VCI - Virtual CAN Interface

.Net-API Programmers Manual

Software Version 3

IXXAT
The expert for industrial and automotive communication

IXXAT

IXXAT Automation GmbH
Leibnizstr. 15
D-88250 Weingarten

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

Support

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

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

Copyright

Contents

.NET Programming Manual, Version 1.6

1   System overview

The VCI (Virtual Card Interface) is a driver that is used to enable applications uniform access to different IXXAT CAN interface boards. The following diagram shows the principle structure of the system and its individual components.
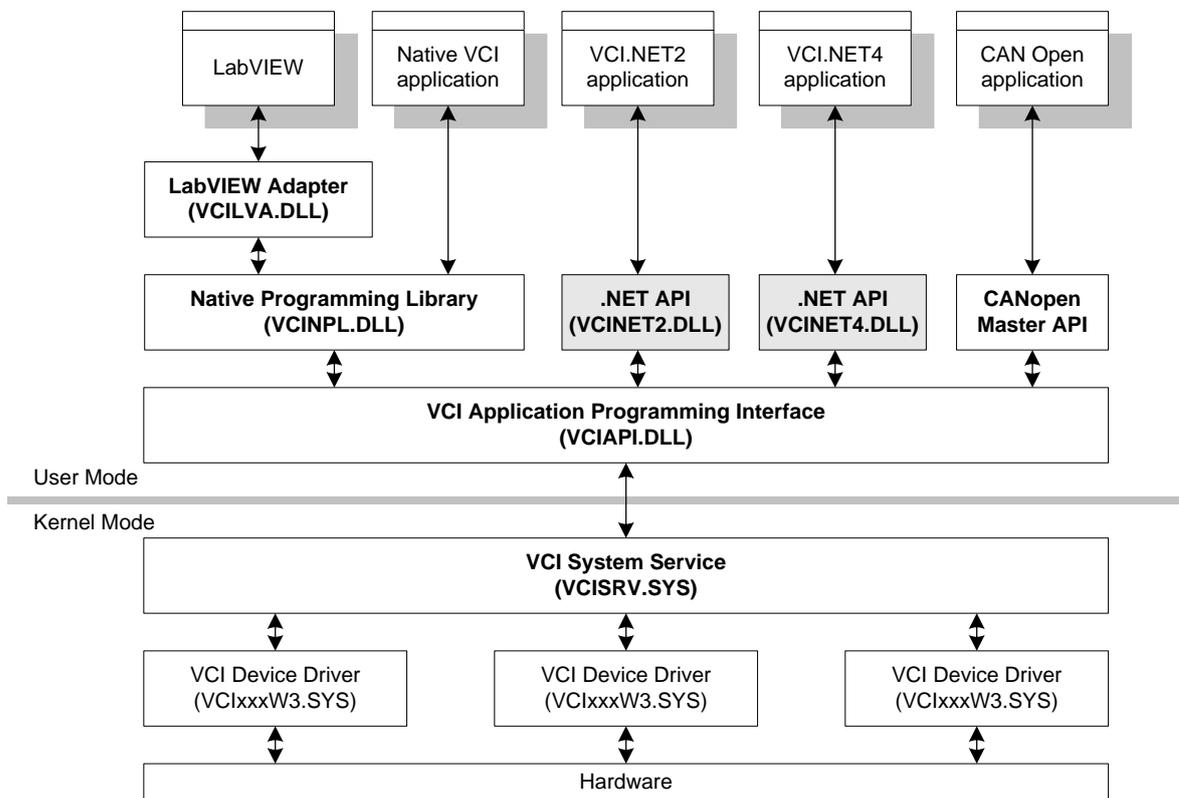
Fig. 1-1: System components

The VCI essentially consists of the following components:

Native VCI programming interface (VCINPL.DLL)

VCI.NET 2.0 programming interface (VCINET2.DLL)

VCI.NET 4.0 programming interface (VCINET4.DLL)

VCI system service API (VCIAPI.DLL)

VCI System Service (VCISRV.SYS)

One or more VCI device drivers (VCIxxxW3.SYS)

The programming interfaces make the connection between the VCI System Service, or VCI Server for short, and the application programs via a set of pre-defined interfaces and functions. The .NET 2.0/4.0 API is only used for adaptation to the COM-based programming interface.

