IXXAT®

VCI: .NET-API

Software Version 4

SOFTWARE DESIGN GUIDE

4.02.0250.20021 1.0 ENGLISH

HMS Connecting Devices™

Important User Information

Liability

Every care has been taken in the preparation of this document. Please inform HMS Industrial Networks of any inaccuracies or omissions. The data and illustrations found in this document are not binding. We, HMS Industrial Networks, reserve the right to modify our products in line with our policy of continuous product development. The information in this document is subject to change without notice and therefore should not be considered as a binding description of the range of functions (neither for future product versions). HMS Industrial Networks assumes no responsibility for any errors that may appear in this document.

There are many applications of the described product. Those responsible for the use of this device must ensure that all the necessary steps have been taken to verify that the applications meet all performance and safety requirements including any applicable laws, regulations, codes, and standards.

HMS Industrial Networks will under no circumstances assume liability or responsibility for any problems that may arise as a result from improper use or use that is not in accordance with the documented features of this product.

The examples and illustrations in this document are included solely for illustrative purposes.

Intellectual Property Rights

HMS Industrial Networks has intellectual property rights relating to technology embodied in the product described in this document. These intellectual property rights may include patents and pending patent applications in the USA and other countries.

Trademark Acknowledgements

Ixxat® is a registered trademark of HMS Industrial Networks. All other trademarks are the property of their respective holders.

VCI: .NET-API Software Version 4 Software Design Guide
4.02.0250.20021 1.0

Table of Contents

Page

This page intentionally left blank

1 User Guide

Please read the manual carefully. Make sure you fully understand the manual before using the product.

1.1 Document History

Version	Date	Author	Description
1.0	July 2016	CoMi	First release

1.2 Conventions

Instructions and results are structured as follows:

- ▶ instruction 1
- ▶ instruction 2
  - ▷ result 1
  - ▷ result 2

Lists are structured as follows:

- item 1
- item 2

Bold typeface indicates interactive parts such as connectors and switches on the hardware, or menus and buttons in a graphical user interface.

```
This font is used to indicate program code and other
kinds of data input/output such as configuration scripts.
```

This is a cross-reference within this document: *Conventions, p. 3*

This is an external link (URL): www.hms-networks.com

This is additional information which may facilitate installation and/or operation.

! This instruction must be followed to avoid a risk of reduced functionality and/or damage to the equipment, or to avoid a network security risk.

1.3 Glossary

Abbreviations	
VCI	Virtual Communication Interface
VCI server	VCI system service
FIFO	First In/First Out Memory
BAL	Bus Access Layer
VCIID	system-wide unique ID of a device
GUID	Unique ID of device class
API	Application Programming Interface

2 System Overview

The VCI (Virtual Communication Interface) is a driver that provides common access to the different devices by HMS Industrial Networks for applications.

The VCI .Net adapter is based on the VCI, that provides an interface based C++ API.

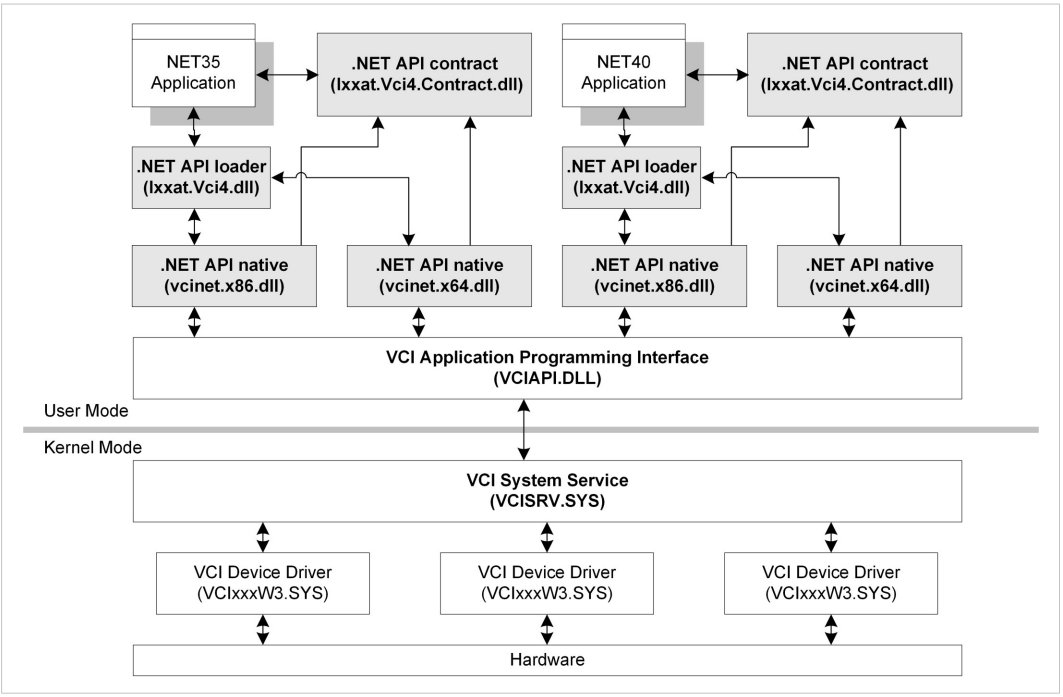


Fig. 1 System components

The VCI consists of the following components:

- VCISRV.SYS: kernel mode system service
- VCIxxxWy.SYS: one or more VCI device drivers
- VCINET.DLL: .NET programming interface

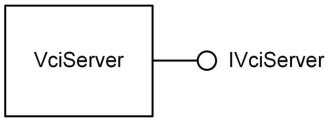
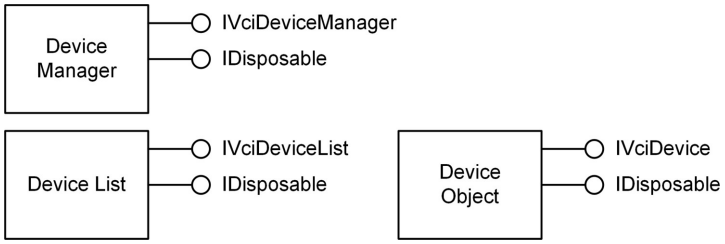
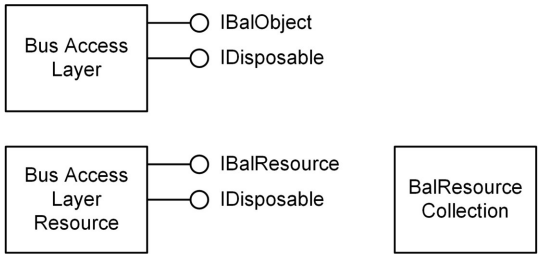
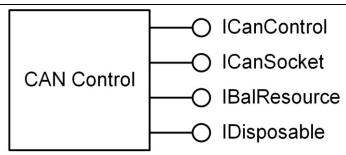
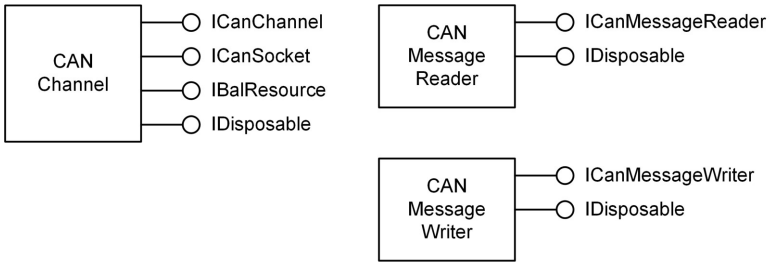
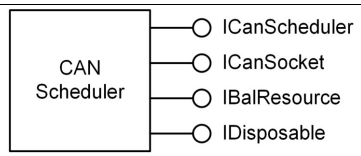
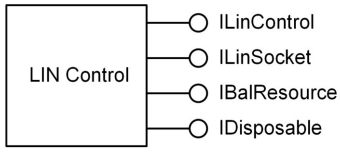
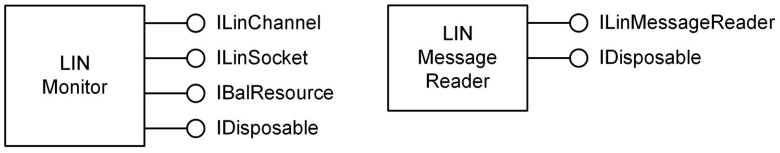
The following further components are described in other manuals:

- VCINPL.DLL: C programming interface (not for CAN FD)
- VCILVA.DLL: LabView adapter (not for CAN FD), allows access to VCI with LabView
- VCIAPI.DLL: user mode programming interface (C++ API)

The VCI system service (VCI server) takes over the following tasks:

- management of VCI specific device drivers
- management of the access to the device
- providing of mechanisms for the exchange of data and commands between application and operating system or between user and kernel mode

Sub-components and .NET interfaces/classes of the VCI .NET API programming interface:

Entry point	
Device management and device access	
Bus access	
CAN control	
CAN message channels	
Cyclic transmitting list	
LIN control	
LIN message monitor	

3 Including .NET API

The VCI .NET API contains a set of .NET assemblies for .NET 3.5 and for .NET 4.0 and higher, located in the corresponding subdirectories NET35 and NET40. If included via NuGet the correct version is copied to the project directory.

Except for the dependencies to corresponding system assemblies, the assemblies have the same functionality.

Components of the VCI .NET Adapter:

- *Ixxat.Vci4.Contract.dll*: Contains basic class and interface declarations, defines interface (contract) between VCI .NET adapter and application.
- *Ixxat.Vci4.dll*: Contains minimal loader, which loads depending on the process architecture in use the corresponding native component (*vcinet.x86.dll* or *vcinet.x64.dll*). Simplifies the deployment of applications, that are compiled independently of the architecture (AnyCPU).
- *vcinet.x86.dll*: native component for x86 systems:
- *vcinet.x64.dll*: native component for x64 systems:

Differences to VCI .NET API Version 3:

- simplifies deployment and reduces dependencies between various interfaces, since there is no installation within GAC
- different declarations due to the moving of the interfaces in *Ixxat.Vci4.Contract.dll* and the implementation of the loader *Ixxat.Vci4.dll*
- additional interfaces `ICanChannel2`, `ICanSocket2`, `ICanScheduler2`, `ICanMessage2` and value types*CanBitrate2*, *CanFdBitrate* and *CanLineStatus2* for CAN FD support

3.1 Including Manually into Own Projects

▶ Add dependencies to project.

Ixxat.Vci4.Contact.dll and loader *Ixxat.Vci4.dll* are necessary.

▶ Copy native components (*vcinet.x86.dll* and *vcinet.x64.dll*) to bin directory.

3.2 Including into Own Projects via NuGet

The including via NuGet automates the steps that are necessary when including manually.

▶ Instal package *Ixxat.Vci4* for the project.

▶ Observe further information in manuals (in package *Ixxat.Vci4.Manuals*) and on www.nuget.org.

3.3 Porting Applications

The VCIAPI.DLL of the VCI 4 is binary compatible to the VCI 3.

- To run functions that used the former VCI .NET API version 3 on VCI 4, install VCI .NET API Version 3.
- Use setup on www.ixxat.com/support.
- To port applications of VCI 3 .NET API to current VCI 4 .NET adapter, change the following sources:
  - using statements
  - access to device manager
  - use of CAN/LIN messages
  - requesting the channel status

Using Statements

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

Access to Device Manager

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

Use of CAN/LIN Messages (Transmit)

Because of the abstraction of messages via interfaces the use of a factory class is necessary:

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

Use of CAN/LIN Messages (Receive)

Exclusively the declaration is affected.

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

Requesting the Channel Status

The change of the implementation of the LineStatus (to distinguish uninitialized statuses) eventually makes adaptations of the access to this objects necessary.

