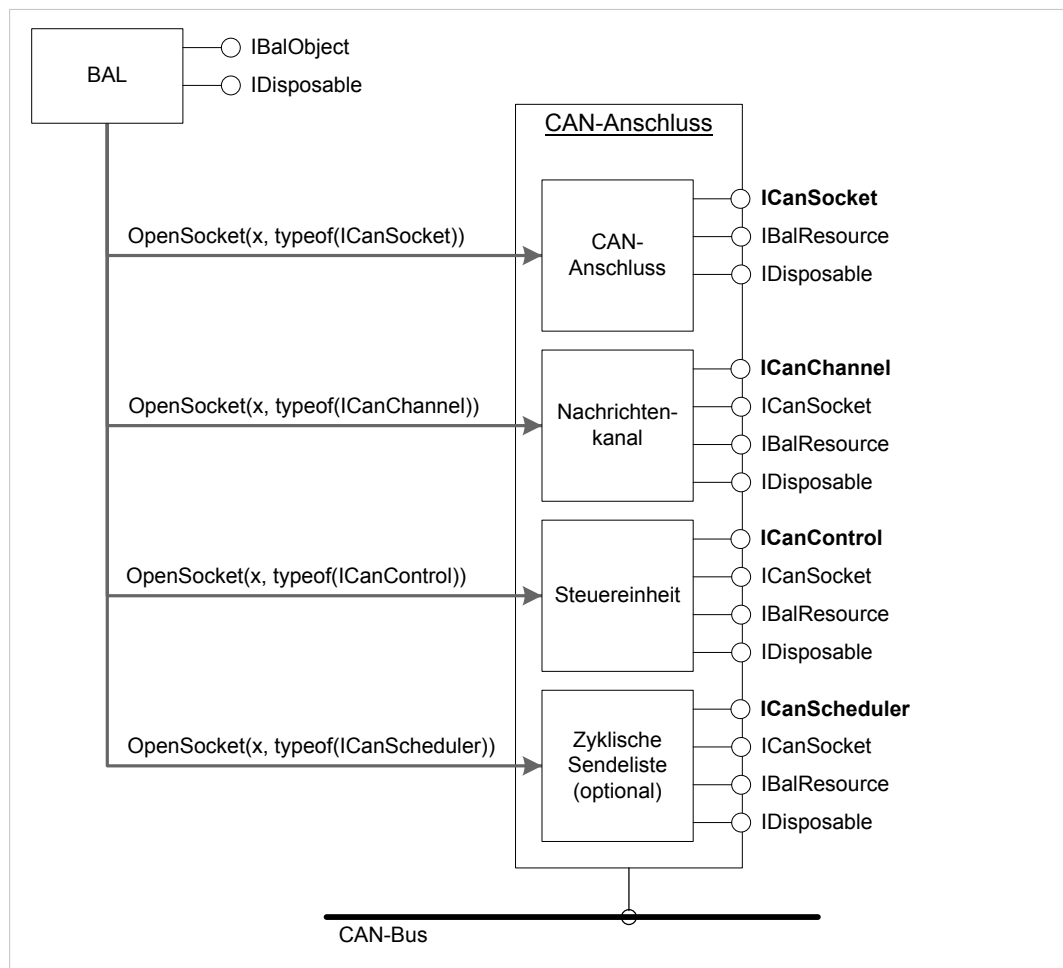


Fig. 17 Komponenten CAN-Anschluss

Zugriff auf einzelne Komponenten eines CAN-Anschlusses über folgende Schnittstellen:

- **ICanSocket**, **ICanSocket2** (CAN-Anschluss), siehe [Socket-Schnittstelle, S. 23](#)
- **ICanControl**, **ICanControl2** (Steuereinheit), siehe [Steuereinheit, S. 30](#)
- **ICanChannel**, **ICanChannel2** (Nachrichtenkanäle), siehe [Nachrichtenkanäle, S. 23](#)
- **ICanScheduler**, **ICanScheduler2** (zyklische Sendeliste), siehe [Zyklische Sendeliste, S. 38](#), optional, ausschließlich bei Geräten mit eigenem Mikroprozessor

Die erweiterten Schnittstellen **ICanSocket2**, **ICanControl2**, **ICanChannel2** und **ICanScheduler2** ermöglichen den Zugang zu den neuen Funktionen bei CAN-FD-Controllern. Sie können auch bei Standard-Controllern für erweiterte Filtermöglichkeiten verwendet werden.

6.1.1 Socket-Schnittstelle

Die Socket-Schnittstelle `ICanSocket` bzw. `ICanSocket2` dient zur Abfrage der Eigenschaften, der Möglichkeiten und des Betriebszustands des CAN-Controllers. Die Schnittstelle unterliegt keinen Zugriffsbeschränkungen und kann von beliebig vielen Anwendungen gleichzeitig geöffnet werden. Die Steuerung des Anschlusses ist über diese Schnittstelle nicht möglich.

Mit Methode `IBalObject.OpenSocket` öffnen.

- In Parameter `socketType` Typ `ICanSocket` oder `ICanSocket2` angeben.
- Methode aufrufen.

Die Eigenschaften des CAN-Anschlusses, wie z. B. unterstützte Features sind durch Properties bereitgestellt.

- Um aktuellen Betriebszustands des Controllers zu ermitteln, Property `LineStatus` aufrufen.

6.1.2 Nachrichtenkanäle

Nachrichtenkanäle bestehen aus einem Empfangs- und einem optionalen Sende-FIFO. Es sind ein oder mehrere Nachrichtenkanäle pro CAN-Anschluss möglich. CAN-Nachrichten werden ausschließlich über Nachrichtenkanäle empfangen und gesendet.

Nachrichtenkanäle mit erweiterter Funktionalität (CAN-FD) besitzen einen zusätzlichen, optionalen Eingangsfiler.

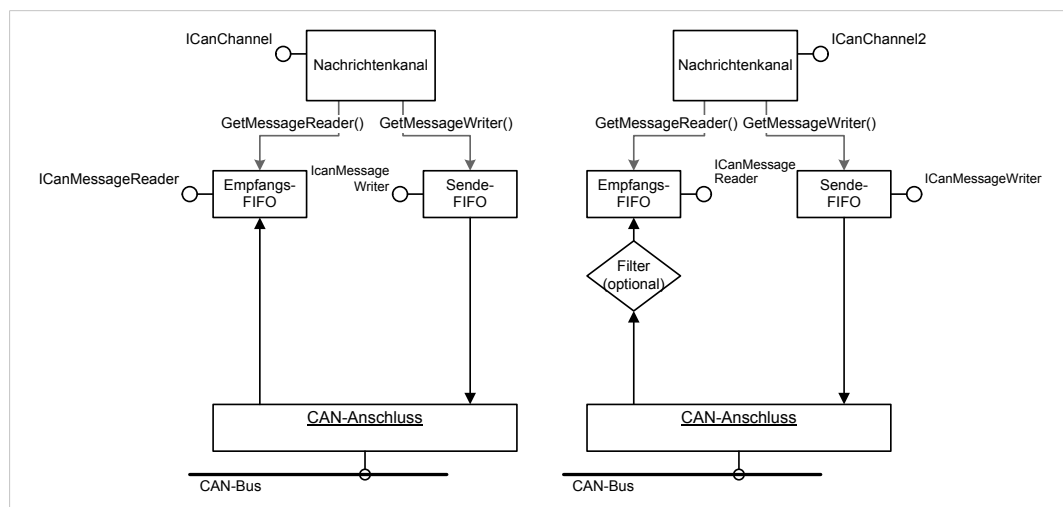


Fig. 18 Komponenten und Schnittstellen eines Nachrichtenkanals

Alle CAN-Anschlüsse unterstützen Nachrichtenkanäle vom Typ `ICanChannel` und vom Typ `ICanChannel2`. Ob bei einem Nachrichtenkanal vom Typ `ICanChannel2` die erweiterte Funktionalität nutzbar ist, hängt vom CAN-Controller des Anschlusses ab. Besitzt der Anschluss z. B. nur einen normalen CAN-Controller, kann die erweiterte Funktionalität nicht genutzt werden. Mit einem Nachrichtenkanal vom Typ `ICanChannel` kann die erweiterte Funktionalität eines CAN-FD-fähigen Controllers ebenfalls nicht genutzt werden.

Die grundlegende Funktionsweise eines Nachrichtenkanals ist unabhängig davon, ob ein Anschluss exklusiv verwendet wird oder nicht. Bei exklusiver Verwendung ist der Nachrichtenkanal direkt mit dem CAN-Controller verbunden.

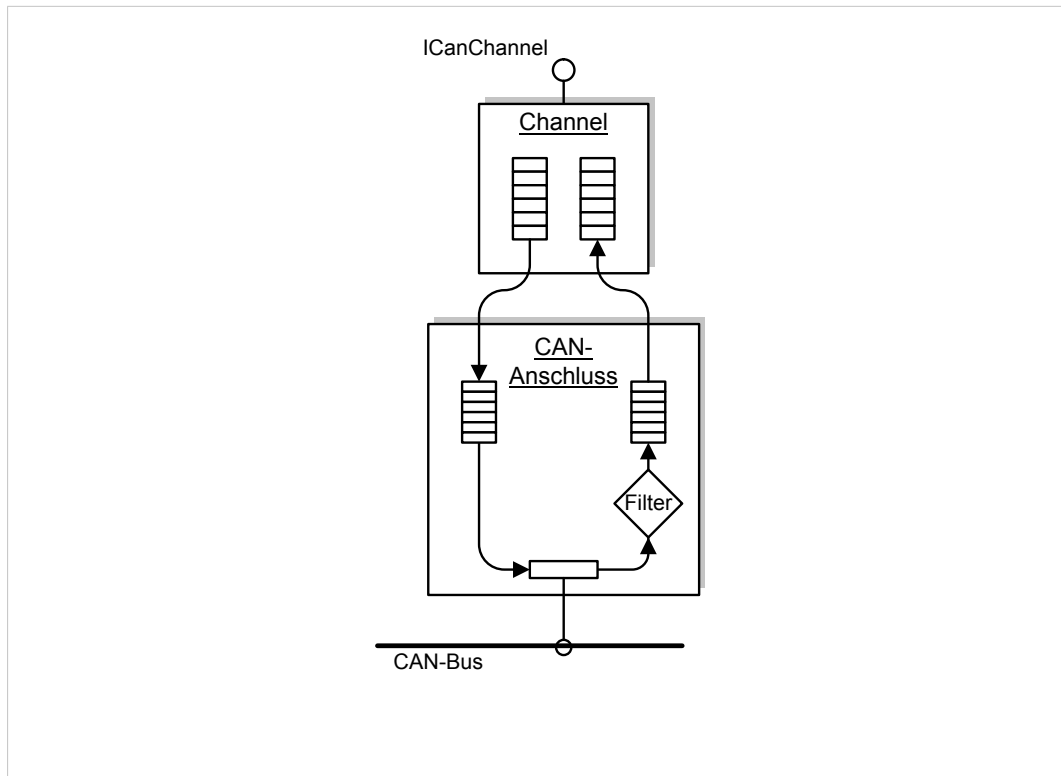


Fig. 19 Exklusive Verwendung eines Nachrichtenkanals

Bei nicht-exklusiver Verwendung sind die einzelnen Nachrichtenkanäle über einen Verteiler mit dem Controller verbunden.