System overview

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

The programming interface discussed in the following consists of the following sub-components and .NET interfaces/classes:

Entry point	VciServer
Device management and device access	**Device Manager** — ○ IVciDeviceManager, ○ IDisposable **Device List** — ○ IVciDeviceList, ○ IDisposable **Device Object** — ○ IVciDeviceList, ○ IDisposable
Bus access	**Bus Access Layer** — ○ IBalObject, ○ IDisposable **Bus Access Layer Resource** — ○ IBalResource, ○ IDisposable **BalResource Collection**
CAN control	**CAN Control** — ○ ICanControl, ○ ICanSocket, ○ IBalResource, ○ IDisposable
CAN message channels	**CAN Channel** — ○ ICanChannel, ○ ICanSocket, ○ IBalResource, ○ IDisposable **CAN Message Reader** — ○ ICanMessageReader, ○ IDisposable **CanMessage** **CAN Message Writer** — ○ ICanMessageWriter, ○ IDisposable
Cyclic CAN transmit list	**CAN Scheduler** — ○ ICanScheduler, ○ ICanSocket, ○ IBalResource, ○ IDisposable **CanCyclicTX Msg**

System overview

LIN control	LIN Control — ○ ILinControl, ○ ILinSocket, ○ IBalResource, ○ IDisposable
LIN message monitor	LIN Monitor — ○ ILinChannel, ○ ILinSocket, ○ IBalResource, ○ IDisposable; LinMessage; LIN Message Reader — ○ ILinMessageReader, ○ IDisposable

2 Device management and device access

2.1 Overview

The components of the device management allow listing of and access to the device drivers and CAN interface boards registered with the VCI Server.
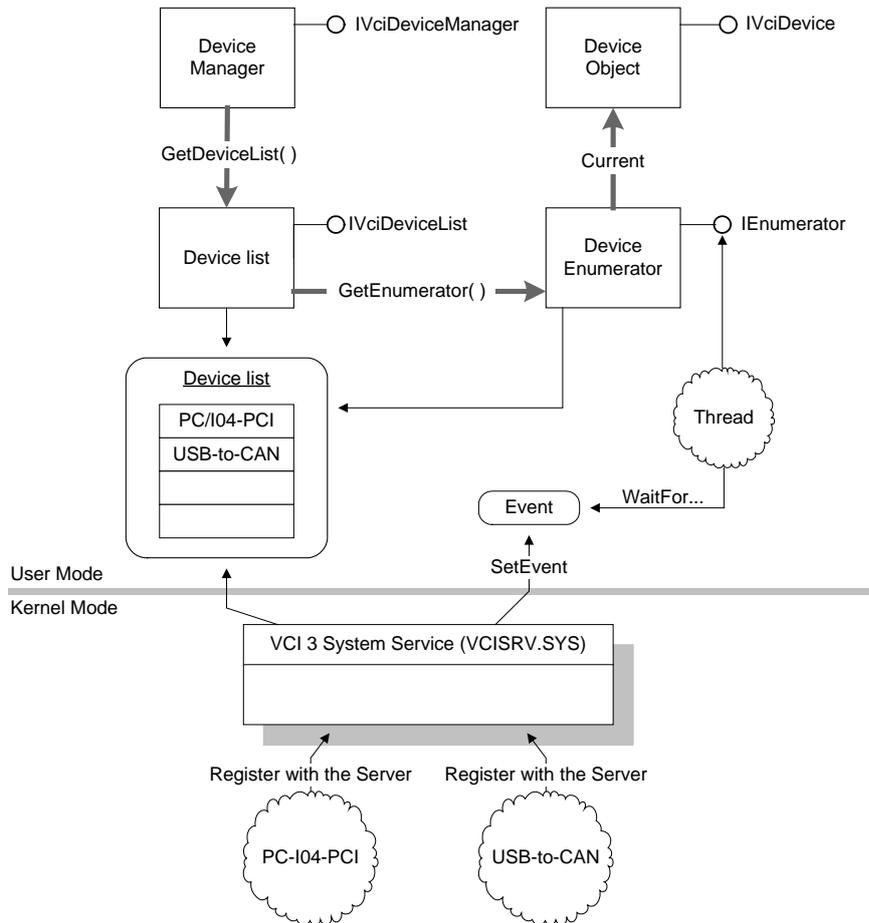