4 Device Management and Device Access

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

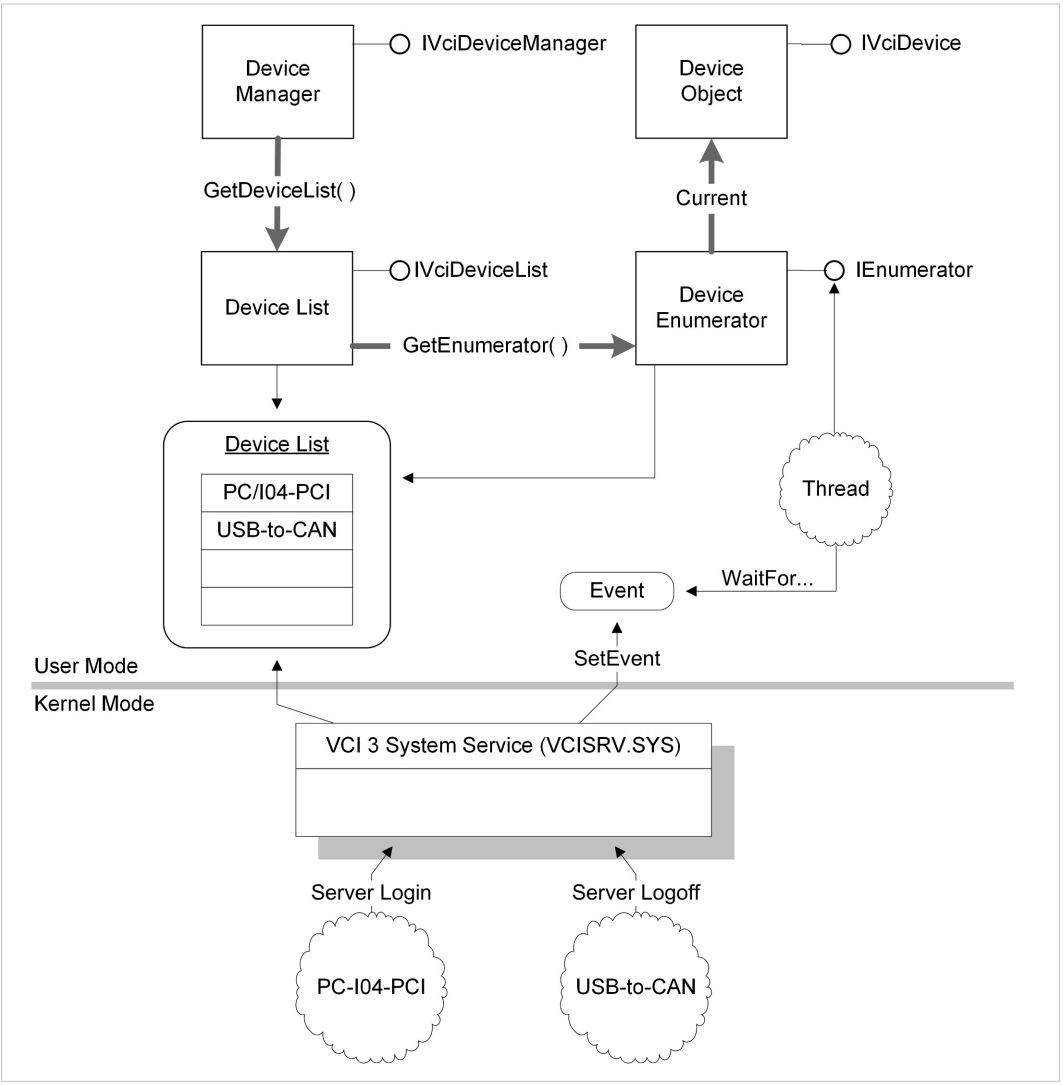


Fig. 2 Device management components

The VCI server manages all devices in a system-wide global device list. When the computer is booted or a connection between device and computer is established the device is automatically logged into the server. If a device is no longer available e. g. because the connection is interrupted, the device is automatically removed from the device list.

The logged in devices are accessed via the VCI device manager or its interface `IVciDeviceManager`. The property `VciServer.DeviceManager` returns a reference to this interface.

Main information of a device:

Interface	Type	Description
Description	String with description of the interface	e. g. USB-to-CAN compact
VciObjectId	Unique ID of device	When a device logs in, it is allocated a system-wide unique ID (VCIID). This ID is required for later access to the device.
DeviceClass	Device class	All device drivers identify their supported device class by a worldwide unique ID (GUID). Different devices belong to different device classes, e. g. the USB-to-CAN belongs to a different device class as PC-I04/PCI.
UniqueHardwareId	Hardware ID	All devices have a unique hardware ID. The ID can be used to differentiate between two interfaces or to search for a device with a certain hardware ID. Remains after restart of the system. Because of that it can be stored in the configuration file and enables automatic configuration of the application after program and system start.
DriverVersion	Version number of driver	
HardwareVersion	Version number of interface	
Equipment	Technical equipment of interface	Included table of VciCtrlInfo structures provides information about number and type of bus connections present on the interface. Table entry 0 describes bus connection 1, table entry 1 the bus connection 2 etc.

4.1 Listing Available Devices

- ▶ To access global device list, call method `IVciDeviceManager.GetDeviceList`.
  - ▷ Returns pointer to interface `IVciDeviceList` of device list.

It is possible to monitor changes in the device list and to request enumerators for the device list. There are different possibilities to navigate in the device list.

Requesting Enumerators

Method `IVciDeviceList.GetEnumerator` returns `IEnumerator` interface of a new enumerator object for the device list.

- ▶ Call method `IEnumerator.Current`.
  - ▷ Returns a new device object with information of the interface with each call.
- ▶ To access the information of the property `Current` of the standard interface `IEnumerator`, convert the provided pure object reference to `IVciDevice` type.
- ▶ To increment internal index, call method `IEnumerator.MoveNext` .
  - ▷ `IEnumerator.Current` returns device object for the next interface.

List is completely passed when method `IEnumerator.MoveNext` returns value `FALSE`.

Resetting Internal List Index

▶ Call method `IEnumerator.Reset`.

▷ The following call of method `IEnumerator.MoveNext` returns again information about the first device in the device list.

Devices that can be added or removed during operation, like e. g. USB devices are logged in with connecting and logged off with disconnecting. The devices are also logged in or off when the operation system activates or deactivates a device driver in the device manager.

Monitoring Changes in the Device List

▶ Create `AutoResetEvent` object or `ManualResetEvent` object.

▶ Assign object to list with `IVciDeviceList.AssignEvent`.

Use `AutoResetEvent` to set event in signalized state, when a device logs in or off the VCI server after calling the method.

4.2 Accessing Individual Devices

All Ixxat interfaces provide one or more components resp. access levels for various application areas. Here the Bus Access Layer (BAL) is relevant. The BAL allows the control of the controller and the communication with the fieldbus.

The different access levels of a Ixxat interface cannot be opened simultaneously. For example, if an application opens the BAL, the access level used by CANopen master API can only be opened again after the BAL is released or closed.

Certain access levels are additionally protected against multiple opening, e. g. two CANopen applications cannot use one Ixxat interface simultaneously.

The BAL can be opened by several programs simultaneously, to allow different applications the simultaneous access to different bus connections (further information see *Accessing the Bus Controller, p. 18*).

5 Communication Components

5.1 First In/First Out Memory (FIFO)

The VCI contains an implementation for First In/First Out memory objects.

FIFO features:

- Dual-port memory, in which data is written on the input side and read on the output side.
- Chronological sequence is preserved, i. e. data that is written in the FIFO at first is also read at first.
- Similar to the functionality of a pipe connection and therefore also named pipe.
- Used to transfer data from a transmitter to the parallel receiver. Agreement with a lock mechanism, that has access to the common memory area at a certain point of time is not necessary.
- No locking, possible to be overcrowded, if receiver does not manage to read the data in time.
- Transmitter writes the messages to transmit with writer interface in the FIFO. Receiver parallel reads data with reader interface.

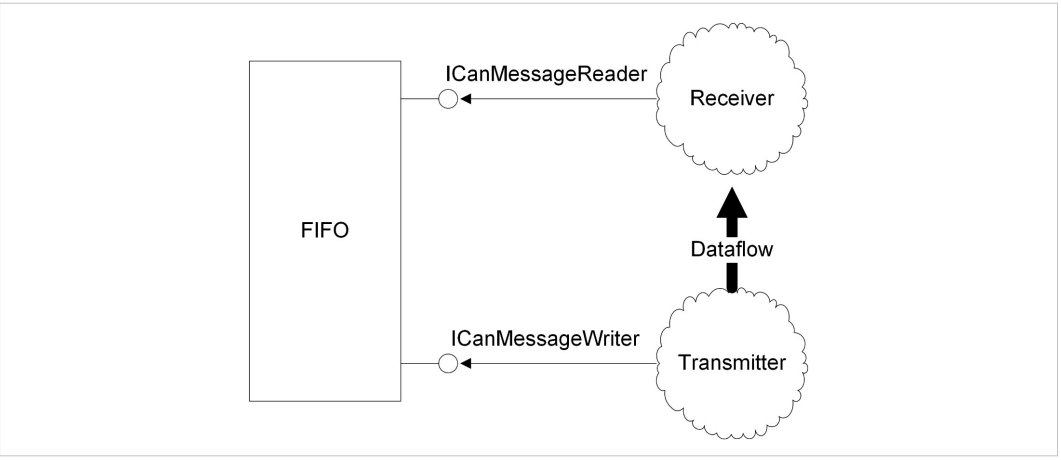


Fig. 3 FIFO data flow

Access:

- Writing and reading access to a FIFO is possible simultaneously, a receiver is able to read data while a transmitter writes new data to the FIFO.
- Simultaneous access of several transmitters resp. receivers to the FIFO is not possible.
- Multiple access to interfaces `ICanMessageReader` and `ICanMessageWriter` is prevented, because the respective interface of the FIFO can only be opened once, i. e. not until the interface is released with `IDisposable.Dispose` it can be opened again.
- To prevent simultaneous access to one interface by different threads of an application:
  - Make sure, that methods of an interface can only be called by one thread of the application (e. g. create a separate message channel for the second thread).

    or
  - Synchronize access to an interface with respective thread: Call method `Lock` before every access of the FIFO and method `Unlock` of the respective interface after accessing.

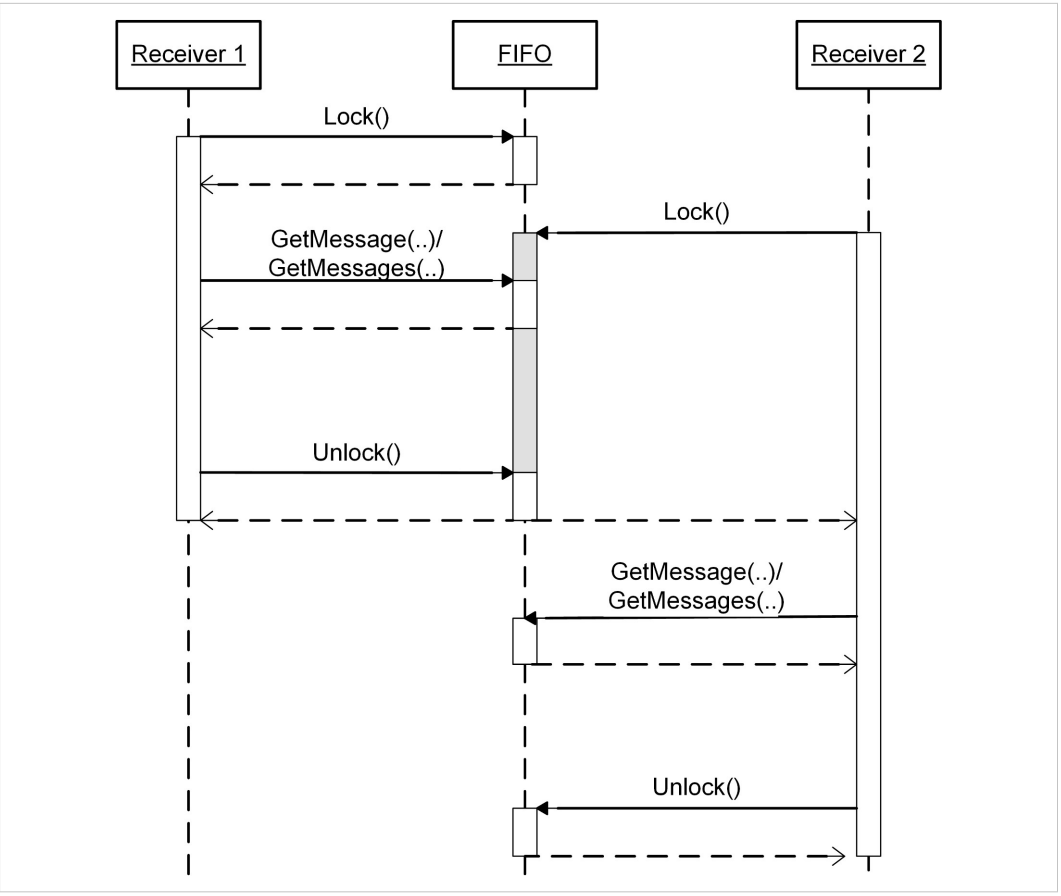


Fig. 4 FIFO locking mechanism

Receiver 1 calls method `Lock` and gains access to the FIFO. The following call of `Lock` by receiver 2 is blocked as long as receiver 1 releases the FIFO with calling method `Unlock` . Now receiver 2 can start processing. In the same way two transmitters that access the FIFO with the interface `ICanMessageWriter` can be synchronized.

The FIFOs provided by the VCI also allow the exchange of data between two processes, i. e. over the boundaries of the process.

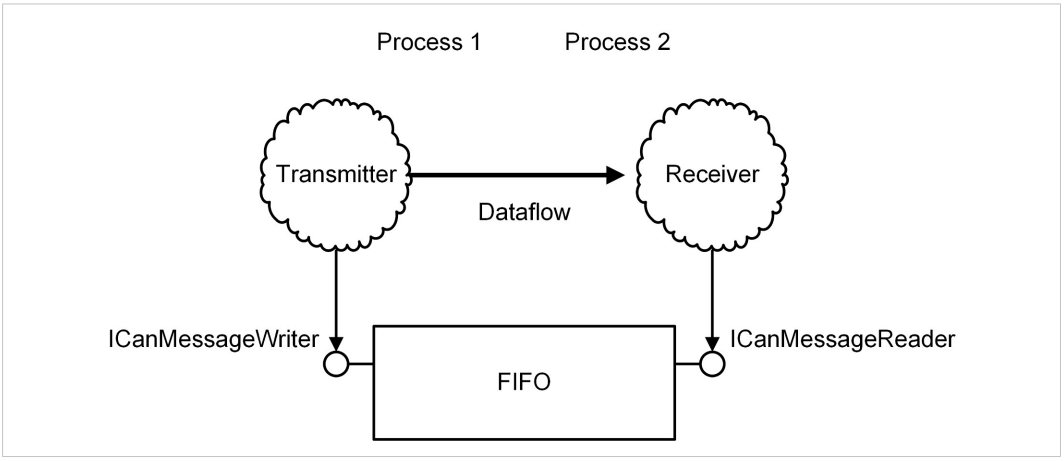


Fig. 5 FIFO for data exchange between two processes

FIFOs are also used to exchange data between components running in the kernel mode and programs running in the user mode.