Der Verteiler leitet die empfangenen Nachrichten an alle Kanäle weiter und überträgt parallel dazu deren Sendenachrichten an den Controller. Kein Kanal wird priorisiert, d. h. der vom Verteiler verwendete Algorithmus ist so gestaltet, dass alle Kanäle möglichst gleichberechtigt behandelt werden.

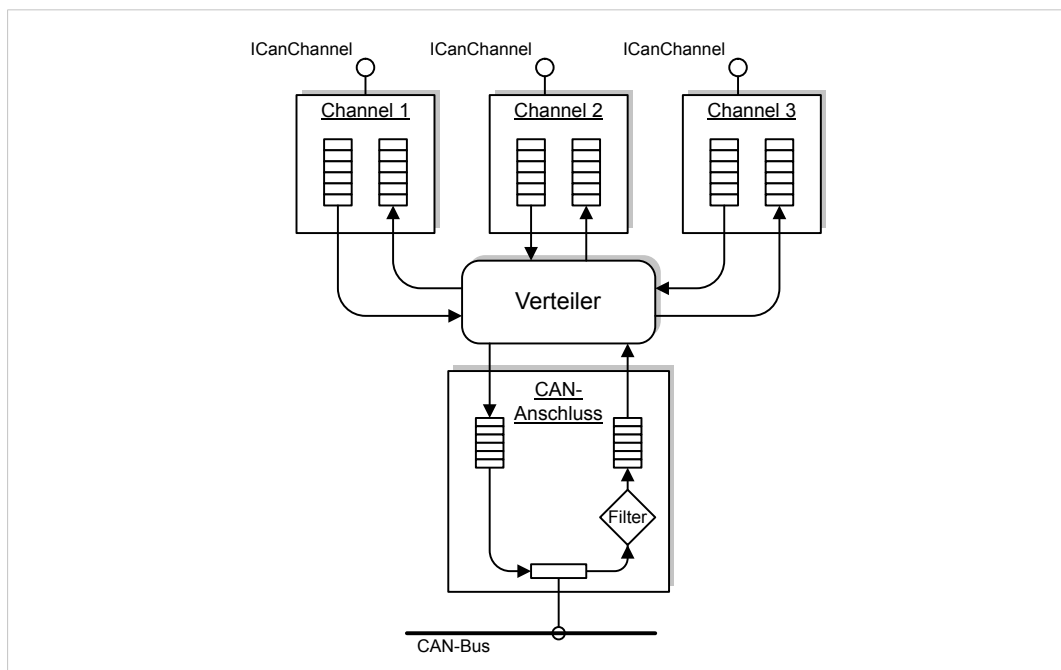


Fig. 20 CAN-Nachrichtenverteiler: mögliche Konfiguration mit drei Kanälen

Nachrichtenkanal erstellen

Mit Methode `ICanSocket.OpenSocket` bzw. für Kanäle mit erweiterter Funktionalität mit `ICanSocket2.OpenSocket` erstellen.

- ▶ In Parameter `socketType` Typ `ICanChannel` angeben.
- ▶ Wenn Controller exklusiv verwendet wird (nach erfolgreicher Ausführung kann kein weiterer Nachrichtenkanal verwendet werden), Wert `TRUE` in Parameter `exclusive` eingeben.

oder

Wenn Controller nicht-exklusiv verwendet wird (weitere Nachrichtenkanäle können geöffnet und Anschluss kann von anderen Applikationen verwendet werden), Wert `FALSE` in Parameter `exclusive` angeben.

Für die FIFOs benötigter Arbeitsspeicher schränkt die Anzahl möglicher Kanäle ein.

Nachrichtenkanal initialisieren

Ein neu erzeugter Nachrichtenkanal besitzt weder Empfangs-FIFO noch Sende-FIFO. Vor der ersten Verwendung ist eine Initialisierung notwendig.

Mit Methode `ICanChannel.Initialize` bzw. bei Kanäle mit erweiterter Funktionalität mit `ICanChannel2.Initialize` initialisieren und Empfangs-FIFO und Sende-FIFO erstellen.

- ▶ In den Parametern Größe des jeweiligen FIFOs in Anzahl CAN-Nachrichten bestimmen.
- ▶ Methode aufrufen.

Bei Nachrichtenkanälen mit erweiterter Funktionalität kann ein zusätzliches, optionales Eingangsfiler eingerichtet werden.

- ▶ Bei 29-Bit-ID-Filter Größe der Filtertabelle in Anzahl IDs in Parameter `filterSize` bestimmen.
Bei 11-Bit-ID-Filter ist Größe der Filtertabelle auf 2048 eingestellt und kann nicht geändert werden.
- ▶ Wenn kein Eingangsfiler benötigt wird, `filterSize` auf Wert 0 setzen.
- ▶ Funktionsweise für 11-Bit-ID-Filter und 29-Bit-ID-Filter in Parameter `filterMode` bestimmen.
- ▶ Methode aufrufen.



Anfängliche Funktionsweise kann später bei inaktiven Nachrichtenkanal für beide Filter getrennt mit der Funktion `SetFilterMode` geändert werden.

Nachrichtenkanal aktivieren

Ein neuer Nachrichtenkanal ist inaktiv. Nachrichten können ausschließlich gesendet und empfangen werden, wenn der Nachrichtenkanal aktiv und der CAN-Controller gestartet ist.

- ▶ Nachrichtenkanal mit Methode `ICanChannel.Activate` aktivieren.
- ▶ Nachrichtenkanal mit Methode `ICanChannel.Deactivate` deaktivieren.

CAN-Nachrichten empfangen

Die auf dem Bus ankommenden und vom Filter akzeptierten Nachrichten werden in den Empfangs-FIFO eingetragen.

- ▶ Zum Lesen erforderliche Schnittstelle `ICanMessageReader` mit `ICanChannel1.GetMessageReader` bzw. bei Kanälen mit erweiterter Funktionalität mit `ICanChannel2.GetMessageReader` anfordern.

Nachrichten aus dem FIFO lesen:

- ▶ Methode `ReadMessage` aufrufen.
oder
- ▶ Um mehrere Nachrichten über einen Aufruf zu lesen (optimiert auf hohen Datendurchsatz), Feld von CAN-Nachrichten anlegen.
- ▶ Feld an Methode `ReadMessages` übergeben.
 - ➡ `ReadMessages` versucht Feld mit empfangenen Daten zu füllen.
 - ➡ Anzahl tatsächlich gelesener Nachrichten wird über Rückgabewert signalisiert.

Mögliche Verwendung von `ReadMessage`

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

Mögliche Verwendung von `ReadMessages`

```
void DoMessages( ICanMessageReader reader )
{
    ICanMessage[] messages;

    int readCount = reader.ReadMessages(out messages);
    for( int idx = 0; idx < readCount; idx++ )
    {
        // Verarbeitung der Nachricht
    }
}
```

Empfangszeitpunkt einer Nachricht

Der Empfangszeitpunkt einer Nachricht ist im Property *TimeStamp* über das Interface *ICanMessage* bzw. *ICanMessage2* verfügbar. Das Property enthält die Anzahl der Timer-Ticks, die seit dem Start des Controllers bzw. der Hardware oder seit dem letzten Überlauf des Zählers vergangen sind.

Berechnung der Dauer eines Ticks bzw. Auflösung der Zeitstempel in Sekunden: (t_{tsc}):

- $t_{\text{tsc}} [\text{s}] = \text{TimeStampCounterDivisor} / \text{ClockFrequency}$

Felder *TimeStampCounterDivisor* und *ClockFrequency*, siehe Properties *ICanSocket.ClockFrequency* und *ICanSocket.TimeStampCounterDivisor*

- bei Kanälen mit erweiterter Funktionalität:

$$t_{\text{tsc}} [\text{s}] = \text{TimeStampCounterDivisor} / \text{ClockFrequency}$$

Felder *TimeStampCounterDivisor* und *ClockFrequency*, siehe Properties *ICanSocket2.ClockFrequency* und *ICanSocket2.TimeStampCounterDivisor*

Berechnung des relativen Empfangszeitpunkts (T_{rx}):

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

Beim Start der Steuereinheit wird eine Nachricht vom Typ *CanMsgFrameType.Info* in die Empfangs-FIFOs aller aktiven Nachrichtenkanäle geschrieben. Der Zeitstempel dieser Nachricht enthält den relativen Startzeitpunkt des Controllers.

CAN-Nachrichten senden

Nachrichten werden über den Sende-FIFO des Nachrichtenkanals gesendet.

- ▶ Zum Senden erforderliche Schnittstelle *ICanMessageWriter* mit Methode *ICanChannel.GetMessageWriter* bzw. bei Kanälen mit erweiterter Funktionalität mit *ICanChannel2.GetMessageWriter* anfordern.
- ▶ Nachrichten mit Methode *SendMessage* senden.
- ▶ Im Parameter *message* die zu sendende Nachricht vom Typ *CanMessage* übergeben.
- ▶ Um Nachricht verzögert zu senden, in Parameter *TimeStamp* Wert ungleich 0 angeben (weitere Informationen siehe [Nachrichten verzögert senden, S. 28](#)).