Fig. 2-1: Components of device management

The VCI Server manages all IXXAT CAN interface boards in a system-wide global list, referred to in the following as "device list" for short.

A CAN interface board is automatically registered with the server when the computer is started, or when a connection is made between the computer and the CAN interface board. If a CAN interface board is no longer available, because, for example, the connection was interrupted, it is automatically removed from the device list.

All available IXXAT CAN interface boards are accessed by the device manager, or its interface *IVciDeviceManager*. A reference to this interface is provided by the static method *VciServer.GetDeviceManager*.

2.2 List of available IXXAT CAN interface boards

The global device list is accessed by calling the method *IVciDeviceManager.GetDeviceList*. When successfully run, the method returns a reference to the interface *IVciDeviceList* of the device list with which changes to the device list can be monitored and enumerators requested for the device list.

The method *IVciDeviceList.GetEnumerator* provides the *IEnumerator* interface of a new enumerator object for the device list. Each time it is called, the property *IEnumerator.Current* provides a new device object with information on a CAN interface board. To access this information, the pure object reference provided by the property *Current* of the standard interface *IEnumerator* must be converted to the *IVciDevice* type. The method *IEnumerator.MoveNext* increments an internal index, so that *IEnumerator.Current* can provide a device object for the next CAN interface board. The most important information provided by *IVciDevice* via a CAN interface board is given in the following:

Description: String with the name of the CAN interface board, e.g. USB-to-CAN compact.

VciObjectId: Unique ID of the CAN interface board. Every CAN interface board is allocated a system-wide unique ID when logging in.

DeviceClass: Device class. Every device driver identifies its supported CAN interface board class with a unique ID (*GUID*). Different CAN interface boards belong to different device classes. The IPC-I165/PCI, for example, has a different class than the PC-I04/PCI.

UniqueHardwareId: Hardware ID. Every CAN interface board has a unique ID. The hardware ID can be used, for example, to differentiate between two PC-I04/PCI boards or to search for a CAN interface board with a certain hardware ID.

DriverVersion: Version number of the driver.

HardwareVersion: Version number of the CAN interface board.

Equipment: Technical equipment of the IXXAT CAN interface board. The table of *VciCtrlInfo* structures contained in *Equipment* provides information on the number and type of bus connections present on a CAN interface board. The following diagram shows a CAN interface board with two connections.
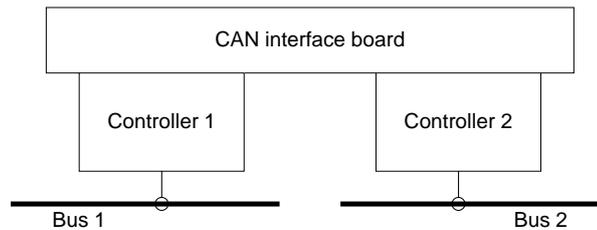
Fig. 2-2: CAN interface board with two bus connections.

The table entry 0 describes bus connection 1, table entry 1 bus connection 2 etc.