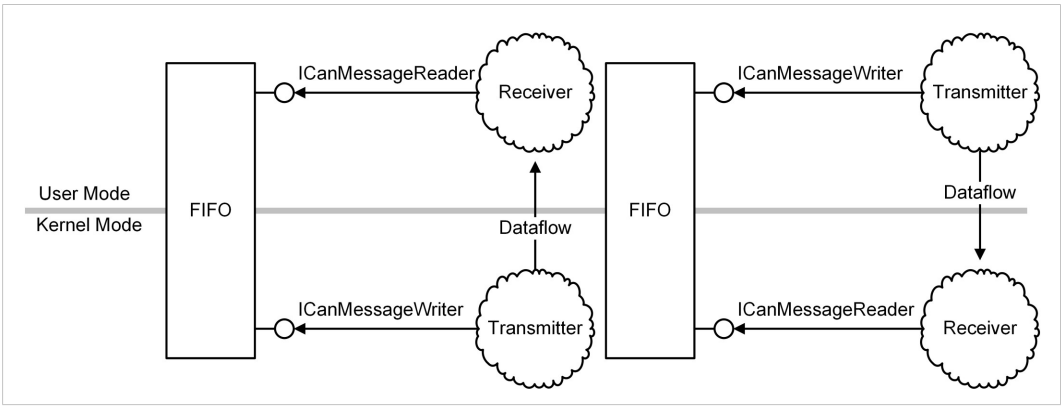


Fig. 6 Possible combination of a FIFO for data exchange between user and kernel mode

5.1.1 Functionality Receiving FIFO

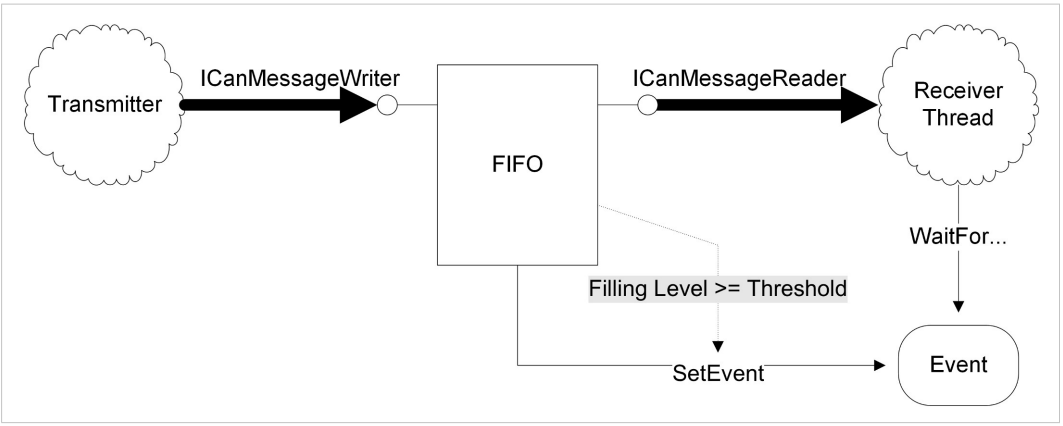


Fig. 7 Functionality Receiving FIFO

At the receiving side FIFOs are addressed via the interface `ICanMessageReader`.

Access files to read:

- To read individual messages, call method `GetMessage`.

  or

- To read several messages, call method `GetMessages`.
- To release on or more read and processed elements, call method `IDisposable.Dispose`.

Event Object

It is possible to assign an event object to the FIFO to prevent that the receiver has to ask if new data is available for reading. The event object is set to a signaled status if a certain filling level is reached.

- Create `AutoResetEvent` or `ManualResetEvent`.
  - ▷ Returned Handle is assigned to FIFO with method `AssignEvent`.
- Set threshold resp. filling level that triggers the event with property `Threshold`.

Afterwards the application is able to wait for the event and to read the received data with one of the methods `WaitOne` or `WaitAll`.

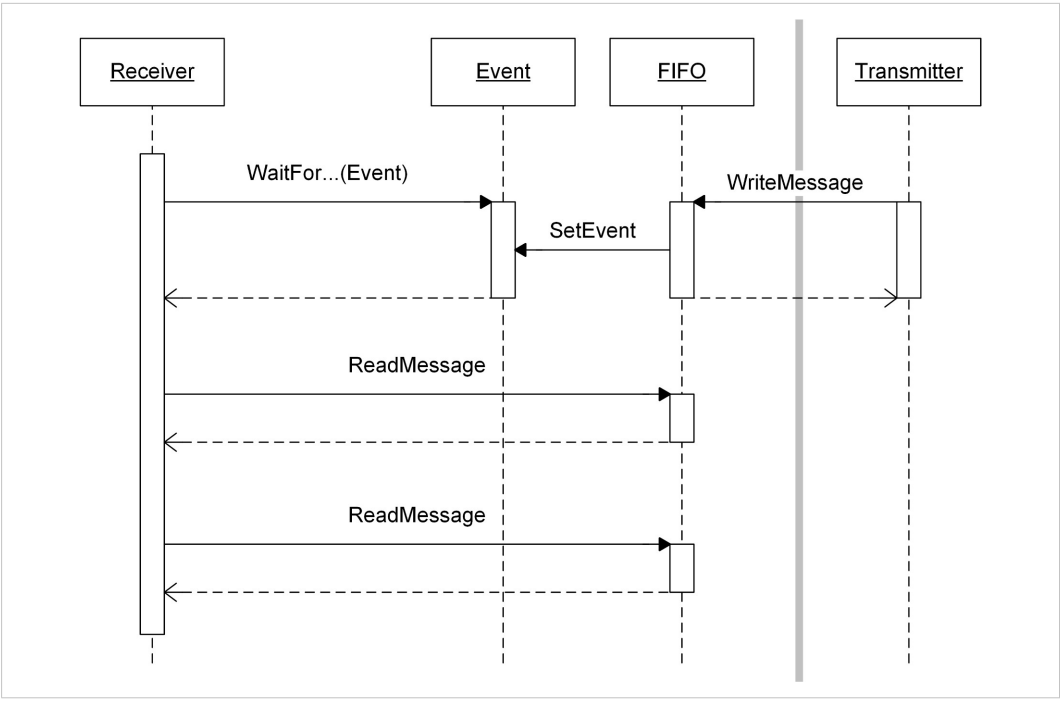


Fig. 8 Receiving sequence event-driven reading of data from the FIFO

Since the event is exclusively triggered with the exceedance of the set threshold, make sure that all entries of the FIFO are read in case of event-driven reading. If the threshold is set e. g. 1 and already 10 elements are in the FIFO when the event happens and only one is read, a following event will not be triggered until the next write-access. If no further write-access follows by the transmitter 9 unread elements are in the FIFO that aren't shown as event anymore.

5.1.2 Functionality Transmitting FIFO

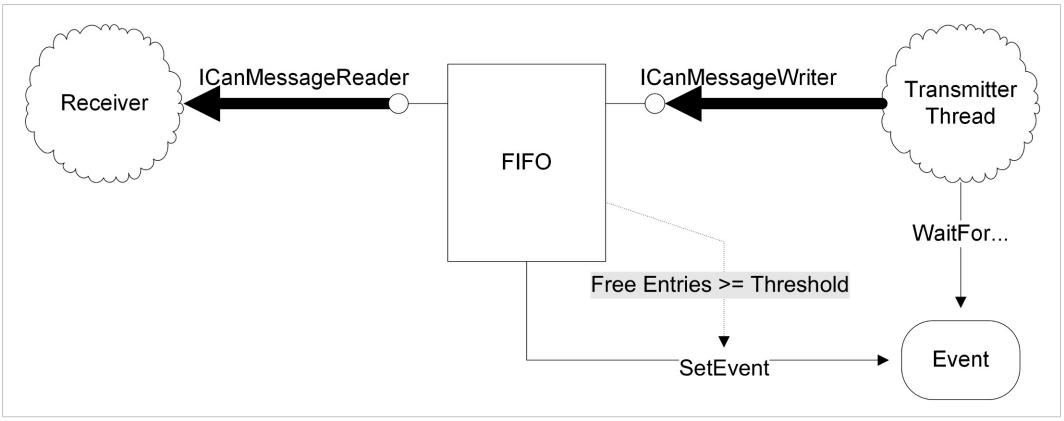


Fig. 9 Functionality transmitting FIFO

At the transmitting side FIFOs are addressed via the interface `ICanMessageWriter`.

Write data to be transmitted in the FIFO:

- To write individual message to the FIFO call method `WriteMessage`.

  or

- To write several messages to the FIFO, call method `WriteMessages`.

Event Object

It is possible to assign an event object to the FIFO to prevent that the receiver has to check if free elements are available. The event object is set to a signaled status if the number of free elements exceeds a certain value.

- Create `AutoResetEvent` or `ManualResetEvent`.
  - Returned Handle is assigned to FIFO with method `AssignEvent`.
- Set threshold resp. number of free elements that trigger the event with property `Threshold`.

Afterwards the application is able to wait for the event and to write new data in the FIFO with one of the methods `WaitOne` or `WaitAll`.

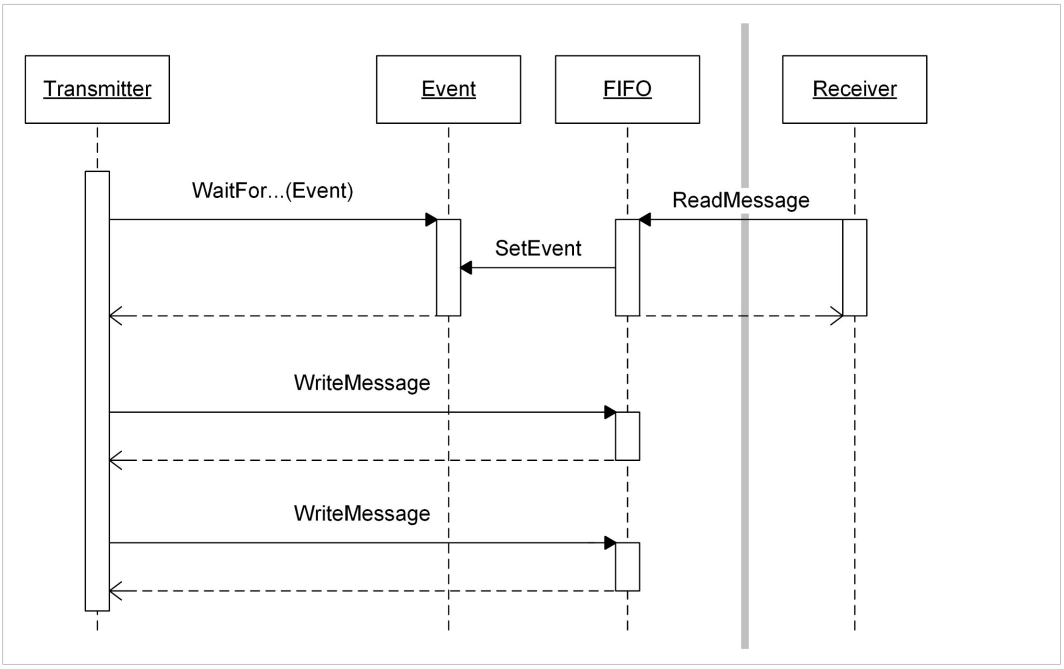


Fig. 10 Transmitting sequence event-driven writing of data to FIFO

6 Accessing the Bus Controller

Via the Bus Access Layer (BAL) the fieldbusses connected to the CAN interface are accessed.