Ausschließlich Nachrichten vom Typ *CanMsgFrameType.Data* können gesendet werden. Andere Nachrichtentypen werden vom Anschluss ignoriert und automatisch verworfen.

Mögliche Verwendung von SendMessage

```
bool SendByte( ICanMessageWriter writer, UInt32 id, Byte data )
{
    IMessageFactory factory = VciServer.Instance().MsgFactory;
    ICanMessage canMsg = (ICanMessage)factory.CreateMsg(typeof(ICanMessage));

    // 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);
}
```

Nachrichten verzögert senden

Controller mit gesetztem Bit `ICanSocket.SupportsDelayedTransmission` unterstützen die Möglichkeit Nachrichten verzögert, mit einer Wartezeit zwischen zwei aufeinanderfolgenden Nachrichten zu senden.

Verzögertes Senden kann verwendet werden, um die Nachrichtenlast auf dem Bus zu reduzieren. Damit lässt sich verhindern, dass andere am Bus angeschlossene Teilnehmer zu viele Nachrichten in zu kurzer Zeit erhalten, was bei leistungsschwachen Knoten zu Datenverlust führen kann.

- Im Feld *CanMessage.TimeStamp* Zeit in Ticks angegeben, die mindestens verstreichen muss bevor die Nachricht an den Controller weitergegeben wird.

Verzögerungszeit

- Wert 0 bewirkt keine Verzögerung, d. h. die Nachricht wird zum nächst möglichen Zeitpunkt gesendet.
- Maximal mögliche Verzögerungszeit bestimmt das Feld *ICanSocket.MaxDelayedTXTicks*.
- Auflösung eines Ticks in Sekunden wird berechnet mit den Werten aus den Feldern *ICanSocket.ClockFrequency* und *ICanSocket.DelayedTXTimeDivisor* bzw. *ICanSocket2.DelayedTXTimerClockFrequency* und *ICanSocket2.DelayedTXTimerDivisor*.

Berechnung der Auflösung eines Ticks in Sekunden

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

Die angegebene Verzögerungszeit ist ein Minimalwert, da nicht garantiert werden kann, dass die Nachricht exakt nach Ablauf der angegebenen Zeit gesendet wird. Außerdem muss beachtet werden, dass bei gleichzeitiger Verwendung mehrerer Nachrichtenkanäle an einem Anschluss der angegebene Wert prinzipiell überschritten wird, da der Verteiler alle Kanäle nacheinander abarbeitet.

Empfehlung:

- Bei Applikationen, die eine genauere zeitliche Abfolge benötigen, Anschluss exklusiv verwenden.

Nachrichten einmalig senden

Sendenachrichten mit gesetztem `SingleShotMode`-Flag versucht der Controller nur einmal zu senden. Gelingt dieser Sendeversuch nicht, wird die Nachricht verworfen und es erfolgt keine automatische Sendewiederholung.

Diese Situation tritt z. B. auf, wenn zwei oder mehrere Busteilnehmer gleichzeitig senden. Verliert der Teilnehmer, der eine Nachricht mit gesetztem `SingleShotMode`-Flag sendet die Buszuteilung (Arbitrierung), wird die Nachricht verworfen und es erfolgt kein weiterer Sendeversuch.

Die Funktionalität ist ausschließlich verfügbar, wenn das Property `ISocket2.SupportsSingleShotMessages` `TRUE` liefert.

Sendenachrichten mit hoher Priorität

Sendenachrichten mit gesetztem `HighPriorityMsg`-Flag werden vom Controller in einen controller-spezifischen Sendepuffer eingetragen, der Vorrang gegenüber Nachrichten im normalen Sendepuffer hat und vorrangig sendet.

Die Funktionalität ist ausschließlich verfügbar, wenn das Property `ISocket2.SupportsHighPriorityMessages` `TRUE` liefert. Bei Verwendung des Bits beachten, dass sich damit keine Nachrichten überholen lassen, die bereits im Sende-FIFO sind. Die Funktionalität ist von untergeordneter Bedeutung bzw. kann nur dann sinnvoll eingesetzt werden, wenn der Anschluss exklusiv geöffnet ist und der Sende-FIFO vor dem Eintragen einer Nachricht mit gesetztem `HighPriorityMsg`-Flag leer ist.

6.1.3 Steuereinheit

Die Steuereinheit bietet über die Schnittstelle `ICanControl` folgende Funktionen:

- Konfiguration des CAN-Controllers
- Konfiguration der Übertragungseigenschaften des CAN-Controllers
- Konfiguration von CAN-Nachrichtenfiltern
- Abfrage des aktuellen Betriebszustands

Um sicherzustellen, dass nicht mehrere Applikationen z. B. gleichzeitig versuchen den CAN-Controller zu starten und zu stoppen, kann die Steuereinheit immer nur von einer Applikation geöffnet werden.

Schnittstelle öffnen

Mit Methode `IBalObject.OpenSocket` öffnen.

- Im Parameter `socketType` Typ `ICanControl` bzw. bei Kanälen mit erweiterter Funktionalität `ICanControl2` angeben.
 - ➔ Liefert die Methode eine *Exception* zurück, wird die Komponente bereits von einem anderen Programm verwendet.
- Geöffnete Steuereinheit mit Methode `IDisposable.Dispose` schließen und für Zugriff durch andere Applikationen freigeben.



Falls beim Schließen der Steuereinheit noch andere Schnittstellen des Anschlusses offen sind, bleiben die momentanen Einstellungen erhalten.

Controller-Zustände

Die Steuereinheit bzw. der CAN-Controller ist immer in einem der folgenden Zustände:

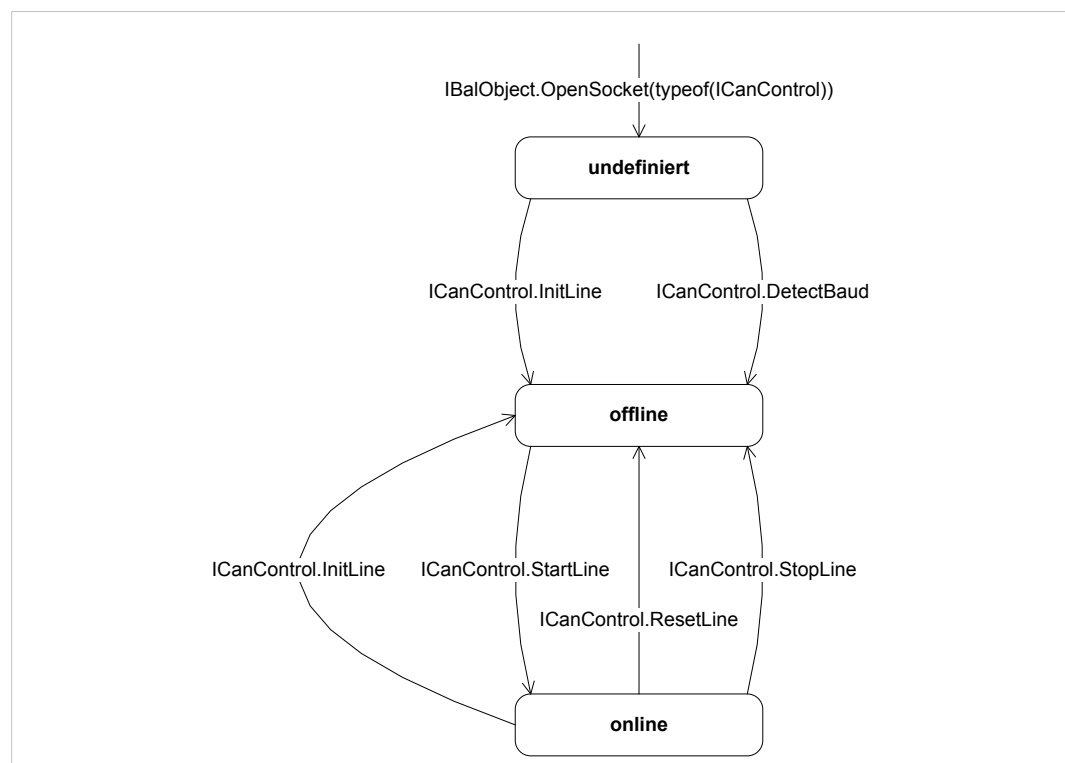


Fig. 21 Controller-Zustände

Controller initialisieren

Nach dem ersten Öffnen der Steuereinheit über die Schnittstelle `ICanControl` oder `ICanControl2` ist der Controller in undefiniertem Zustand.

- ▶ Um undefinierten Zustand zu verlassen, Methode `InitLine` oder `DetectBaud` aufrufen.
 - ➡ Controller ist im Zustand *offline*.
- ▶ Betriebsart und Bitrate des Controllers mit Methode `InitLine` einstellen.
- ▶ Betriebsart in Feld *operatingMode* einstellen.
- ▶ Bitrate in `bitrate` einstellen (siehe [Bitrate einstellen, S. 32](#)).
- ▶ Methode aufrufen.
 - ➡ Controller wird mit angegebenen Werten initialisiert.

Controller starten

Um CAN-Controller und Datenübertragung zwischen Anschluss und Bus zu starten:

- ▶ Sicherstellen, dass CAN-Controller initialisiert ist (siehe [Controller initialisieren, S. 31](#)).
- ▶ Methode `StartLine` aufrufen.
 - ➡ Steuereinheit ist im Zustand *online*.
 - ➡ Eingehende Nachrichten werden an alle geöffneten und aktiven Nachrichtenkanäle weitergeleitet.
 - ➡ Sendenachrichten werden auf den Bus übertragen.

Bei erfolgreichem Start des Controllers sendet die Steuereinheit eine Infonachricht an alle aktiven Nachrichtenkanäle. Das Property *FrameType* dieser Nachricht enthält den Wert `CanMsgFrameType.Info`, das erste Datenbyte *Data[0]* den Wert `CanMsgInfoValue.Start` und das Property *TimeStamp* den relativen Startzeitpunkt (normalerweise 0).