The list has been completely traversed when the method *IEnumerator.MoveNext* returns the value **false**.

The internal index can be reset to the beginning with the method *IEnumerator.Reset*, so that a later call of *IEnumerator.MoveNext* resets the enumerator position to the first CAN interface board.

IXXAT CAN interface boards that can be added or removed during operation, such as USB-to-CAN compact, log in on the VCI Server after being inserted or log off again when the CAN interface board is removed.

The login and logout of CAN interface boards also occurs when a device driver is activated or deactivated in the device manager of the operating system device driver (see following Fig. 2-3).
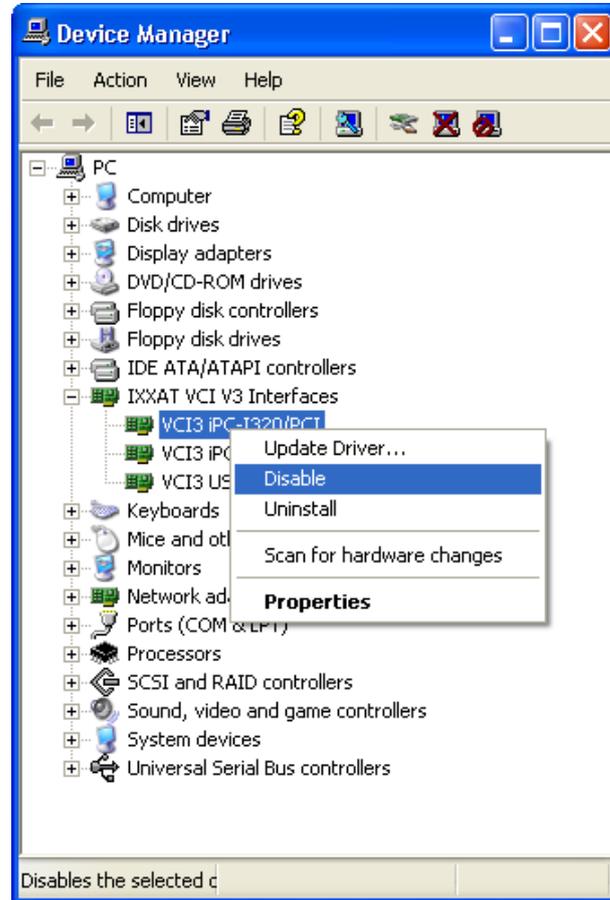

Fig. 2-3: Windows device manager

Applications can monitor changes in the device list by generating an *AutoResetEvent* or a *ManualResetEvent* object and adding it the list via *IVciDeviceList.AssignEvent*. It is recommended to use an *AutoResetEvent* here. If a device logs into or out of the VCI Server after the method is called, the event is set to signaled state.

2.3 Access to an IXXAT CAN interface board

All IXXAT CAN interface boards provide one or more components or access levels for various application areas. However, only the Bus Access Layer (BAL) is of interest here. This allows the controller to be controlled and enables communication with the fieldbus. The BAL can be opened via the method *IVciDevice.OpenBusAccessLayer*.

The various access levels of a CAN interface board cannot be opened simultaneously. For example, if an application opens the Bus Access Layer, the access level used by the CANopen Master API can only be opened again after the BAL has been released or closed.

Certain access levels are also protected against being opened several times. It is therefore not possible, for example, for two CANopen applications to use a CAN interface board simultaneously. However, this restriction does not apply to the BAL.

The BAL can be opened by more than one program simultaneously. It is therefore possible for different applications to access the various bus connections at the same time. Further information on the BAL is given in section 4.