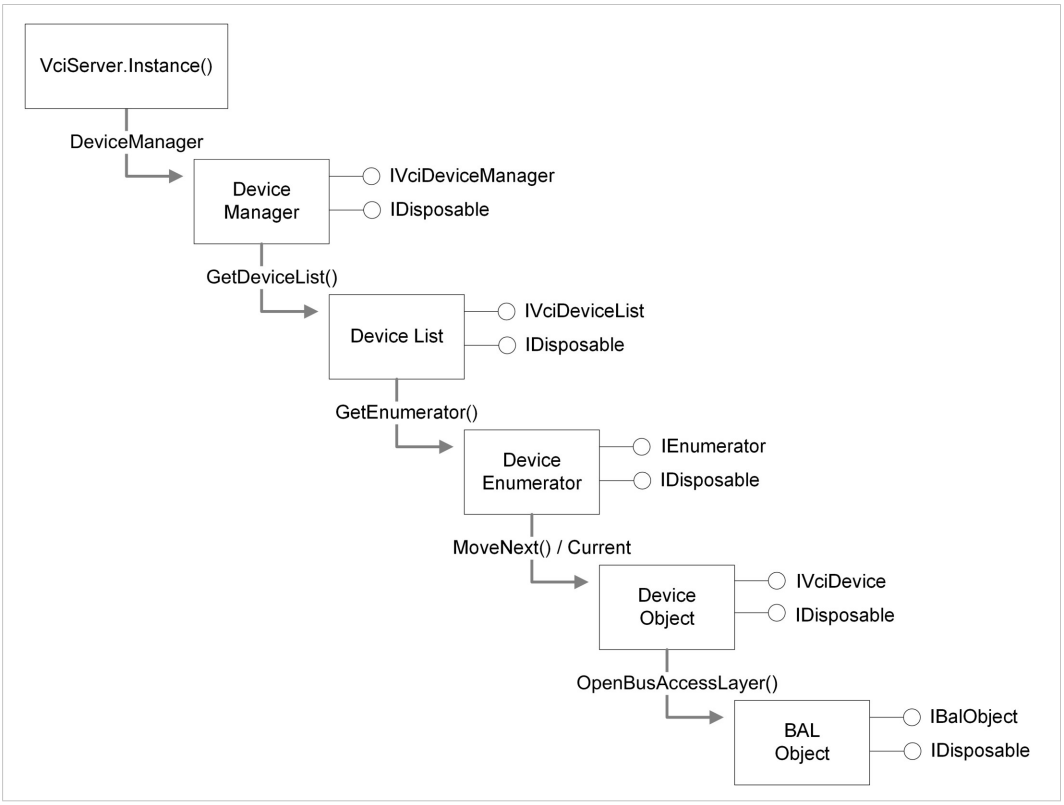


Fig. 11 Components for accessing the bus

- Search adapter in device list and open BAL with `IVciDeviceManager.OpenBusAccessLayer`.
- After opening release references to the device manager, device list, device enumerator or device object that are no longer needed with `IDisposable.Dispose`.

For further work with the adapter only the BAL object resp. its interface `IBalObject` is necessary. The BAL of an interface can be opened simultaneously by several programs.

The BAL object supports several types of bus connections.

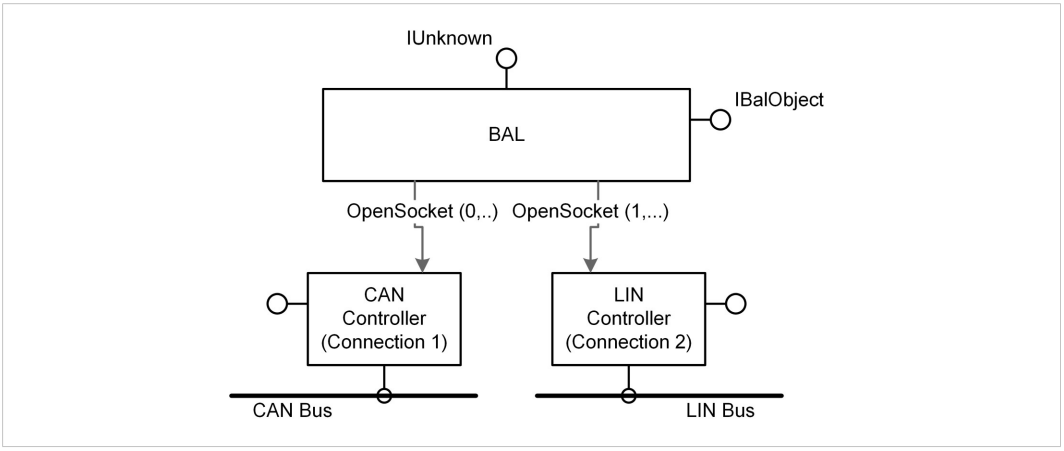


Fig. 12 BAL with CAN and LIN controller

Determine Number and Type of Provided Connections

- Call property `IBalObject.Resources`.
  - Returns information in form of a `BalResourceCollection`, that contains a BAL resource object for every provided bus connection.
  - BAL returns version number of device firmware via property `IBalObject.FirmwareVersion`.

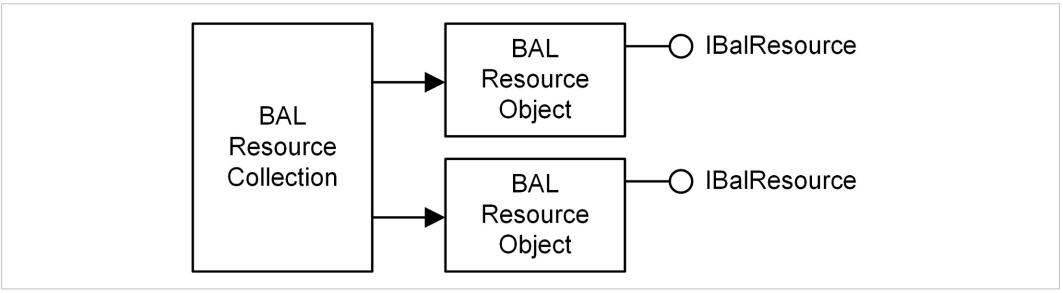


Fig. 13 BalResourceCollection with two bus connections

Accessing Connection or Interface of Connection

Access connections with method `IBalObject.OpenSocket`.

- In first parameter determine number of the connection to be opened. Value has to be in the range 0 to *IBalObject.Resources.Count*-1. To open connection 1 enter value 0, for connection 2 value 1 etc.
- In second parameter determine ID of the interface to access the connection.
- Call method.
  - Returns reference to requested interface.
  - Possibilities resp. interfaces of a connection are dependent on the supported bus.

Certain interfaces of a controller can only be accessed by one program, others can be accessed by any number of programs simultaneously. The rules of accessing the particular interfaces are dependent on the type of the connection and are described in detail in the following chapters.

6.1 CAN Controller

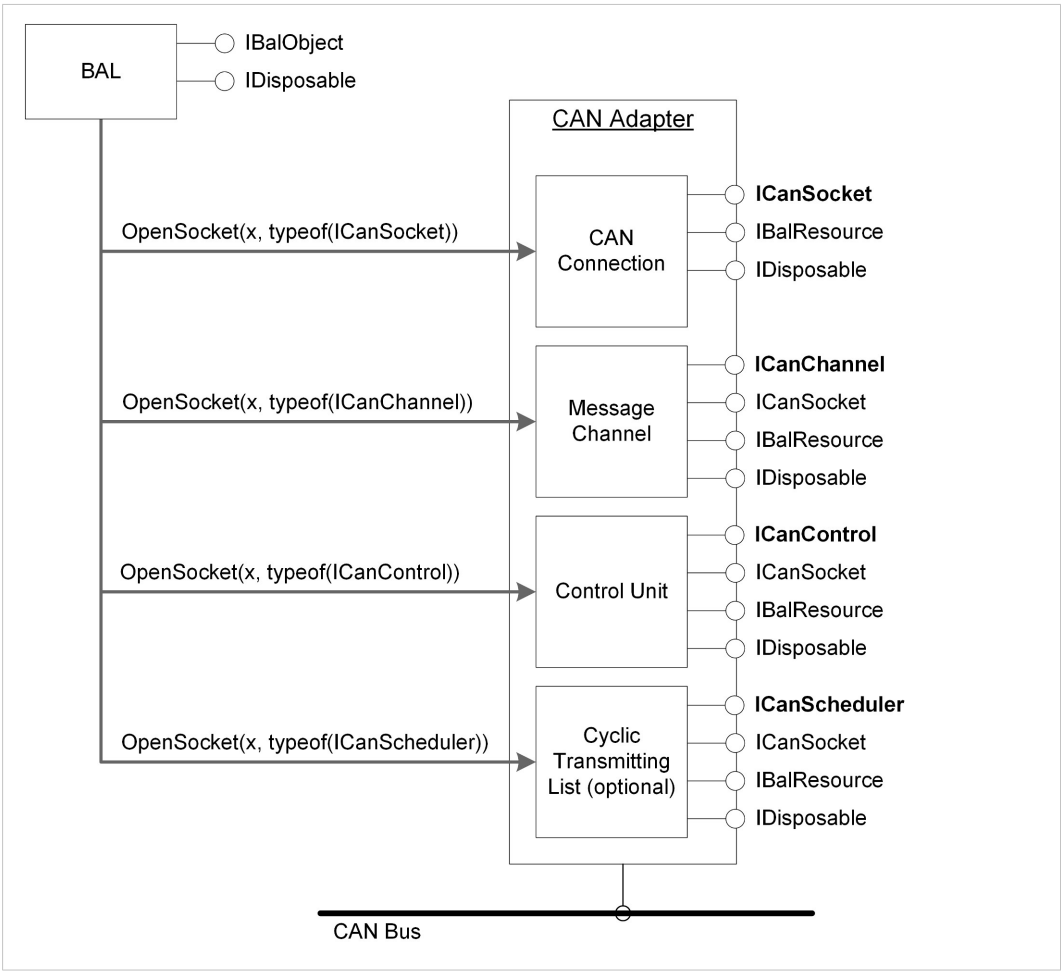


Fig. 14 Components CAN controller

Access to individual components of the CAN controller via the following interfaces:

- `ICanSocket`, `ICanSocket2` (CAN controller), see *Socket Interface, p. 21*
- `ICanControl`, `ICanControl2` (control unit), see *Control Unit, p. 28*
- `ICanChannel`, `ICanChannel2` (message channels), see *Message Channels, p. 21*
- `ICanScheduler`, `ICanScheduler2` (cyclic transmitting list), see *Cyclic Transmitting List, p. 35*, optional, exclusively with devices with their own microprocessor

The extended interfaces `ICanSocket2`, `ICanControl2`, `ICanChannel2` and `ICanScheduler2` allow the access to the new functions of CAN FD controllers. With standard controllers they can be used for further filter possibilities

6.1.1 Socket Interface

The socket interface `ICanSocket` resp. `ICanSocket2` is used to request features, possibilities and operating status of the CAN controller. The interface is not subjected to any access restrictions and can be opened by multiple applications simultaneously. Controlling via this interface is not possible.

Open with method `IBalObject.OpenSocket`.

- In parameter *socketType* enter type `ICanSocket` or `ICanSocket2`.
- Call method.

The properties of the CAN controller, like e. g. supported features are provided via properties.

- To determine current operating status of the controller, call property `LineStatus`.

6.1.2 Message Channels

Message channels consist of a receiving and an optional transmitting FIFO. One or more message channels are possible per CAN controller. CAN messages are exclusively received and transmitted via message channels.

Message channels with extended functionality (CAN FD) contain an additional, optional input filter.

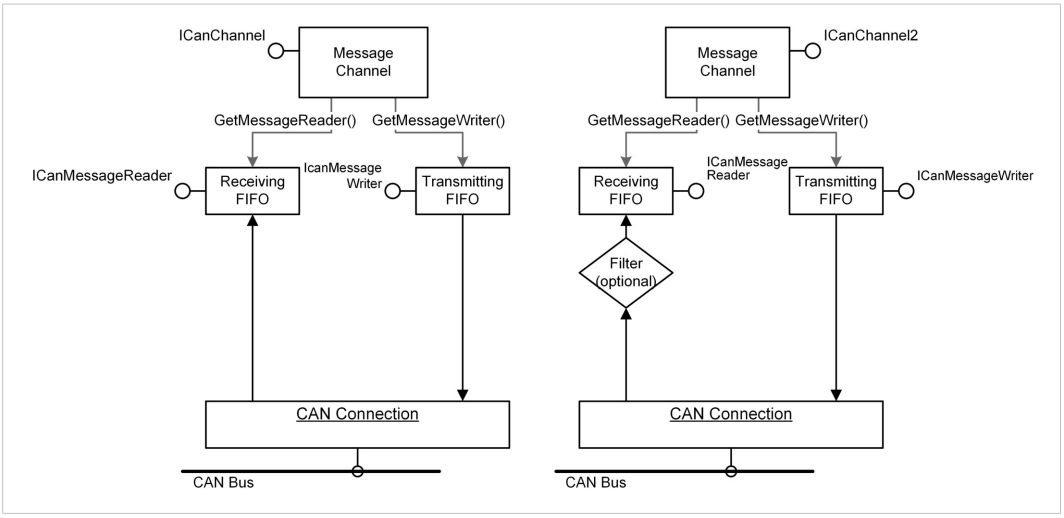


Fig. 15 Components and interfaces of a message channel

All CAN connections support message channels of the type `ICanChannel` and `ICanChannel2`. If the extended functionality of a message channel of type `ICanChannel2` is usable, is depending on the CAN controller of the connection. If the connection provides e. g. only a standard CAN controller, the extended functionality can not be used. With a message channel of type `ICanChannel` the extended functionality of a CAN FD can neither be used.

The basic functionality of a message channel is the same, irrespective whether the connection is used exclusively or not. In case of exclusive usage the message channel is directly connected to the CAN controller.