Controller stoppen (bzw. zurücksetzen)

- ▶ Methode `StopLine` aufrufen.
 - ➡ Controller ist im Zustand *offline*.
 - ➡ Datenübertragung zwischen Anschluss und Bus ist gestoppt.
 - ➡ Controller ist deaktiviert.
 - ➡ Eingestellte Akzeptanzfilter und Filterlisten bleiben bestehen.
 - ➡ Bei laufender Datenübertragung des Controllers wartet Funktion bis die Nachricht vollständig über den Bus gesendet ist, bevor Nachrichtentransport unterbrochen wird. Es gibt keine fehlerhaften Telegramme auf dem Bus.

oder

- ▶ Methode `ResetLine` aufrufen.
 - ➡ Controller ist im Zustand *offline*.
 - ➡ Controller-Hardware wird zurückgesetzt.
 - ➡ Nachrichtenfilter werden gelöscht.



Durch Aufruf der Methode `ResetLine` kann es zu einem fehlerhaften Nachrichtentelegramm auf dem Bus kommen, falls bei Aufruf der Methode eine Nachricht im Sendepuffer des Controllers ist, die noch nicht vollständig übertragen ist, da der Sendevorgang auch während einer laufenden Datenübertragung abgebrochen wird.

Bei Aufruf von `StopLine` und bei Aufruf von `ResetLine` sendet die Steuereinheit eine Infonachricht an alle aktiven Kanäle. Das Property *FrameType* der Nachricht enthält den Wert `CanMsgFrameType.Info`, das erste Datenbyte *Data[0]* den Wert `CanMsgInfoValue.Stop` bzw. `CanMsgInfoValue.Reset` und das Property *TimeStamp* den Wert 0. Weder `ResetLine` noch `StopLine` löschen den Inhalt der Sende-FIFOs und Empfangs-FIFOs von Nachrichtenkanälen.

Bitrate einstellen

- ▶ Mit Feldern `CanBitrate.Btr0` und `CanBitrate.Btr1` einstellen.

Die Werte der Felder `CanBitrate.Btr0` und `CanBitrate.Btr1` entsprechen den Werten für die Bus-Timing-Register BTR0 und BTR1 des Philips SJA1000 CAN-Controller bei einer Taktfrequenz von 16 MHz.

Bus-Timing-Werte mit CiA- bzw. CANopen-konformen Bitraten

Bitrate (KBit)	Vordefinierte CiA Bitraten	BTR0	BTR1
10	<code>CanBitrate.Cia10KBit</code>	0x31	0x1C
20	<code>CanBitrate.Cia20KBit</code>	0x18	0x1C
50	<code>CanBitrate.Cia50KBit</code>	0x09	0x1C
125	<code>CanBitrate.Cia125KBit</code>	0x03	0x1C
250	<code>CanBitrate.Cia250KBit</code>	0x01	0x1C
500	<code>CanBitrate.Cia500KBit</code>	0x00	0x1C
800	<code>CanBitrate.Cia800KBit</code>	0x00	0x16
1000	<code>CanBitrate.Cia1000KBit</code>	0x00	0x14
100	<code>CanBitrate._100KBit</code>	0x04	0x1C

Im Netzwerk verwendete Bitrate ermitteln

Wenn der CAN-Anschluss mit einem laufendem Netzwerk verbunden ist, bei dem die Bitrate unbekannt ist, kann die aktuelle Bitrate ermittelt werden.

Methode `DetectBaud` benötigt ein Feld mit vordefinierten Bus-Timing-Werten.

- ▶ Methode `DetectBaud` aufrufen.
- ▶ Ermittelte Bus-Timing-Werte können an `InitLine` übergeben werden.