3   Communication components

3.1   First-In First-Out memory (FIFO)

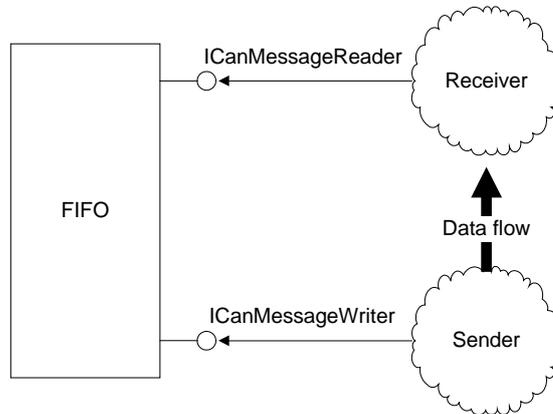The VCI contains an implementation for so-called First-In First-Out memory (FIFO).

Fig. 3-1: FIFO component and data flow

FIFOs are used to transmit data from a transmitter to a receiver. To this end, the transmitter enters data in the FIFO via a writer interface (e.g. *ICanMessageWriter*). A receiver can read out these data again later via a reader interface (e.g. *ICanMessageReader*).

Write and read accesses to a FIFO are possible simultaneously, i.e. the receiver can read data while the writer writes new data. In contrast to this, it is not possible for more than one transmitter or receiver to access a FIFO simultaneously.

Simultaneous access to a FIFO by more than one receiver or transmitter is prevented by the FIFO due to the fact that the relevant reader and writer interfaces (e.g. *ICanMessageReader / ICanMessageWriter*) can only be opened once. An interface can only be opened again when it has been released with *IDisposable.Dispose*. However, this does not prevent different threads of an application from accessing an interface simultaneously.

The programmer must ensure that the functions of an interface are not called by more than one thread at the same time. Otherwise he is responsible for mutual blocking of these calls. As a rule, however, the better alternative is to create a separate message channel for the second thread.

3.1.1 How receive FIFOs work

The following diagram shows how Receive-FIFOs work. The bold arrows show the data flow from the transmitter to the receiver.
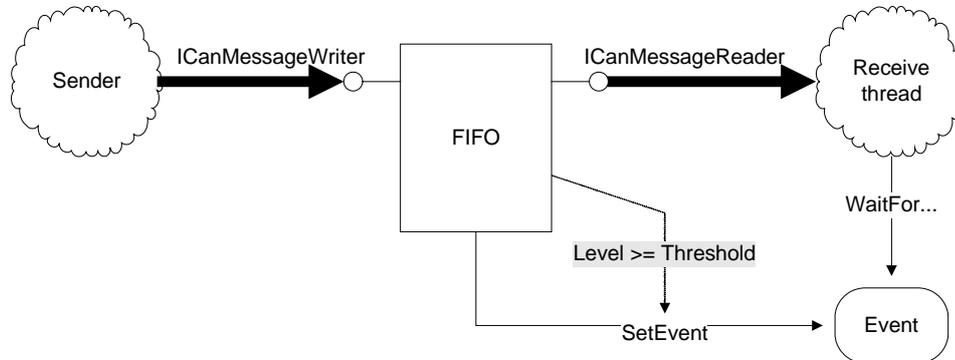

Fig. 3-2: How receive FIFOs work

Receive FIFOs are addressed via a reader interface (here *ICanMessageReader*). Messages to be read individually are accessed via the method *GetMessage.* The method *GetMessages* supplies various FIFO entries via a call

So that a receiver does not have to constantly check whether new data are available, the FIFO can be allocated an event object that is always set to the signaled state when a certain fill level is reached.

To this end, an *AutoResetEvent* or a *ManualResetEvent* is generated and then transferred to the FIFO with the method *AssignEvent*. The threshold, or the fill level at which the event is triggered, can be set with the property *Threshold*.