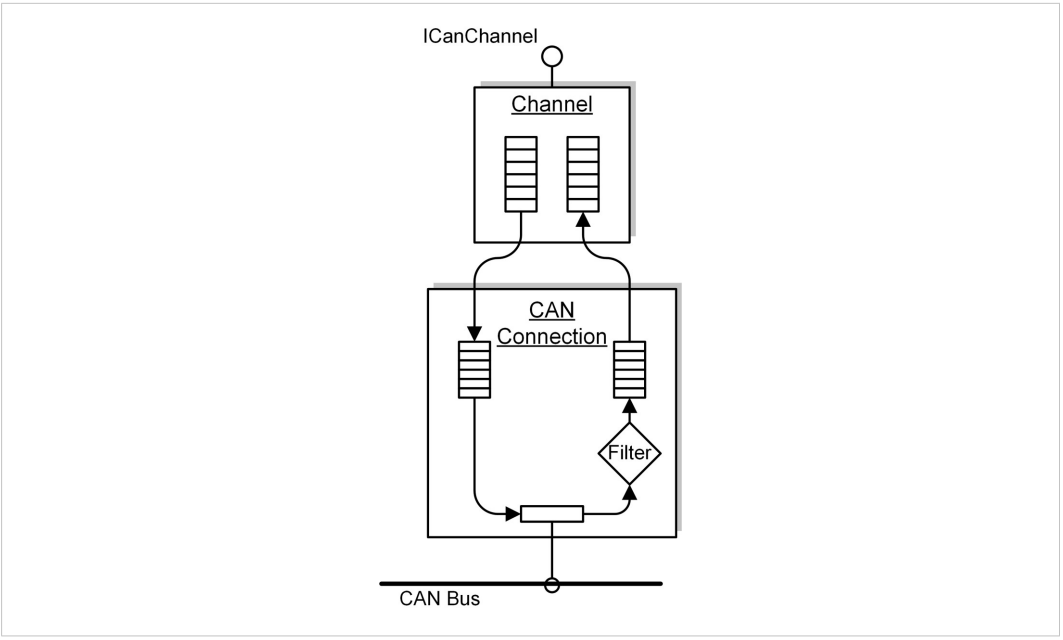


Fig. 16 Exclusive usage of a message channel

In case of non-exclusive usage the individual message channels are connected to the controller via a distributor.

The distributor transfers all received messages to all active channels and parallel the transmitted messages to the controller. No channel is prioritized i. e. the algorithm used by the distributor is designed to treat all channels as equal as possible.

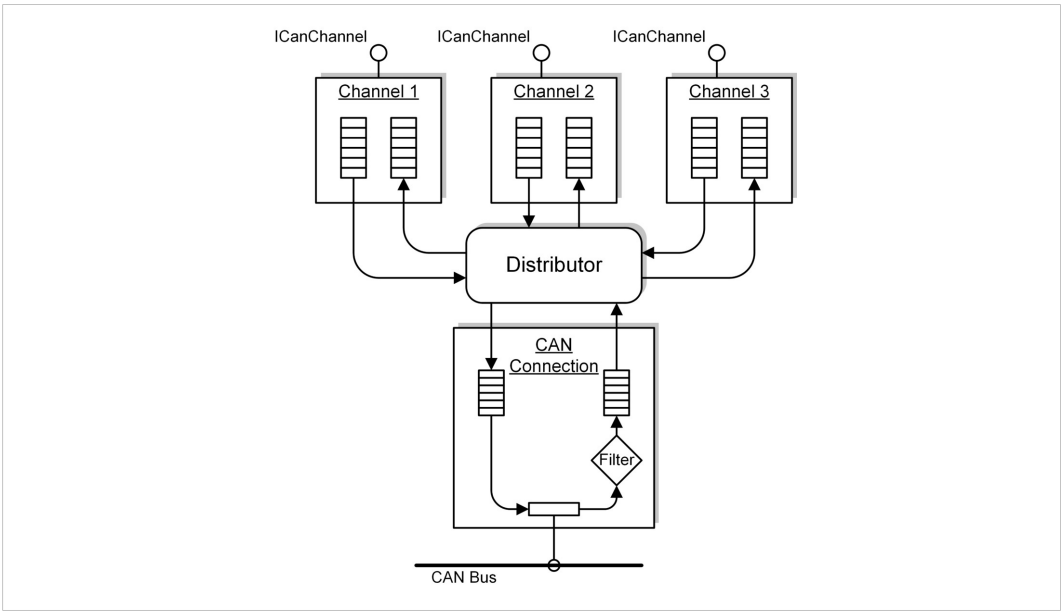


Fig. 17 CAN message distributor: possible configuration with three channels

Creating a Message Channel

Create with method `ICanSocket.OpenSocket` resp. for channels with extended functionality with `ICanSocket2.OpenSocket`.

- In Parameter *socketType* enter type `ICanChannel`.
- If controller is used exclusively (after successful execution no further message channels can be used) in parameter `exclusive` enter value `TRUE`.

  or

  If controller is used non-exclusively (further message channels can be opened and controller can be used by other applications) enter in parameter `exclusive` value `FALSE` .

The random access memory required for the FIFOs limits the possible number of channels.

Initializing the Message Channel

A newly generated message channel contains neither a receiving nor a transmitting FIFO. Before first use an initialization is necessary.

Initialize and create receiving and transmitting FIFOs with method `ICanChannel.Initialize` resp. with channels with extended functionality with `ICanChannel2.Initialize`.

- In parameters specify size of each FIFO in number of CAN messages.
- Call method.

In case of the usage of message channels with extended functionality an additional optional receiving filter can be created.

- In case of a 29 bit ID filter specify size of filter table in number of IDs in parameter *filterSize*.

  In case of 11 bit ID filter size of filter table is set to 2048 and can not be changed.
- If no receiving filter is needed set *filterSize* to 0.
- Specify functionality of 11 bit and 29 bit ID filter in parameter *filterMode*.
- Call method.

Initially specified functionality can be changed later for both filters separately with the method `SetFilterMode` at inactive message channels.

Activating the Message Channel

A new message channel is inactive. Messages can only be transmitted and received, if the message channel is active and the CAN controller is started.

- Activate message channel with method `ICanChannel.Activate`.
- Deactivate message channel with method `ICanChannel.Deactivate`.

Receiving CAN messages

The messages received on the bus and accepted by the filter are written to the receiving FIFO.

- Request interface `ICanMessageReader`, which is required to read with `ICanChannel.GetMessageReader` resp. with channels with extended functionality with `ICanChannel2.GetMessageReader`.

Reading messages from the FIFO:

- Call method `ReadMessage`.

  or

- To read several messages with one method call (optimized for high data throughput), create a field of CAN messages.
- Assign field to method `ReadMessages`.
  - ▷ `ReadMessages` tries to fill the field with received data.
  - ▷ Number of actually read messages is indicated with response value.

Possible usage of method `ReadMessage`:

```
void DoMessages( ICanMessageReader reader )
{
 ICanMessage message;
 while( reader.ReadMessage(out message) )
 {
 // Processing of message
 }
}
```

Possible usage of method `ReadMessages`:

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

 int readCount = reader.ReadMessages(out messages);
 for( int idx = 0; idx < readCount; idx++ )
 {
 // Processing of message
 }
```

Receiving Time of a Message

The receiving time of a message is available via the interface `ICanMessage` resp. `ICanMessage2` in property *TimeStamp*. The property contains the number of timer ticks that elapsed since the start of the controller resp. the hardware or since the last overrun of the counter.

Calculation of the length of a tick resp. the resolution of a time stamp in seconds: (t_{tsc}):

- t_{tsc} [s] = *TimeStampCounterDivisor* / *ClockFrequency*

  Fields *TimeStampCounterDivisor* and *ClockFrequency*, see properties `ICanSocket.ClockFrequency` and `ICanSocket.TimeStampCounterDivisor`

- channels with extended functionality:

  t_{tsc} [s] = *TimeStampCounterDivisor* / *ClockFrequency*

  Fields *TimeStampCounterDivisor* and *ClockFrequency*, see properties `ICanSocket2.ClockFrequency` and `ICanSocket2.TimeStampCounterDivisor`

Calculation of the relative receiving time (T_{rx}):

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

When the control unit is started a message of type `CanMsgFrameType.Info` is written in the receiving FIFOs of all active message channels. The time stamp of this message contains the relative starting point of the controller.

Transmitting CAN Messages

Messages are transmitted via the transmitting FIFO of the message channel.

- ▶ Request interface `ICanMessageWriter` that is required for transmitting with method `ICanChannel.GetMessageWriter` resp. with channels with extended functionality with `ICanChannel2.GetMessageWriter`.
- ▶ Transmit messages with method `SendMessage`.
- ▶ Assign the message of type `CanMessage` to be transmitted in parameter *message*.
- ▶ To transmit message delayed, enter value unequal 0 in parameter *TimeStamp* (further information see *Transmitting Messages Delayed, p. 26*).

  Exclusively messages of the type `CanMsgFrameType.Data` can be transmitted. Other message types are ignored by the controller and automatically rejected.

Possible usage of method `SendMessage`:

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

  // Initialize CAN message.
  message.TimeStamp            = 0; // No delayed transmitting
  message.Identifier           = id; // Message ID (CAN ID)
  message.FrameType            = CanMsgFrameType.Data;
  message.SelfReceptionRequest = false; // No Self Reception
  message.ExtendedFrameFormat  = false; // Standard frame
  message.DataLength           = 1; // only 1 data byte
  message[0]                   = data;

  // transmit message
  return writer.SendMessage(message);
}
```

Transmitting Messages Delayed

Controller with set bit `ICanSocket.SupportsDelayedTransmission` support the possibility to transmit messages delayed, with a latency between two consecutive messages.

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

▶ In field *CanMessage.TimeStamp* specify time in ticks that have to pass at a minimum before the next message is passed to the controller.

Delay Time

- Value 0 triggers no delay, that means a message is transmitted the next possible time.
- The maximal possible delay time is specified by field *ICanSocket.MaxDelayedTXTicks*.
- Resolution of a tick in seconds is calculated with the values of the fields `ICanSocket.ClockFrequency` and `ICanSocket.DelayedTXTimeDivisor` resp. `ICanSocket2.DelayedTXTimerClockFrequency` and `ICanSocket2.DelayedTXTimerDivisor`.

Calculation of the resolution of a tick in seconds

- Resolution [s] = DelayedTXTimeDivisor / ClockFrequency

The specified delay time represents a minimal value as it can not be guaranteed that the message is transmitted exactly after the specified time. Also, it has to be considered that if several message channels are used simultaneously on one connection the specified value is basically exceeded because the distributor handles all channels one after another.

Recommendation:

▶ If an application requires a precise time sequence, use connection exclusively.

Transmitting Messages Uniquely

Transmitting messages with set `SingleShotMode` flag the controller tries to transmit only once. If this transmitting attempt is not successful the message is rejected and there is no automatic transmitting repetition.

This happens e. g. if one or more bus participants are transmitting simultaneously. If the participant that is transmitting a message with set `SingleShotMode` flag bit loses the bus assignment (arbitration), the message is rejected and further transmitting is not attempted.

The functionality is exclusively available if the property `ICanSocket2.SupportsSingleShotMessages` returns TRUE.

Transmitting Messages with High Priority

Transmitting messages with set `HighPriorityMsg` flag are registered by the controller in a controller specific transmitting buffer that takes precedence over messages in the standard transmitting buffer and primarily transmits.

The functionality is exclusively available if the property `ICanSocket2.SupportsHighPriorityMessages` returns TRUE. If the bit is used observe that messages that are already in the transmitting FIFO can not be overtaken. The functionality is of minor impact resp. can only be sensibly used if the controller is opened exclusively and the transmitting FIFO is empty before addressing a message with set flag `HighPriorityMsg`.

6.1.3 Control Unit

The control unit provides the following functions via the interface `ICanControl`:

- configuration of CAN controller
- configuration of the transmitting features of the CAN controller
- configuration of CAN message filters
- requesting of current operating state

To make sure, to stop several applications e. g. from trying to start and stop the CAN controller simultaneously, the control unit can exclusively be opened by one application.

Opening the Interface

Open with method `IBalObject.OpenSocket`.

▶ Specify type `ICanControl` resp. with channels with extended functionality `ICanControl2` in parameter *socketType*.

▷ If the method returns *Exception*, the component is already used by another program.

▶ Close opened control unit with method `IDisposable.Dispose` and release access by other applications.

If other interfaces are opened during the closing of the control unit, the current settings remain.

Controller states

The control unit resp. the CAN controller is always in one of the following states:

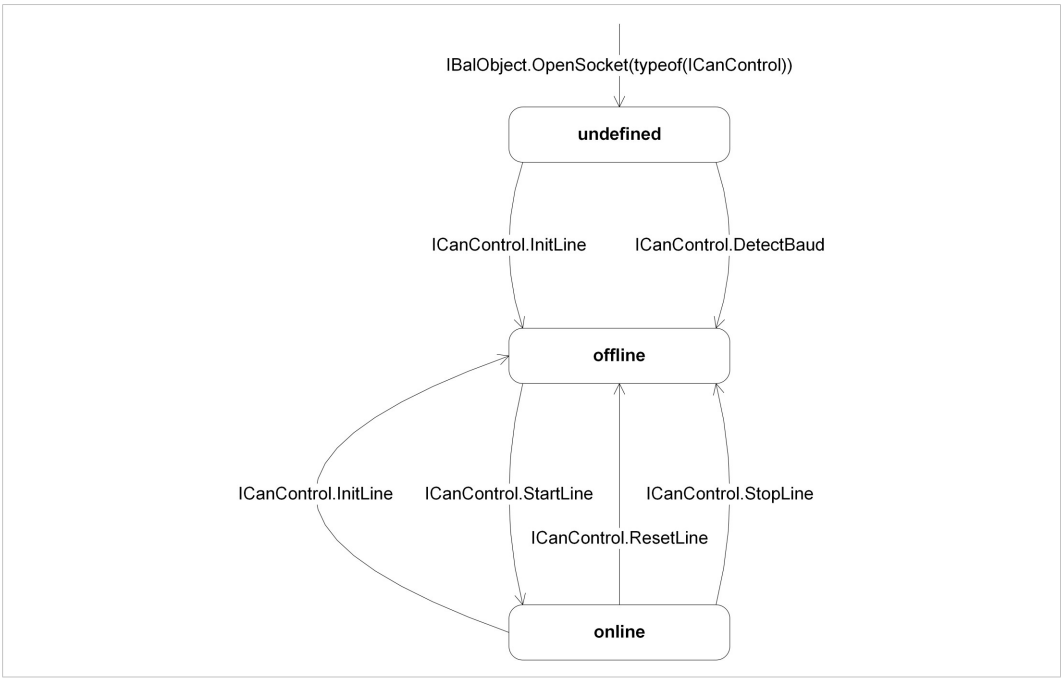


Fig. 18 Controller states

Initializing the controller

After the first opening of the control unit via the interface `ICanControl` or `ICanControl2` the controller is in an undefined state.