Beispiel zur 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]);
    }
}
```

Nachrichtenfilter

Alle Steuereinheiten und Nachrichtenkanäle mit erweiterter Funktionalität haben ein zweistufiges Nachrichtenfilter. Die Datennachrichten werden ausschließlich anhand der ID (CAN-ID) gefiltert. Datenbytes werden nicht berücksichtigt.

Sendenachrichten mit gesetztem *Self-Reception-Request*-Bit werden in den Empfangspuffer eingetragen, sobald sie über den Bus gesendet sind. Der Nachrichtenfilter wird umgangen.

Betriebsarten

Nachrichtenfilter können in unterschiedlichen Betriebsarten betrieben werden:

- Sperrbetrieb (`CanFilterModes.Lock`):
Filter sperrt alle Datennachrichten, unabhängig von der ID. Verwendung z. B. wenn eine Applikation nur an Info-, Fehler- und Status-Nachrichten interessiert ist.
- Durchlassbetrieb (`CanFilterModes.Pass`):
Filter ist vollständig offen und lässt alle Datennachrichten passieren. Standardbetriebsart bei Verwendung der Schnittstelle `ICanChannel`.
- Inklusive Filterung (`CanFilterModes.Inclusive`):
Filter lässt alle Nachrichten passieren, deren IDs entweder im Akzeptanzfilter freigeschaltet oder in der Filterliste eingetragen sind (d. h. alle registrierten IDs). Standardbetriebsart bei Verwendung der Schnittstelle `ICanControl`.
- Exklusive Filterung (`CanFilterModes.Exclusive`):
Filter sperrt alle Nachrichten deren IDs entweder im Akzeptanzfilter freigeschaltet oder die in der Filterliste eingetragen sind (d. h. alle registrierten IDs).

Bei Verwendung der Schnittstelle `ICanControl` kann die Betriebsart des Filters nicht geändert werden und ist auf `CanFilterModes.Inclusive` voreingestellt. Wird die Schnittstelle `ICanControl2` bzw. `ICanChannel2` verwendet, kann die Betriebsart mit der Methode `SetFilterMode` auf eine der oben genannten Arten eingestellt werden.



Um Betriebsart des Filters abzufragen, Methode `GetFilterMode` aufrufen.

Inklusive und exklusive Betriebsart

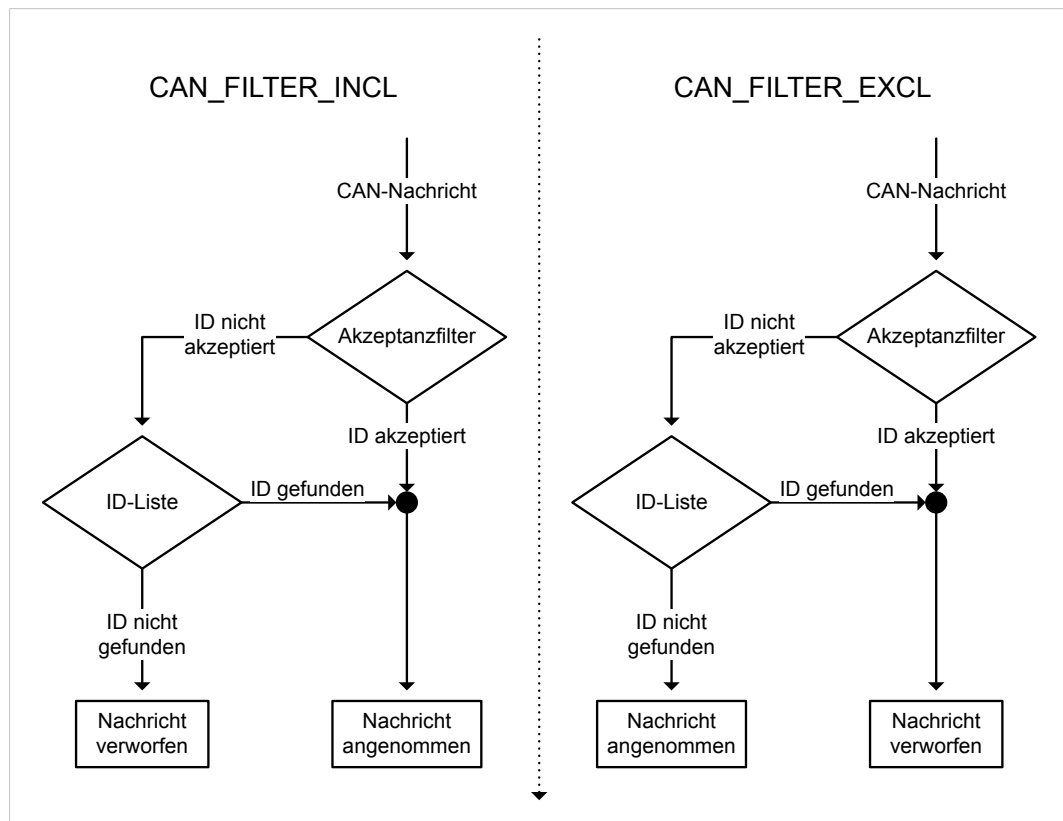


Fig. 22 Filtermechanismus bei inklusiver und exklusiver Betriebsart

Die erste Filterstufe besteht aus einem Akzeptanzfilter, der die ID einer empfangenen Nachricht mit einem binären Bitmuster vergleicht. Korreliert die ID mit dem eingestellten Bitmuster, wird die ID akzeptiert. Bei inklusiver Betriebsart wird die Nachricht angenommen. Bei exklusiver Betriebsart wird die Nachricht sofort verworfen.

Akzeptiert die erste Filterstufe die ID nicht, wird diese der zweiten Filterstufe zugeführt. Die zweite Filterstufe besteht aus einer Liste mit registrierten Nachrichten-IDs. Entspricht die ID der empfangenen Nachricht einer ID aus der Liste, wird die Nachricht bei inklusiver Filterung angenommen und bei exklusiver Filterung verworfen.

Filterkette

Jeder Nachrichtenkanal ist entweder direkt oder indirekt über einen Verteiler mit einem Anschluss verbunden (siehe [Nachrichtenkanäle](#), S. 23). Wird sowohl beim Anschluss als auch beim Nachrichtenkanal ein Filter verwendet, entsteht eine mehrstufige Filterkette. Nachrichten, die vom Anschluss ausgefiltert werden, sind für die nachgeschalteten Kanäle unsichtbar.

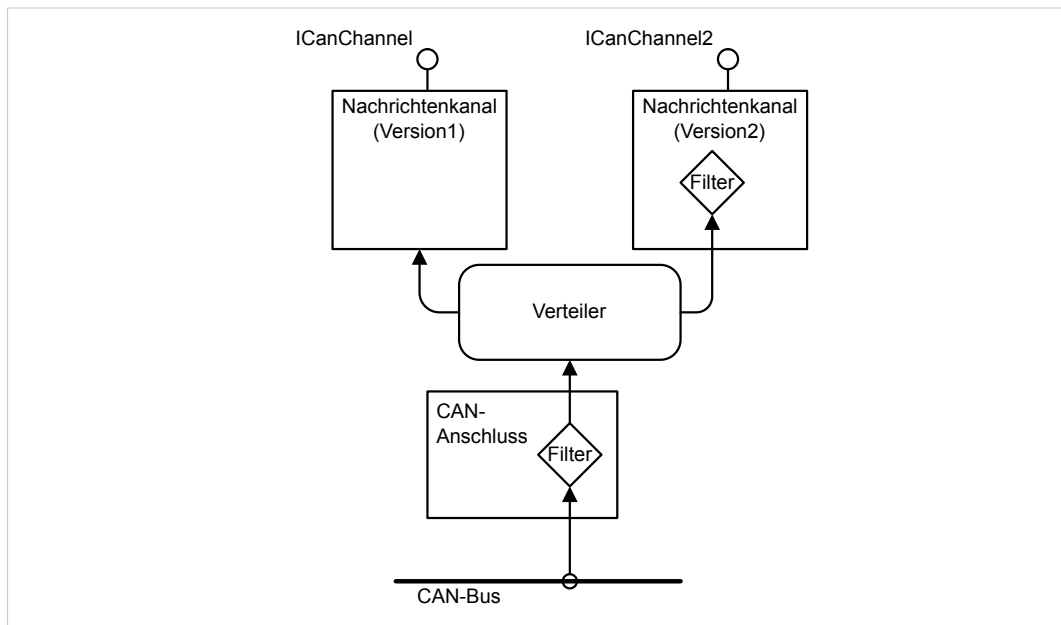


Fig. 23 Filterkette

Filter einstellen

Steuereinheiten und Nachrichtenkanäle besitzen für 11-Bit- und 29-Bit-IDs jeweils getrennte und voneinander unabhängige Filter. Nachrichten mit 11-Bit-ID werden vom 11-Bit-Filter und Nachrichten mit 29-Bit-ID vom 29-Bit-Filter gefiltert.

Zur Unterscheidung zwischen 11- und 29-Bit-Filter besitzen alle genannten Methoden den Parameter *bSelect*.



Änderungen an den Filtern während des laufenden Betriebs sind nicht möglich.



Beim Zurücksetzen oder Initialisieren des Controllers werden alle Filter so eingestellt, dass alle Nachrichten durchgelassen werden.

- Sicherstellen, dass Steuereinheit *offline* bzw. der Nachrichtenkanal inaktiv ist.

Bei Verwendung der Schnittstellen `ICanControl2` bzw. `ICanChannel2` wird die Betriebsart des Filters bereits bei der Initialisierung der Komponente voreingestellt. Der hier angegeben Wert dient der Methode `ICanControl2.ResetLine` gleichzeitig als Vorgabewert.

- Sicherstellen, dass Controller im Zustand *offline* ist.
- Um Filter nach Initialisierung einzustellen, Methode `SetFilterMode` aufrufen.
- Filter mit Methoden `SetAccFilter`, `AddFilterIds` und `RemFilterIds` einstellen.
- Im Parameter *bSelect* 11- oder 29-Bit-Filter wählen.

Die Bitmuster in den Parametern *code* und *mask* bestimmen welche IDs vom Filter durchgelassen werden.

- In Parametern *code* und *mask* zwei Bitmuster eingeben.
 - ➡ Wert von *code* bestimmt das Bitmuster der ID.
 - ➡ *mask* bestimmt welche Bits für einen Vergleich herangezogen werden.

Hat ein Bit in *mask* den Wert 0, wird das entsprechende Bit in *code* nicht für den Vergleich herangezogen. Hat es den Wert 1, ist es beim Vergleich relevant.

Beim 11-Bit-Filter werden ausschließlich die unteren 12 Bits verwendet. Beim 29-Bit-Filter werden die Bits 0 bis 29 verwendet. Alle anderen Bits des 32-Bit-Werts müssen vor Aufruf einer der Methoden auf 0 gesetzt werden.

Zusammenhang zwischen den Bits in den Parametern *code* und *mask* und den Bits der Nachrichten-ID:

11-Bit-ID-Filter

Bit	11	10	9	8	7	6	5	4	3	2	1	0
	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR

29-Bit-ID-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 1 bis 11 bzw. 1 bis 29 entsprechen den Bits 0 bis 10 bzw. 0 bis 28. Bit 0 eines jeden Wertes definiert den Wert des Remote-Transmission-Request-Bit (RTR) einer Nachricht.

Folgendes Beispiel zeigt die Werte, die für *code* und *mask* verwendet werden müssen, um Nachrichten-IDs im Bereich 100 h bis 103 h (bei denen gleichzeitig das RTR-Bit 0 sein muss) beim Filter zu registrieren:

<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

Das Beispiel zeigt, dass bei einem einfachen Akzeptanzfilter nur einzelne IDs oder Gruppen von IDs freigeschaltet werden können. Entsprechen die gewünschten Identifier nicht einem bestimmten Bitmuster, muss eine zweite Filterstufe, die Liste mit IDs, verwendet werden. Die Anzahl der IDs, die eine Liste aufnehmen kann ist konfigurierbar. Jede Liste kann bis zu 2048 IDs bzw. 4096 Einträge aufnehmen.

- ▶ Mit Methode `AddFilterIds` einzelne IDs oder Gruppen von IDs in die Liste eintragen.
- ▶ Wenn notwendig, mit Methode `RemFilterIds` von Liste entfernen.

Die Parameter *code* und *mask* haben das gleiche Format wie oben gezeigt.

Wenn Funktion `AddFilterIds` z. B. mit den Werten aus vorherigem Beispiel aufgerufen wird, trägt die Funktion die Identifier 100 h bis 103 h in die Liste ein.

- ▶ Um ausschließlich eine einzelne ID in Liste einzutragen, in *code* die gewünschte ID (einschließlich RTR-Bit) und in *mask* den Wert `FFFh` (11-Bit-ID) bzw. `3FFFFFFh` (29-Bit-ID) angeben.
- ▶ Um Akzeptanzfilter vollständig abzuschalten, bei Aufruf der Methode `SetAccFilter` in *code* den Wert `CanAccCode.None` und in *mask* den Wert `CanAccMask.None` angeben.
 - ➔ Filterung erfolgt ausschließlich mit ID-Liste.
- oder
- ▶ Akzeptanzfilter mit den Werten `CanAccCode.All` und `CanAccMask.All` konfigurieren.
 - ➔ Akzeptanzfilter akzeptiert alle IDs und ID-Liste ist wirkungslos.

6.1.4 Zyklische Sendeliste

Mit der optional vom Anschluss bereitgestellten Sendeliste lassen sich bis zu 16 Nachrichtenobjekte zyklisch senden. Der Zugriff auf diese Liste ist auf eine Applikation begrenzt und kann daher nicht von mehreren Programmen gleichzeitig genutzt werden. Es ist möglich, dass nach jedem Sendevorgang ein bestimmter Teil einer CAN-Nachricht automatisch inkrementiert wird.

Schnittstelle mit Methode `IBalObject.OpenSocket` öffnen.

- ▶ In Parameter `socketType` Typ `ICanScheduler` angeben.
 - ➔ Wenn Methode einen Fehlercode entsprechend `VciException` zurückliefert, ist die Sendeliste bereits unter Kontrolle eines anderen Programms und kann nicht erneut geöffnet werden.
 - ➔ Wenn Methode einen Fehlercode entsprechend `NotImplementedException` zurückliefert, unterstützt der CAN-Controller keine zyklische Sendeliste.
- ▶ Wenn andere Sendeliste geöffnet ist, geöffnete Sendeliste mit Methode `IDisposable.Dispose` schließen.
- ▶ Nachrichtenobjekte mit `ICanScheduler.AddMessage` bzw. bei Anschlüssen mit erweiterter Funktionalität mit `ICanScheduler2.AddMessage` zur Liste hinzufügen.
 - ➔ Bei erfolgreicher Ausführung liefert die Methode ein neues zyklisches Sendeobjekt mit der Schnittstelle `ICanCyclicTXMsg` zurück.

Ein Anschluss unterstützt ausschließlich eine Sendeliste. Die Methoden der Schnittstellen `ICanScheduler` oder `ICanScheduler2` beziehen sich deshalb auf dieselbe Liste. Da die Schnittstellen ausschließlich im Datentyp der gesendeten Nachrichten unterschiedlich sind, die Funktionsweise aber identisch ist, werden im Folgenden ausschließlich die Funktionen der Schnittstelle `ICanScheduler` beschrieben.

- ▶ Zykluszeit einer Nachricht in Anzahl Ticks im Feld `CanCyclicTXMsg.CycleTicks` angeben.
- ▶ Sicherstellen, dass angegebener Wert größer 0 ist, aber kleiner als oder gleich Wert im Feld `ICanSocket.MaxCyclicMsgTicks`.
- ▶ Dauer eines Ticks bzw. die Zykluszeit (t_z) der Sendeliste mit den Werten in den Feldern `ICanSocket.ClockFrequency` und `ICanSocket.CyclicMessageTimeDivisor` nach folgender Formel berechnen:

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

Die Sendetask der zyklischen Sendeliste unterteilt die ihr zur Verfügung stehende Zeit in einzelne Abschnitte bzw. Zeitfenster. Die Dauer eines Zeitfensters entspricht exakt der Dauer eines Ticks bzw. der Zykluszeit.

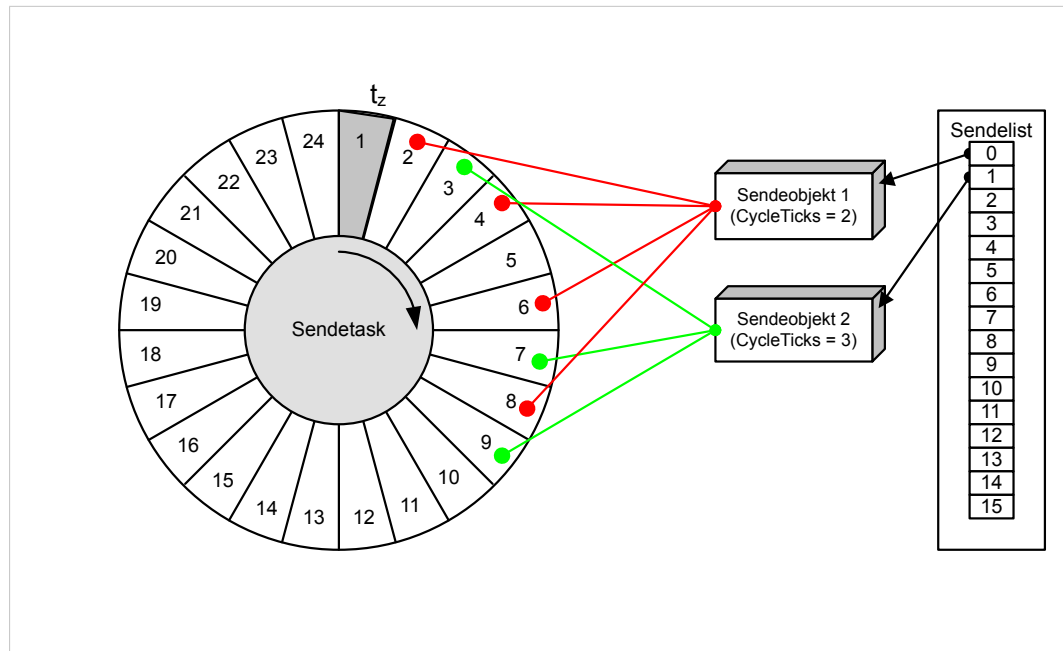


Fig. 24 Sendetask der zyklischen Sendeliste mit 24 Zeitfenstern

Die Anzahl der von der Sendetask unterstützten Zeitfenster entspricht dem Wert im Feld *ISocket.MaxCyclicMsgTicks*.

Die Sendetask kann pro Tick ausschließlich eine Nachricht senden, d. h. einem Zeitfenster kann ausschließlich ein Sendeobjekt zugeordnet werden. Wird das erste Sendeobjekt mit einer Zykluszeit von 1 angelegt, sind alle Zeitfenster belegt und es können keine weiteren Objekte eingerichtet werden. Je mehr Sendeobjekte angelegt werden, desto größer muss deren Zykluszeit gewählt werden. Die Regel lautet: Die Summe aller $1/\text{CycleTime}$ muss kleiner sein als 1.

Im Beispiel soll eine Nachricht alle 2 Ticks und eine weitere Nachricht alle 3 Ticks gesendet werden, dies ergibt $1/2 + 1/3 = 5/6 = 0,833$ und damit einen zulässigen Wert.

Beim Einrichten von Sendeobjekt 1 werden die Zeitfenster 2, 4, 6, 8, usw. belegt. Beim Einrichten vom zweiten Sendeobjekt mit einer Zykluszeit von 3 kommt es in den Zeitfenstern 6, 12, 18, usw. zu Kollisionen, da diese Zeitfenster bereits von Sendeobjekt 1 belegt sind.

Kollisionen werden aufgelöst, indem das neue Sendeobjekt in das jeweils nächste, freie Zeitfenster gelegt werden. Das Sendeobjekt 2 des obigen Beispiels besetzt dann die Zeitfenster 2, 7, 9, 13, etc. Die Zykluszeit vom zweiten Objekt wird also nicht exakt eingehalten und führt in diesem Fall zu einer Ungenauigkeit von +1 Tick.

Die zeitliche Genauigkeit mit der die einzelnen Objekte gesendet werden, hängt stark von der Nachrichtenlast auf dem Bus ab. Der exakte Sendezeitpunkt wird mit steigender Last unpräziser. Generell gilt, dass die Genauigkeit mit steigender Bus-Last, kleineren Zykluszeiten und steigender Anzahl von Sendeobjekte abnimmt.

Das Feld *CanCyclicTXMsg.AutoIncrementMode* der Struktur bestimmt, ob bestimmte Teile der Nachricht nach dem Senden automatisch erhöht werden oder unverändert bleiben.

Wird der Wert *CanCyclicTXIncMode.NoInc* angegeben, bleibt der Inhalt der Nachricht unverändert. Beim Wert *CanCyclicTXIncMode.IncId* wird das Feld *Identifier* der Nachricht nach jedem Senden 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.