- ▶ To leave undefined state, call method `InitLine` or `DetectBaud`.
  - ▷ Controller is in state *offline*.
- ▶ Specify operating mode and bit rate of the controller with method `InitLine`.
- ▶ Specify operating mode in field *operatingMode*.
- ▶ Specify bitrate in `bitrate` (see *Specifying the Bit Rate, p. 30*).
- ▶ Call method.
  - ▷ Controller is initialized with specified values.

Starting CAN Controller

To start CAN controller and data transmission between controller and bus:

- ▶ Make sure that CAN controller is initialized (see *Initializing the controller, p. 29*).
- ▶ Call method `StartLine`.
  - ▷ Control unit is in state *online*.
  - ▷ Incoming messages are forwarded to all opened and active message channels.
  - ▷ Transmitting messages are transferred to the bus.

After successful start of the controller the control unit transmits an information message to all active message channels. The property *FrameType* of this message contains the value `CanMsgFrameType.Info`, the first data byte *Data[0]* the value `CanMsgInfoValue.Start` and the property *TimeStamp* the relative starting point (normally 0).

Stopping (resp. Resetting) the Controller

- ▶ Call method `StopLine`.
  - ▷ Controller is in state `offline`.
  - ▷ Data transfer between controller and bus is stopped.
  - ▷ Controller is deactivated.
  - ▷ Specified acceptance filter and filter list remain.
  - ▷ In case of an ongoing data transfer of the controller the function waits until the message is transmitted completely over the bus, before the message transmission is stopped. No faulty telegram is on the bus.

  or
- ▶ Call method `ResetLine`.
  - ▷ Controller is in state *offline*.
  - ▷ Controller hardware is reset.
  - ▷ Message filters are deleted.

> **!** After calling the method `ResetLine` a faulty message telegram on the bus is possible, if a not completely transferred message is in the transmitting buffer of the controller, because the transmitting is canceled also during an ongoing data transfer.

If `StopLine` or `ResetLine` are called the control unit transmits an information message to all active channels. The property *FrameType* of the message contains the value `CanMsgFrameType.Info`, the first data byte *Data[0]* the value `CanMsgInfoValue.Stop` resp. `CanMsgInfoValue.Reset` and the property *TimeStamp* the value 0. Neither `ResetLine` nor `StopLine` delete the content of the receiving and transmitting FIFOs of a message channel.

Specifying the Bit Rate

▶ Specify with fields *CanBitrate.Btr0* and *CanBitrate.Btr1*.

The values of the fields *CanBitrate.Btr0* and *CanBitrate.Btr1* correspond to the values of the bus timing register BTR0 and BTR1 of Philips SJA1000 CAN controller with a clock frequency of 16 MHz.

Bus timing values with CiA resp. CANopen conform bit rates:

Bit rate (KBit)	Predefined CiA bit rates	BTR0	BTR1
10	CanBitrate.Cia10KBit	0x31	0x1C
20	CanBitrate.Cia20KBit	0x18	0x1C
50	CanBitrate.Cia50KBit	0x09	0x1C
125	CanBitrate.Cia125KBit	0x03	0x1C
250	CanBitrate.Cia250KBit	0x01	0x1C
500	CanBitrate.Cia500KBit	0x00	0x1C
800	CanBitrate.Cia800KBit	0x00	0x16
1000	CanBitrate.Cia1000KBit	0x00	0x14
100	CanBitrate._100KBit	0x04	0x1C

Determine the Bit Rate Used in the Network

If the CAN connector is connected to a running network with unknown bit rate the current bit rate can be determined.

Method `DetectBaud` requires field with predefined bus timing values.

▶ Call method `DetectBaud`.

▶ Determined bus timing values can be assigned to `InitLine`.