Bei den Werten *CanCyclicTXIncMode.Inc8* bzw. *CanCyclicTXIncMode.Inc16* im Feld *CanCyclicTXMsg.AutoIncrementMode* wird ein einzelner 8-Bit- bzw. 16-Bit-Wert im

Datenfeld der Nachricht nach jedem Senden inkrementiert. Das Feld *AutoIncrementIndex* bestimmt den Index des Datenfelds.

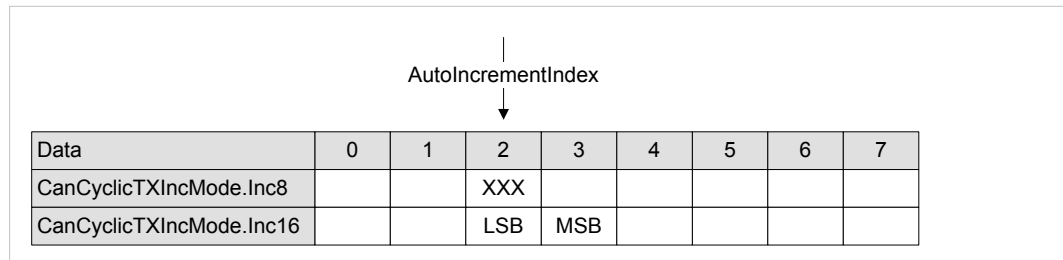


Fig. 25 Auto-Inkrement von Datenfeldern

Bei 16-Bit Werten liegt das niederwertige Byte (LSB) im Feld *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.

- ▶ Wenn notwendig, Sendeobjekt mit Methode `RemoveMessage` von Liste entfernen. Die Methode erwartet den von `AddMessage` gelieferten Listenindex des zu entfernenden Objekts.
- ▶ Um neu erstelltes Sendeobjekt zu senden, Methode `SendMessage` aufrufen.
- ▶ Bei Bedarf Sendevorgang mit Methode `StopMessage` stoppen.

Den momentanen Zustand eines einzelnen Sendeobjekts liefert das Property `Status`. Die Sendeobjektstatus werden durch Methode `UpdateStatus` aktualisiert.

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

- ▶ Zum gleichzeitigen Starten aller Sendeobjekte, alle Sendeobjekte mit Methode `SendMessage` starten.
- ▶ Um Sendetask zu aktivieren oder zu deaktivieren, Methode `Resume` aufrufen.
- ▶ Zum gleichzeitigen Stoppen aller Sendeobjekte, Methode `Suspend` aufrufen.
- ▶ Um Sendetask zurückzusetzen, Methode `Reset` aufrufen.
 - ➔ Sendetask wird gestoppt.
 - ➔ Alle nicht registrierten Sendeobjekte werden aus der angegebenen zyklischen Sendeliste entfernt.

6.2 LIN-Anschluss

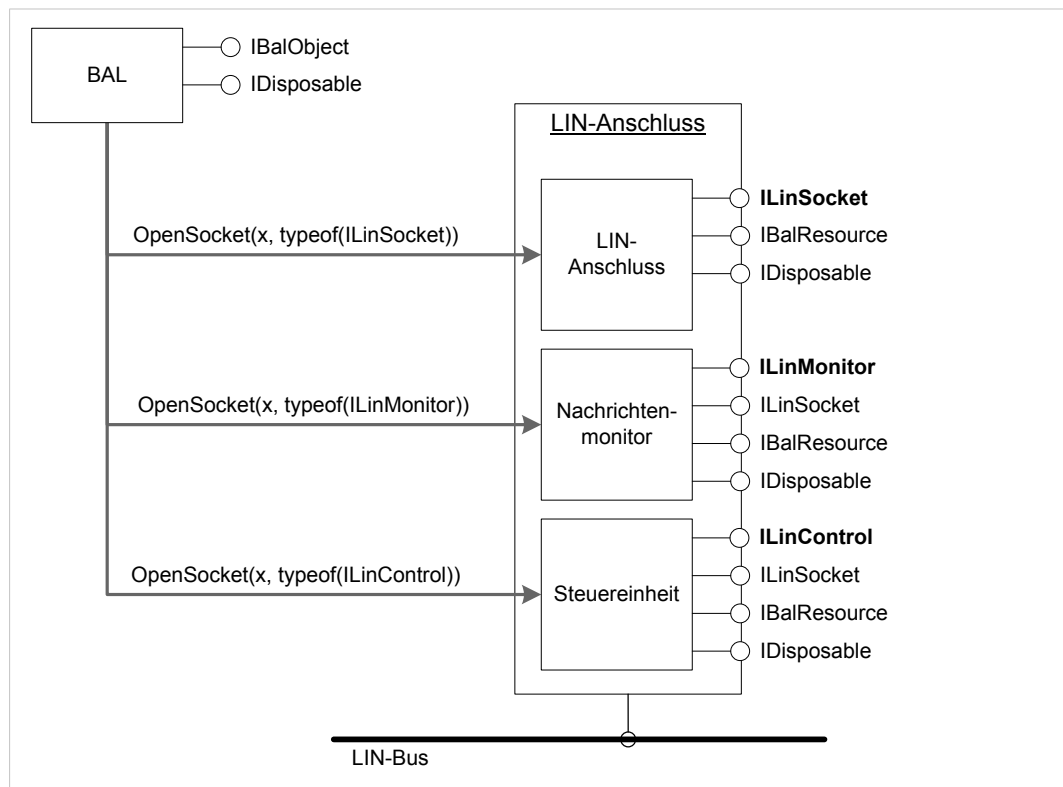


Fig. 26 Komponenten LIN-Anschluss

Zugriff auf einzelne Teilkomponenten über Schnittstellen **ILinSocket**, **ILinMonitor** und **ILinControl**.

ILinSocket (siehe [Socket-Schnittstelle, S. 42](#)) bietet folgende Funktionen:

- Abfrage der LIN-Controller-Eigenschaften
- Abfrage des aktuellen Controllerzustands

ILinMonitor (siehe [Nachrichtenmonitore, S. 42](#)):

- repräsentiert den Nachrichtenmonitor
- ein oder mehrere Nachrichtenmonitore pro LIN-Anschluss möglich
- LIN-Nachrichten werden ausschließlich von Nachrichtenmonitoren empfangen.

ILinControl (siehe [Steuereinheit, S. 45](#)) bietet folgende Funktionen:

- Konfiguration des LIN-Controllers
- Konfiguration der Übertragungseigenschaften
- Abfrage des aktuellen Controllerzustands

6.2.1 Socket-Schnittstelle

Die Schnittstelle `ILinSocket` unterliegt keinen Zugriffsbeschränkungen und kann gleichzeitig von verschiedenen Programmen geöffnet werden. Die Steuerung des Anschlusses ist über diese Schnittstelle nicht möglich.

Mit Methode `IBalObject.OpenSocket` öffnen.

- Im Parameter `socketType` Typ `ILinSocket` angeben.

Die Eigenschaften des LIN-Controllers, wie beispielsweise unterstützte Funktionen sind durch Properties bereitgestellt.

- Um aktuelle Betriebsart und Zustand des Controllers zu ermitteln, Property `LineStatus` aufrufen.

6.2.2 Nachrichtenmonitore

Ein LIN-Nachrichtenmonitor besteht aus einem Empfangs-FIFO.

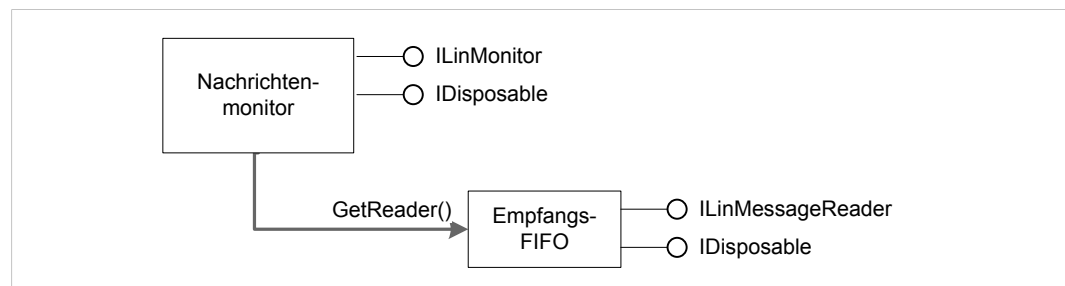


Fig. 27 Komponenten LIN-Nachrichtenmonitor

Die Funktionsweise eines Nachrichtenmonitors ist unabhängig davon, ob der Anschluss exklusiv verwendet wird oder nicht.

Bei exklusiver Verwendung des Anschlusses ist der Nachrichtenmonitor direkt mit dem LIN-Controller verbunden.

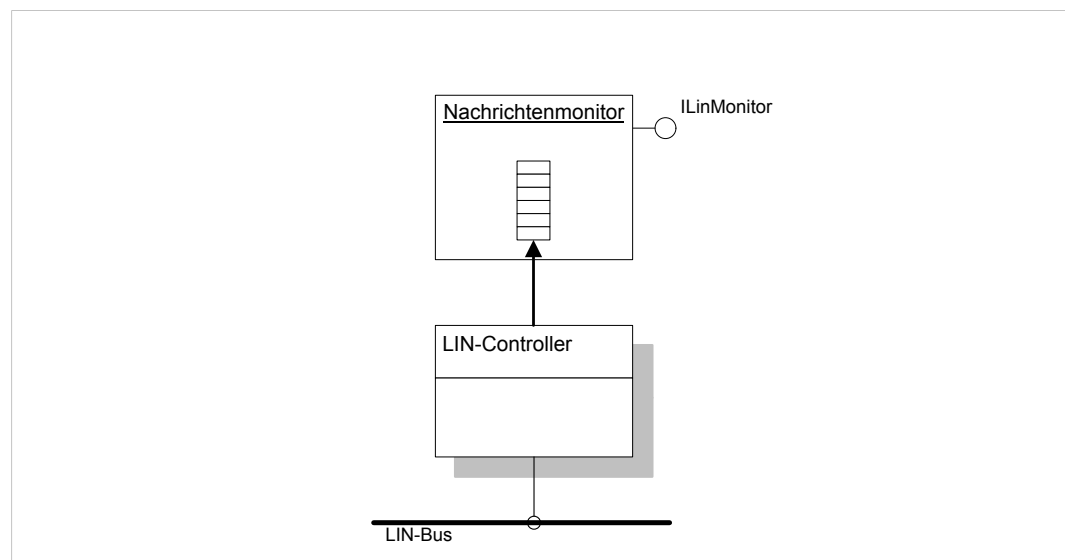


Fig. 28 Exklusive Verwendung

Bei nicht-exklusiver Verwendung des Anschlusses sind die Nachrichtenmonitore über einen Verteiler mit dem LIN-Controller verbunden. Der Verteiler leitet alle beim LIN-Controller eintreffenden Nachrichten an alle Nachrichtenmonitore weiter. Kein Monitor wird priorisiert, d. h. der

vom Verteiler verwendete Algorithmus ist so gestaltet, dass alle Monitore möglichst gleichberechtigt behandelt werden.

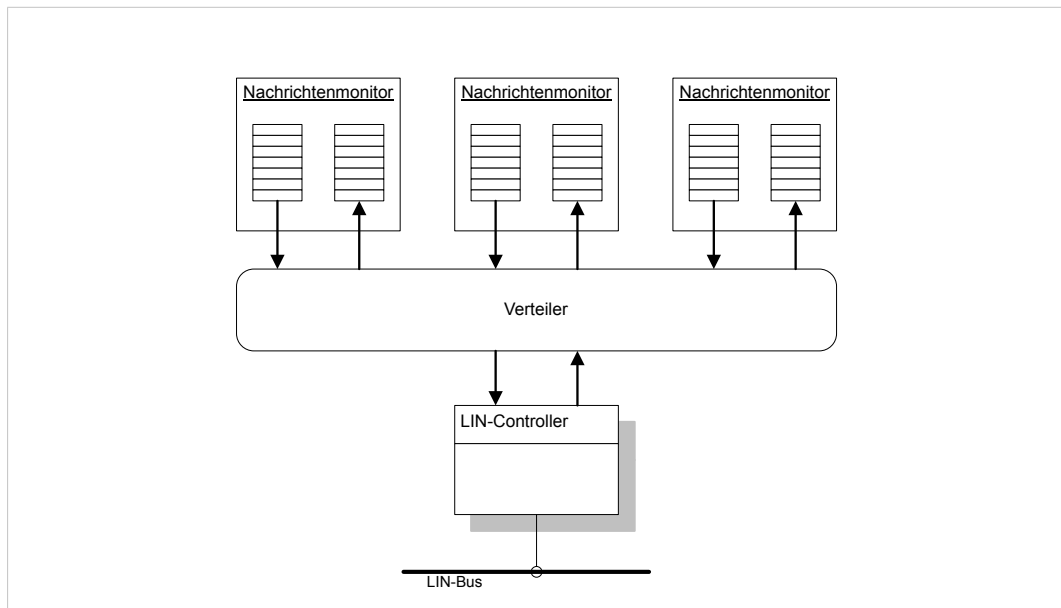


Fig. 29 Nicht-exklusive Verwendung (mit Verteiler)

Nachrichtenmonitor erstellen

Nachrichtenmonitor mit Methode `IBalObject.OpenSocket` erstellen.

- ▶ In Parameter `socketType` Typ `ILinMonitor` angeben.
 - ▶ Um Controller exklusiv zu verwenden (nach erfolgreicher Ausführung können keine weiteren Nachrichtenmonitore verwendet werden), Wert `TRUE` in Parameter `exclusive` angeben.
- oder

Um Controller nicht-exklusiv zu verwenden (Erstellung beliebig vieler Nachrichtenmonitore möglich), Wert `FALSE` in Parameter `exclusive` angeben.

Nachrichtenmonitor initialisieren

Ein neu erstellter Monitor besitzt keinen Empfangs-FIFO.

- ▶ Mit Methode `ILinMonitor.Initialize` Nachrichtenmonitor initialisieren und Empfangs-FIFO erstellen.
- ▶ In Eingabeparametern Größe des Empfangs-FIFOs in Anzahl LIN-Nachrichten bestimmen.

Nachrichtenmonitor aktivieren

Ein neu erstellter Monitor ist deaktiviert. Nachrichten werden vom Bus ausschließlich empfangen, wenn der Nachrichtenmonitor aktiv und der LIN-Controller gestartet ist. Weitere Informationen zum LIN-Controller siehe Kapitel [Steuereinheit, S. 45](#).

- ▶ Nachrichtenmonitor mit Methode `ILinMonitor.Activate` aktivieren.
- ▶ Aktiven Monitor mit Methode `ILinMonitor.Deactivate` trennen.

LIN-Nachrichten empfangen

- ▶ Zum Lesen erforderliche Schnittstelle `ILinMessageReader` mit Methode `ILinMonitor.GetMessageReader` anfordern.

Nachrichten aus dem FIFO lesen:

- ▶ Methode `ReadMessage` aufrufen.
oder
- ▶ Um mehrere LIN-Nachrichten über einen Aufruf zu lesen (optimiert auf hohen Datendurchsatz), Feld von LIN-Nachrichten anlegen.
- ▶ Feld an Methode `ReadMessages` übergeben.
 - ➡ `ReadMessages` versucht Feld mit empfangenen Daten zu füllen.
 - ➡ Anzahl tatsächlich gelesener Nachrichten wird über Rückgabewert signalisiert.

Mögliche Verwendung von `ReadMessage`

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

Mögliche Verwendung von `ReadMessages`

```
void DoMessages( ILinMessageReader reader )
{
    ILinMessage[] messages;

    int readCount = reader.ReadMessages(out messages);
    for( int idx = 0; idx < readCount; idx++ )
    {
        // Verarbeitung der Nachricht
    }
}
```

6.2.3 Steuereinheit

Die Steuereinheit kann ausschließlich von einer Applikation geöffnet werden. Gleichzeitiges, mehrfaches Öffnen der Schnittstelle durch mehrere Programme ist nicht möglich.

Schnittstelle öffnen

Mit Methode `IBalObject.OpenSocket` öffnen.

- ▶ Im Parameter *socketType* Typ `ILinControl` angeben.
 - ➡ Liefert die Methode eine *Exception* zurück, wird die Komponente bereits von einem anderen Programm verwendet.
- ▶ Geöffnete Steuereinheit mit Methode `IDisposable.Dispose` schließen und für Zugriff durch andere Applikationen freigeben.



Falls beim Schließen der Steuereinheit noch andere Schnittstellen des Anschlusses offen sind, bleiben die momentanen Einstellungen erhalten.

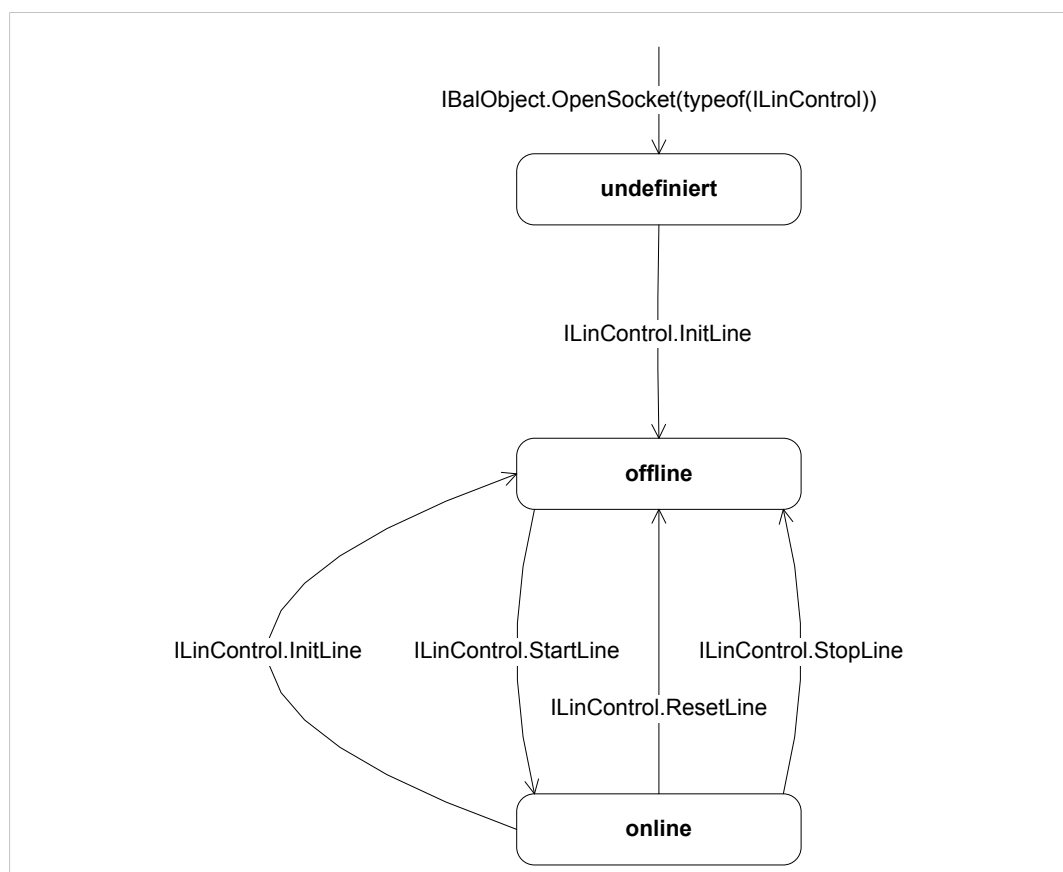


Fig. 30 LIN-Controller-Zustände

Controller initialisieren

Nach dem ersten Öffnen der Schnittstelle `ILinControl` ist der Controller in undefiniertem Zustand.

- ▶ Um undefinierten Zustand zu verlassen Methode `InitLine` aufrufen.
 - ➡ Controller ist im Zustand *offline*.
- ▶ Betriebsart und Datenübertragungsrate mit Methode `InitLine` einstellen.
 - ▶ Methode erwartet Struktur `LinInitLine` mit Werten für Betriebsart und Bitrate.
- ▶ Datenübertragungsrate in Bits pro Sekunde im Feld `LinInitLine.Bitrate` angeben.
 Gültige Werte liegen zwischen 1000 und 20000 Bit/s, bzw. zwischen `LinBitrate.MinBitrate` und `LinBitrate.MaxBitrate`.

Wenn der Controller automatische Bitraten-Erkennung unterstützt, kann die automatische Bitraten-Erkennung mit `LinBitrate.AutoRate` aktiviert werden.

Empfohlene Bitraten:

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

Controller starten und stoppen

- ▶ Um LIN-Controller zu starten, Methode `StartLine` aufrufen.
 - ➡ LIN-Controller ist im Zustand *online*.
 - ➡ LIN-Controller ist aktiv mit dem Bus verbunden.
 - ➡ Eingehende Nachrichten werden an alle geöffneten und aktiven Nachrichtenmonitore weitergeleitet.
- ▶ Um LIN-Controller zu stoppen, Methode `StopLine` aufrufen.
 - ➡ LIN-Controller ist im Zustand *offline*.
 - ➡ Nachrichtentransport zu den Monitoren ist unterbrochen und der Controller deaktiviert.
 - ➡ Bei laufender Datenübertragung des Controllers wartet Methode bis die Nachricht vollständig über den Bus gesendet ist, bevor der Nachrichtentransport unterbrochen wird.
- ▶ Methode `ResetLine` aufrufen, um Controller in Status *offline* zu bringen und Controller-Hardware zurückzusetzen.



Durch Aufruf der Methode `ResetLine` kann es zu einem fehlerhaften Nachrichtentelegramm auf dem Bus kommen, falls dabei ein laufender Sendevorgang abgebrochen wird.

Weder `ResetLine` noch `StopLine` löschen den Inhalt der Empfangs-FIFOs von Nachrichtenmonitoren.

LIN-Nachrichten senden

Nachrichten können mit der Methode `ILinControl.WriteMessage` direkt gesendet oder in eine Antworttabelle im Controller eingetragen werden.

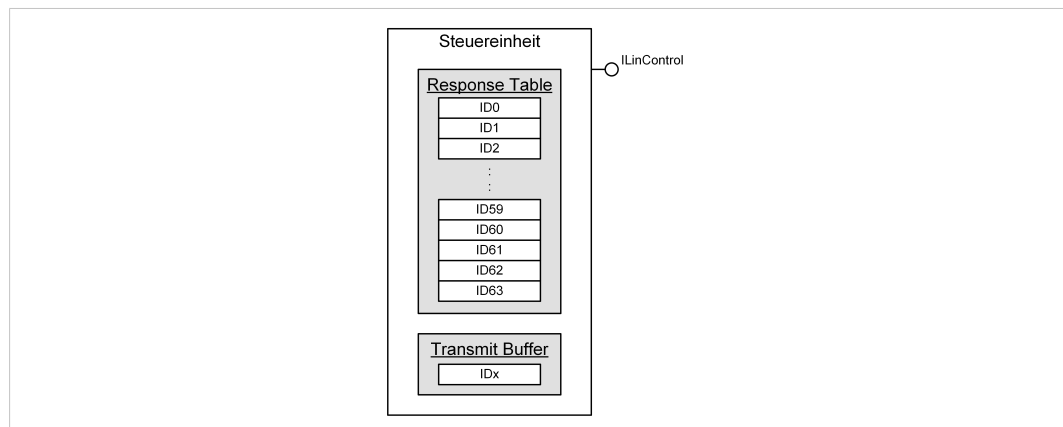


Fig. 31 Interner Aufbau einer Steuereinheit

Die Steuereinheit enthält eine interne Antworttabelle (Response Table) mit den jeweiligen Antwortdaten für die vom Master aufgeschalteten IDs. Erkennt der Controller eine ihm zugeordnete und vom Master gesendete ID, überträgt er die, in der Tabelle an entsprechender Position eingetragenen Antwortdaten automatisch auf dem Bus.

Um Antworttabelle zu ändern oder zu aktualisieren, Methode `ILinControl.WriteMessage` aufrufen.

- Im Parameter *send* den Wert `FALSE` eingeben.
 - ➡ Nachricht mit Antwortdaten im Datenfeld der Struktur `LinMessage` wird der Methode im Parameter *message* übergeben.
- Um Antworttabelle zu leeren, Methode `ILinControl.ResetLine` aufrufen.

Datenfeld der Struktur `LinMessage` enthält die Antwortdaten. Die LIN-Nachricht muss vom Typ `LinMessageType.Data` sein und eine ID im Bereich 0 bis 63 enthalten.

Unabhängig von der Betriebsart (Master oder Slave) muss die Tabelle vor dem Start des Controllers initialisiert werden. Sie kann danach jederzeit aktualisiert werden, ohne dass der Controller gestoppt werden muss.

Mit Methode `ILinControl.WriteMessage` Nachrichten direkt auf Bus senden.

- Parameter *send* auf Wert `TRUE` setzen.
 - ➡ Nachricht wird in Sendepuffer des Controllers eingetragen, statt in die Antworttabelle.
 - ➡ Controller sendet Nachricht auf den Bus, sobald dieser frei ist.

Wenn der Controller als Master konfiguriert ist, können die Steuernachrichten `LinMessageType.Sleep`, `LinMessageType.Wakeup` and `LinMessageType.Data` direkt gesendet werden. Wenn der Controller als Slave konfiguriert ist, können ausschließlich `LinMessageType.Wakeup` Nachrichten direkt gesendet werden. Bei allen anderen Nachrichtentypen liefert die Methode einen Fehlercode zurück.

Eine Nachricht vom Typ `LINMessageType.Sleep` erzeugt ein Go-to-Sleep-Frame, eine Nachricht vom Typ `LINMessageType.Wakeup` einen Wake-Up-Frame auf dem Bus. Für weitere Informationen siehe Kapitel Network Management in den LIN-Spezifikationen.

In der Master-Betriebsart dient die Methode `ILinControl.WriteMessage` auch zum Umschalten von IDs. Hierzu wird eine Nachricht vom Typ `LINMessageType.Data` mit gültiger ID und Datenlänge gesendet, bei der das Flag *IdOnly* auf `TRUE` gesetzt ist.

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

7 Schnittstellenbeschreibung

Für detaillierte Beschreibung der VCI .NET-Schnittstellen und Klassen siehe mitinstallierte Ordnerreferenz *vci4net.chm* im Unterverzeichnis *manual*.