Example for usage of method for automatic initialization of a CAN controller in a CANopen system:

```
void AutoInitLine( ICanControl control )
{
 // Determine bit rate
 int index = control.DetectBaud(10000, CanBitrate.CiaBitRates);

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

6.1.4 Message Filter

All control units and message channels with extended functionality have a two-level message filter. The data messages are exclusively filtered by the ID (CAN ID). Data bytes are not considered.

Transmitting messages with set *Self reception request* bit are entered to the receiving buffer as soon as they are transmitted over the bus. The message filter is bypassed.

Operating Modes

Message filters can be ran in different operation modes:

- Blocking mode (`CanFilterModes.Lock`):

  Filter blocks all data messages, independent of the ID. Used e. g. if an application is only interested in information, error or status messages.

- Passing mode (`CanFilterModes.Pass`):

  Filter is completely opened and all data messages can pass. Default operation mode in case of usage of interface `ICanChannel`.

- Inclusive filtering (`CanFilterModes.Inclusive`):

  All data messages with either in the acceptance filter released ID or registered in the filter list ID can pass the filter (e. i. all registered IDs). Default operation mode in case of usage of interface `ICanControl`.

- Exclusive filtering (`CanFilterModes.Exclusive`):

  All data messages with either in the acceptance filter released ID or registered in the filter list ID are blocked by the filter (e. i. all registered IDs).

In case that the interface `ICanControl` is used, the operating mode of the filter cannot be changed and is preset to `CanFilterModes.Inclusive`. In case that the interface `ICanControl2` resp. `ICanChannel2` is used, the operation mode can be set to one of the above stated modes with the method `SetFilterMode`.

To ask for the operating mode of the filter call method `GetFilterMode`.

Inclusive and Exclusive Operating Mode

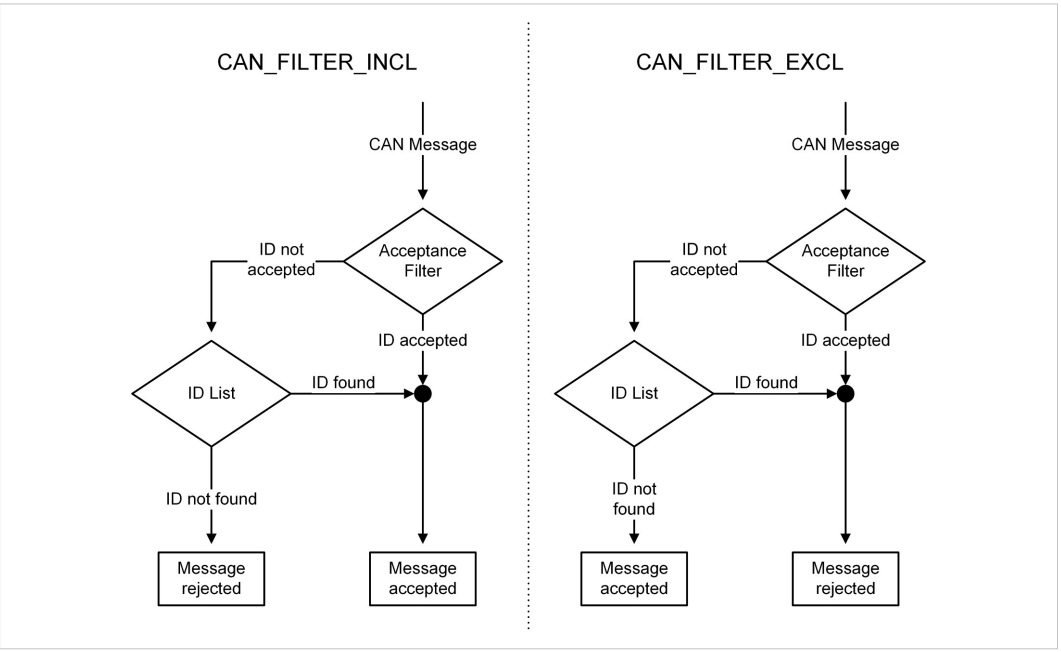


Fig. 19 Filtering mechanism inclusive and exclusive operating mode

The first filter level consists of an acceptance filter that compares the ID of a received message with a binary bit model. If the ID correlates with the set bit model the ID is accepted. In case of inclusive operating mode the message is accepted. In case of exclusive operating mode the message is immediately rejected.

If the first filter level does not accept the ID it is forwarded to the second filter level. The second filter level consists of a list with registered message IDs. If the ID of the received message is equal to an ID in the list, the message is accepted in case of inclusive filtering and rejected in case of exclusive filtering.

Filter Chain

Each message channel is connected to a controller either directly or indirectly via a distributor (see *Message Channels, p. 21*). If a filter is used both with the controller and with the message channel a multi-level filter chain is formed. Messages that are filtered out by the controller are invisible for the down-streamed channels.

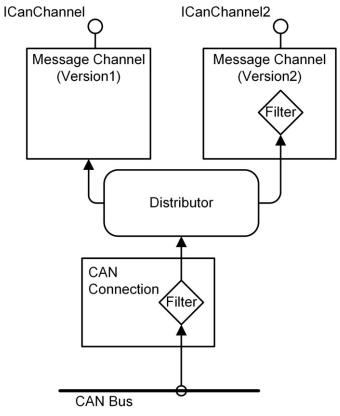


Fig. 20 Filter chain

Setting the Filter

Control units and message channels have separated and independent filters for 11 bit and 29 bit IDs. Messages with 11 bit ID are filtered by the 11 bit filter and messages with 29 bit ID by the 29 bit ID filter.

To distinguish between 11 and 29 bit ID filter all stated methods have the parameter *bSelect*.

Changes of the filters during operation are not possible.

If the controller is reset or initialized, all filters are specified to let all messages pass.

- ▶ Make sure that control unit is *offline* resp. that the message channel is inactive.

If the interfaces `ICanControl2` resp. `ICanChannel2` are used, the operating mode of the filter is preset during the initialization of the component. The specified value serves simultaneously as default value for the method `ICanControl2.ResetLine`.

- ▶ Make sure, that controller is in state *offline*.
- ▶ To set the filter after initialization, call method `SetFilterMode`.
- ▶ Specify filter with methods `SetAccFilter`, `AddFilterIds` and `RemFilterIds`.
- ▶ In parameter *bSelect* select 11 or 29 bit filter.

The bit samples in parameters *code* and *mask* determine which IDs can pass the filter.

- ▶ In parameter *code* and *mask* specify two bit samples.
  - ▷ Value of *code* specifies the bit model of the ID.
  - ▷ *mask* specifies which bit is used for the comparison.

If a bit in *mask* has the value 0, the correlating bit in *code* is not used for the comparison. But if it has the value 1, it is relevant for the comparison.

In case of the 11 bit filter exclusively the lower 12 bits are used. In case of the 29 bit filter the bits 0 to 29 are used. All other bits of the 32 bit value must be set to 0 before one of the methods is called.

Correlation between the bits in the parameters *code* and *mask* and the bits in the message 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

The bits 1 to 11 resp. 1 to 29 correspond to the bits 0 to 10 resp. 0 to 28. Bit 0 of every value defines the value of the remote transmission request bit (RTR) of a message.

The following example shows the values that must be used for *code* and *mask* to register message IDs in the range of 100 h to 103 h (of which also the RTR bit must be 0) in the filter:

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

The example shows that with a simple acceptance filter only individual IDs or groups of IDs can be released. If the desired identifier do not correspond with a certain bit model a second filter level, a list with IDs, must be used. The amount of IDs a list can receive can be configured. Every list has space for up to 2048 IDs resp. 4096 entries.

- Register individual or groups of IDs with method `AddFilterIds`.
- If necessary remove from list with method `RemFilterIds`.

The parameters *code* and *mask* have the same format as showed above.

If method `AddFilterIds` is called with same values as in the above example, the method enters the identifier 100 h to 103 h to the list.

- To register exclusively an individual ID in the list, specify the desired ID (including RTR bit) in *code* and in *mask* the value `FFFh` (11 bit ID) resp. `3FFFFFFFh` (29 bit ID).
- To disable acceptance filter completely, when calling method `SetAccFilter` enter in *code* the value `CanAccCode.None` and in *mask* the value `CanAccMask.None`.
  - ▷ Filtering is exclusively done with ID list.

  or
- Configure acceptance filter with the values `CanAccCode.All` and `CanAccMask.All`.
  - ▷ Acceptance filter accepts all IDs and ID list is ineffective.

6.1.5 Cyclic Transmitting List

With the optionally provided transmitting list of the controller up to 16 messages can be transmitted cyclically. The access to this list is limited to one application and therefore can not be used by several programs simultaneously. It is possible to increment a certain part of a CAN message after each transmitting process.

Open interface with method `IBalObject.OpenSocket`.

- In parameter *socketType* specify type `ICanScheduler`.
  - If method returns an error code respective `VciException`, the transmitting list is already under control of another program and can not be opened again.
  - If method returns an error code respective `NotImplementedException`, the CAN controller does not support a cyclic transmitting list.
- If another transmitting list is opened, close opened transmitting list with method `IDisposable.Dispose`.
- Add message objects with `ICanScheduler.AddMessage` resp. in case of controller with extended functionality with `ICanScheduler2.AddMessage` to the list.
  - If run successfully the method returns a new cyclic transmitting object with the interface `ICanCyclicTXMsg`.

One controller exclusively supports one transmitting list. The methods of the interfaces `ICanScheduler` or `ICanScheduler2` therefore refer to the same list. As the interfaces are exclusively different regarding the data type of the transmitted messages, whereas the functionality is identical, only the functionality of the interface `ICanScheduler` is described hereafter.

- Specify cycle time of a message in number of ticks in field *CanCyclicTXMsg.CycleTicks*.
- Make sure that specified value is higher than 0 but less than or equal the value in field *ICanSocket.MaxCyclicMsgTicks*.
- Calculate length of a tick resp. the cycle time (t_z) of the transmitting list with values in fields *ICanSocket.ClockFrequency* and *ICanSocket.CyclicMessageTimeDivisor* with the following formula:

  t_z [s] = (*CyclicMessageTimeDivisor* / *ClockFrequency*)

The transmitting task of the cyclic transmitting list divides the available time in individual segments resp. time frames. The length of a time frame is exactly the same as the length of a tick resp. the cycle time.

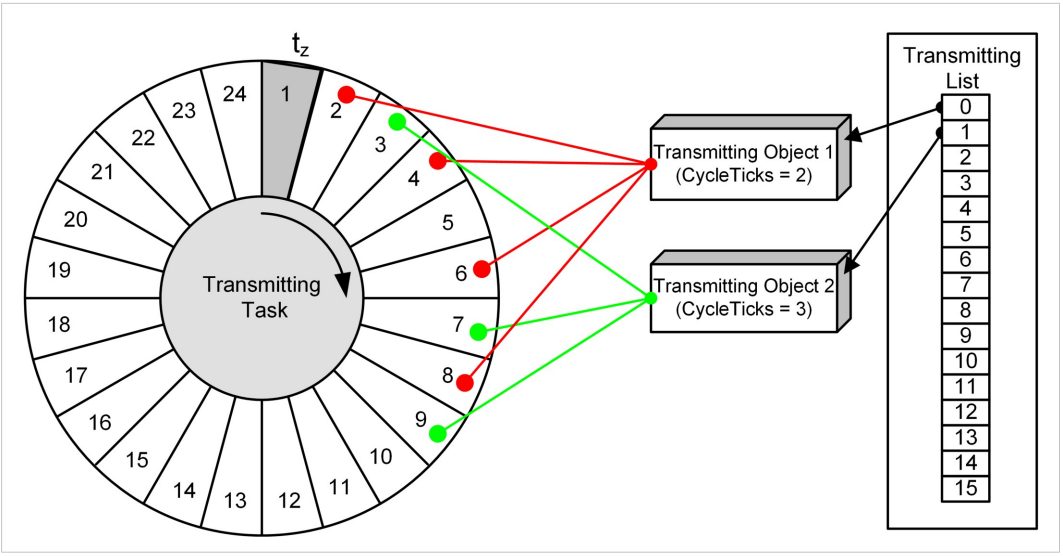


Fig. 21 Transmitting task of the cyclic transmitting list with 24 time frames

The number of time frames supported by the transmitting task is equal to the value in field *ICanSocket.MaxCyclicMsgTicks*.

The transmitting task can transmit exclusively one tick per message, e. i. exclusively one transmitting object can be matched to a time frame. If the transmitting object is created with a cycle time of 1 all time frames are occupied and no other objects can be created. The more transmitting objects are created, the larger their cycle time must be selected. The rule is: The total of all 1/*CycleTime* must be less than 1.

In the example a message shall be transmitted every 2 ticks and a further message every 3 ticks, this amounts 1/2 + 1/3 = 5/6 = 0,833 and therefore a valid value.

When creating the transmitting object 1 the time frames 2, 4, 6, 8, etc. are occupied. If the second transmitting object is created with a cycle time of 3, it leads to a collision in the time frames 6, 12, 18, etc. because these time frames are already occupied by the transmitting object 1.

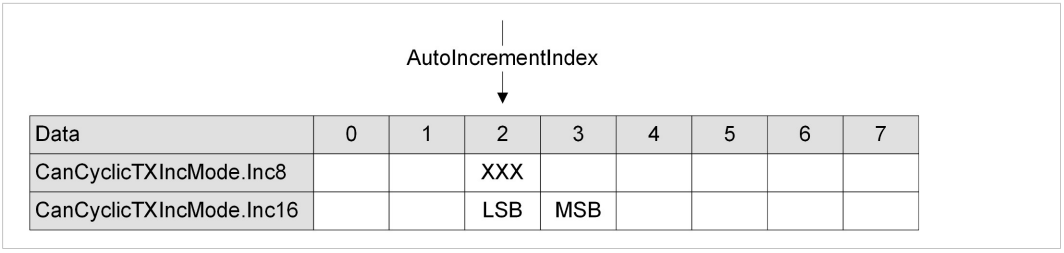
Collisions are resolved in shifting the new transmitting object in the respectively next free time frame. The transmitting object 2 of the example above then occupies the time frames 3, 7, 9, 13, 19, etc. The cycle time of the second object therefore is not met exactly and in this case leads to an inaccuracy of +1 tick.

The temporal accuracy of the transmitting of the objects is heavily depending on the message load on the bus. With increasing load the transmitting time gets more and more imprecise. The general rule is that the accuracy decreases with increasing bus load, smaller cycle times and increasing number of transmitting objects.

The field *CanCyclicTXMsg.AutoIncrementMode* specifies if certain parts of the message are automatically incremented after transmitting or if they remain unmodified.

If the value `CanCyclicTXIncMode.Nolnc` is specified, the content of the message remains unmodified. With the value `CanCyclicTXIncMode.Incld` the field *Identifier* of the message is automatically incremented by 1 after every transmission. If the field *Identifier* reaches the value 2048 (11 bit ID) resp. 536.870.912 (29 bit ID) an overflow to 0 automatically takes place.

With the values `CanCyclicTXIncMode.Inc8` resp. `CanCyclicTXIncMode.Inc16` in field `CanCyclicTXMsg.AutoIncrementMode` an individual 8 bit resp. 16 bit value is incremented in the data field of the message after every transmission. The field *AutoIncrementIndex* specifies the index of the data field.



Data	0	1	2	3	4	5	6	7
CanCyclicTXIncMode.Inc8			XXX					
CanCyclicTXIncMode.Inc16			LSB	MSB				

Fig. 22 Auto increment of data fields

Regarding 16 bit values, the low byte (LSB) is located in field *Data[AutoIncrementIndex]* and the high byte (MSB) in field *Data[AutoIncrementIndex +1]*. If the value 255 (8 bit) resp. 65535 (16 bit) is reached, an overflow to 0 takes place.

- If necessary remove transmitting object from list with method `RemMessage` . The method expects the list index of the object to remove returned by `AddMessage` .
- To transmit newly created transmitting object, call method `StartMessage` .
- If necessary stop transmitting with method `StopMessage` .

The current status of an individual transmitting object is returned by the property `Status`. The transmitting object statuses are updated via method `UpdateStatus`.

The transmitting task is deactivated after opening of the transmitting list. The transmitting task does not transmit any message in deactivated state, even if the list is created and contains started transmitting objects.

- To start all transmitting objects simultaneously, start all transmitting objects with method `StartMessage` .
- To activate or deactivate transmitting task, call method `Resume`.
- To stop all transmitting objects simultaneously, call method `Suspend`.
- To reset a transmitting task call method `Reset`.
  - Transmitting task is stopped.
  - All unregistered transmitting objects are removed from the specified cyclic transmitting list.

6.2 LIN-Controller

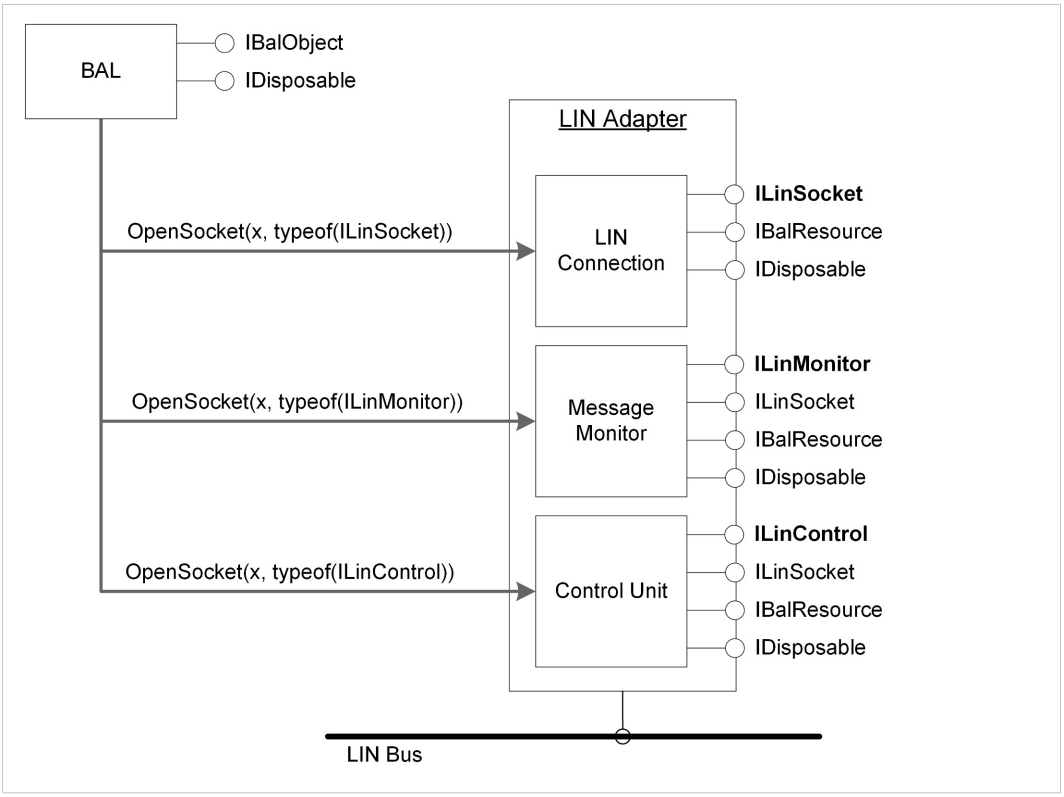


Fig. 23 Components LIN controller

Access to individual sub-components via interfaces `ILinSocket`, `ILinMonitor` and `ILinControl`.

`ILinSocket` (see *Socket Interface, p. 39*) provides the following functions:

- requesting of LIN controller functionality
- requesting of current controller state

`ILinMonitor` (see *Message Monitors, p. 39*):

- represents message monitor
- one or more message monitors possible per LIN connection
- LIN messages are exclusively received via message monitors.

`ILinControl` (see *Control Unit, p. 42*) provides the following functions:

- configuration of LIN controller
- configuration of transmitting features
- requesting of current controller state

6.2.1 Socket Interface

The interface `ILinSocket` is not subjected to any access restrictions and can be opened by multiple applications simultaneously. Controlling the connection via this interface is not possible.

Open with method `IBalObject.OpenSocket`.

- In parameter *socketType* specify type `ILinSocket`.

The properties of the LIN controller, like e. g. supported functions are provided via properties.

- To determine current operating mode and status of the controller, call property `LineStatus`.

6.2.2 Message Monitors

A LIN message monitor consists of a receiving FIFO.

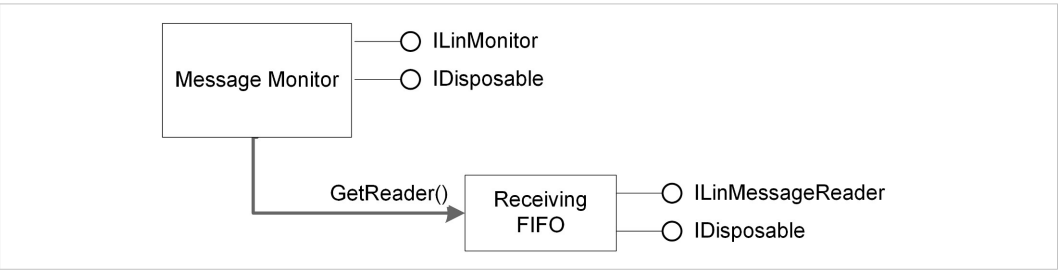


Fig. 24 Components LIN message monitor

The functionality of a message monitor is the same, irrespective whether the connection is used exclusively or not.

In case of exclusive usage the message monitor is directly connected to the LIN controller.

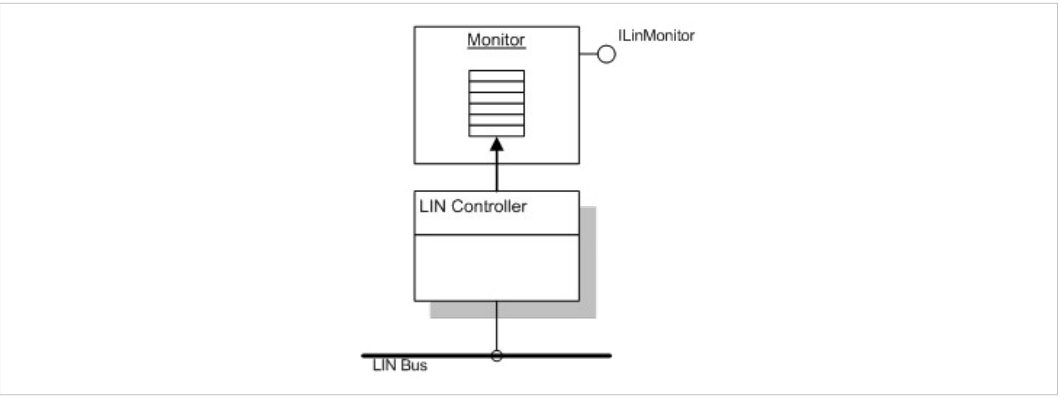


Fig. 25 Exclusive usage

In case of non-exclusive usage the individual message monitors are connected to the LIN controller via a distributor. The distributor transfers all on the LIN controller received messages to all active channels. No channel is prioritized i. e. the algorithm used by the distributor is designed to treat all monitors as equal as possible.

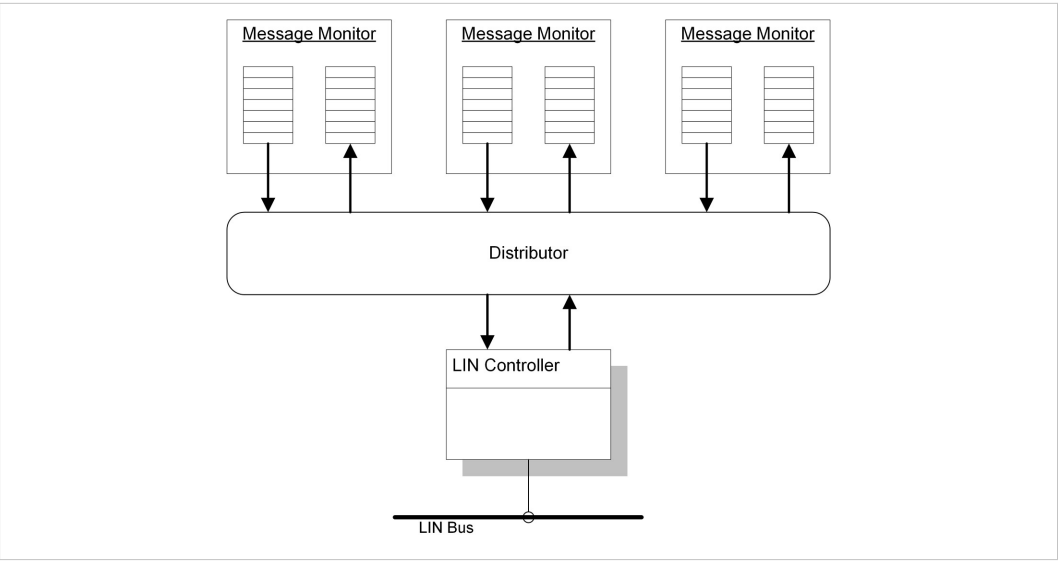


Fig. 26 Non-exclusive usage (with distributor)

Creating a Message Monitor

Create a message monitor with method `IBalObject.OpenSocket`.

- In Parameter *socketType* specify type `ILinMonitor`.
- To use controller exclusively (after successful execution no further message monitors can be used) specify in parameter *exclusive* value `TRUE` .

  or

  To use controller non-exclusively (creation of any number of monitors is possible) specify in parameter *exclusive* value `FALSE` .

Initializing the Message Monitor

A newly generated message monitor contains no receiving FIFO.

- Initialize message monitor and create receiving FIFO with method `ILinMonitor.Initialize`.
- In parameters specify size of receiving FIFO in number of LIN messages.

Activating Message Monitor

A newly generated monitor is deactivated. Messages are exclusively received by the bus if the message monitor is active and if the LIN controller is started. Further information about LIN controllers see chapter *Control Unit, p. 42*.

- Activate message monitor with method `ILinMonitor.Activate`.
- Disconnect active channel with method `ILinMonitor.Deactivate`.

Receiving LIN Messages

▶ Request interface `ILinMessagReader` which is necessary for reading with method `ILinMonitor.GetMessageReader`.

Reading messages from the FIFO:

▶ Call method `ReadMessage`.

or

▶ To read several messages with one method call (optimized for high data throughput), create a field of LIN messages.

▶ Transmit field to method `ReadMessages`.

▷ `ReadMessages` tries to fill the field with received data.

▷ Number of actually read messages is indicated with response value.

Possible usage of method `ReadMessage`:

```
void DoMessages( ILinMessageReader reader )
{
 ILinMessage message;
 while( reader.ReadMessage(out message) )
 {
 // Processing of message
 }
}
```

Possible usage of method `ReadMessages`:

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

 int readCount = reader.ReadMessages(out messages);
 for( int idx = 0; idx < readCount; idx++ )
 {
 // Processing of message
 }
```

6.2.3 Control Unit

The control unit can exclusively be opened by one application. Simultaneous multiple opening of the Interface by several programs is not possible.

Opening the Interface

Open with method `IBalObject.OpenSocket`.

- In parameter *socketType* specify type `ILinControl`.
  - If the method returns *Exception*, the component is already used by another program.
- Close opened control unit with method `IDisposable.Dispose` and release access by other applications.

If other interfaces are opened during the closing of the control unit, the current settings remain.

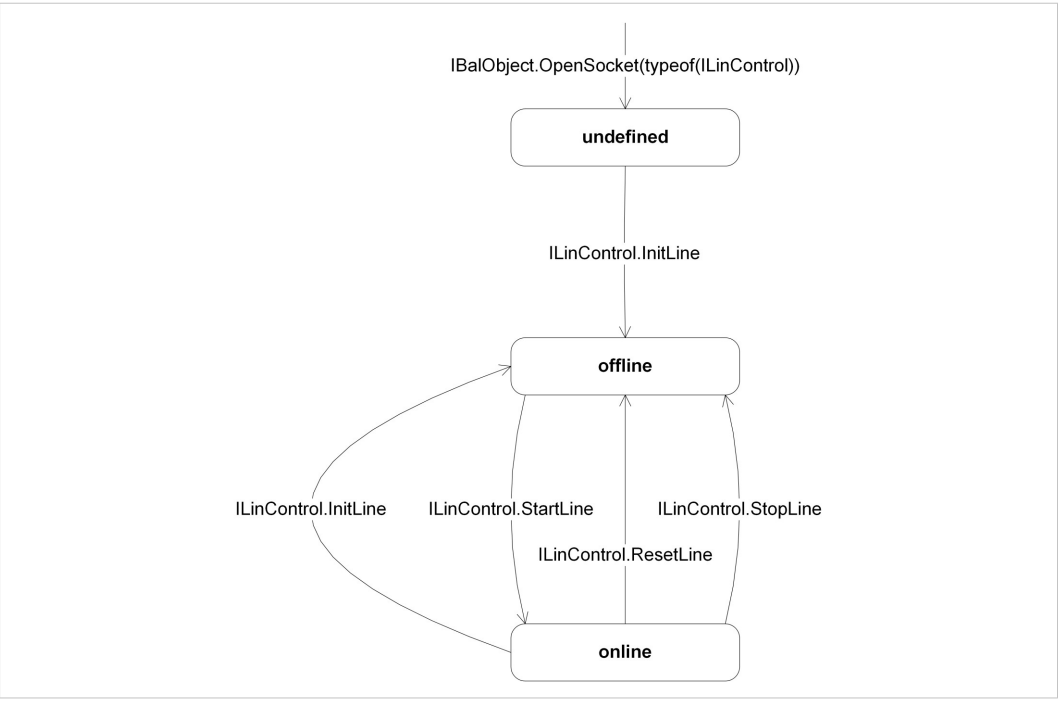


Fig. 27 LIN controller states

Initializing the controller

After the first opening of the interface `ILinControl` the controller is in undefined state.

- To leave undefined state, call method `InitLine`.
  - ▷ Controller is in state *offline*.
- Specify system mode and transmission rate with method `InitLine` .
  - ▷ Method expects structure `LinInitLine` with values for operating mode and bit rate.
- Specify transmission rate in bits per second in field *LinInitLine.Bitrate*.

  Valid values are between 1000 and 20000 bit/s, resp. between `LinBitrate.MinBitrate` and `LinBitrate.MaxBitrate`.

If the controller supports automatic bit identification, automatic bit identification can be activated with `LinBitrate.AutoRate`.

Recommended bit rates:

Slow	Medium	Fast
LinBitrate. Lin2400Bit	LinBitrate. Lin9600Bit	LinBitrate. Lin19200Bit

Starting and Stopping the Controller

- To start LIN controller, call method `StartLine` .
  - ▷ LIN controller is in state *online*.
  - ▷ LIN controller is actively connected to bus.
  - ▷ Incoming messages are forwarded to all opened and active message monitors.
- To stop LIN controller call method `StopLine` .
  - ▷ LIN controller is in state *offline*.
  - ▷ Message transfer to the monitor is interrupted and controller is deactivated.
  - ▷ In case of an ongoing data transfer of the controller the method waits until the message is transmitted completely over the bus, before the message transmission is stopped.
- Call method `ResetLine` to shift controller in state *offline* and to reset controller hardware.

With calling the method `ResetLine` a faulty message telegram on the bus is possible if an ongoing transmission is interrupted.

Neither `ResetLine` nor `StopLine` delete the content of the receiving FIFOs of a message monitor.

Transmitting LIN Messages

Messages can be transmitted directly via the method `ILinControl.WriteMessage` or can be registered in a response table in the controller.

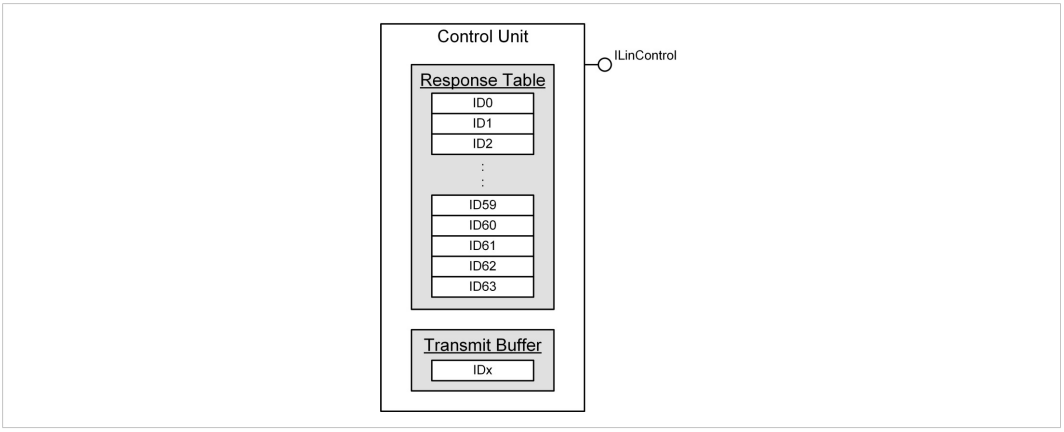


Fig. 28 Internal structure of a control unit

The control unit contains an internal response table with the response data for the IDs transmitted by the master. If the controller detects an ID that is assigned to it and transmitted by the master it transmits the response data entered in the table at the corresponding position automatically to the bus.

To chance or update the content of the response table call method `ILinControl.WriteMessage`.

- In parameter *send* set value `FALSE`.
  - Message with response data in the data field of structure `LinMessage` is assigned to method in parameter *message*.
- To clear response table call method `ILinControl.ResetLine`.

Data field of structure `LinMessage` contains the response data. The LIN message must be of type `LinMessageType.Data` and must contain an ID in the range 0 to 63.

Irrespective of the operating mode (master or slave) the table must be initialized before the controller is started. It can be updated at any time without stopping the controller.

Transmitting messages directly to the bus with method `ILinControl.WriteMessage`.

- Set parameter *send* to value `TRUE`.
  - Message is registered in the transmitting buffer of the controller, instead of the response table.
  - Controller transmits message to bus as soon as it is free.

If the controller is configured as master, control messages `LinMessageType.Sleep`, `LinMessageType.Wakeup` and `LinMessageType.Data` can be directly transmitted. If the controller is configured as slave exclusively `LinMessageType.Wakeup` messages can be directly transmitted. With all other message types the method returns an error code.

A message of type `LinMessageType.Sleep` generates a goto-Sleep frame, a message of type `LinMessageType.Wakeup` a wake-up frame on the bus. For further information see chapter Network Management in LIN specifications

In the master mode the method `ILinControl.WriteMessage` also serves for transmitting IDs. For this a message of type `LinMessageType.Data` with valid ID and data length, where the flag *IdOnly* is set to `TRUE` is transmitted.

Irrespective of the value of the parameter *send* `ILinControl.WriteMessage` always returns immediately to the calling program without waiting for the transmission to be completed. If the method is called before the last transmission is completed or before the transmission buffer is free again, the method returns with a respective error code.

7 Interface Description

For a detailed description of the VCI .NET interfaces and classes see installed folder reference *vci4net.chm* in sub-directory *manual*.

This page intentionally left blank