The application can wait for the occurrence of the event with *WaitOne* or *WaitAll* of the event object. Then the data can be read out of the FIFO. The following sequence diagram shows the time sequence with event-controlled reading of data from the FIFO.
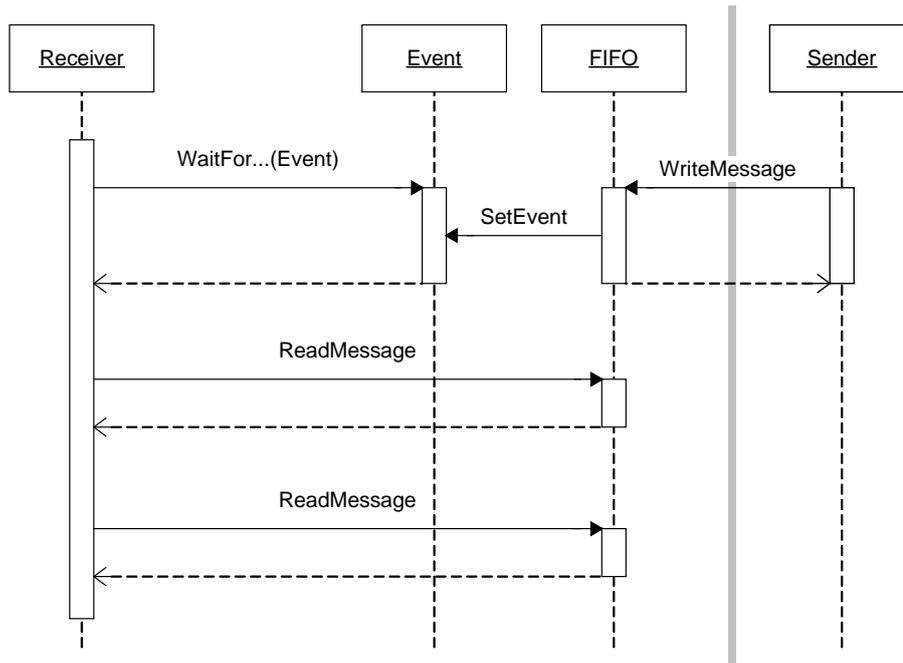
Fig. 3-3: Receive sequence

3.1.2   How transmit FIFOs work

The following diagram shows how transmit-FIFOs work. The thick arrows show the data flow from the transmitter to the receiver.
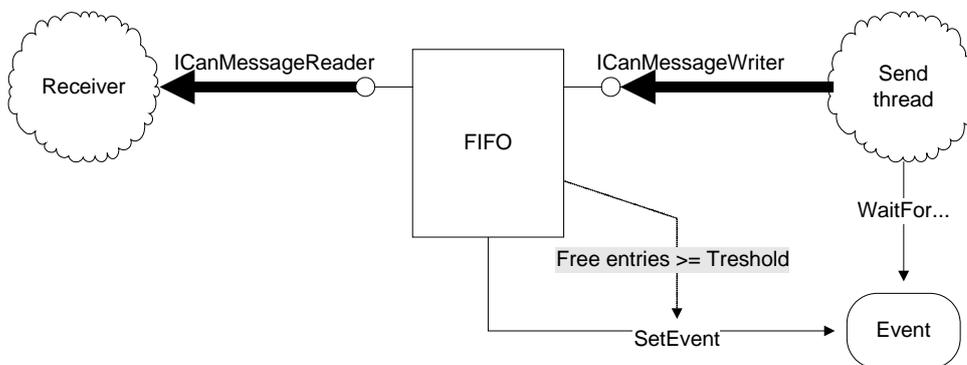
Fig. 3-4: How transmit–FIFOs work

Transmit FIFOs are addressed via a writer interface (here *IcanMessageWriter*). Messages to be transmitted are entered in the FIFO via the method *WriteMessage*. The method *WriteMessage* writes several messages in the FIFO at the same time in one call.

So that a transmitter does not continually have to check whether free elements are available, the FIFO can be allocated an event object, which is always set to the signaled state when the number of free elements exceeds a certain value.

For this, an *AutoResetEvent* or a *ManualResetEvent* is generated and then transmitted to the FIFO with the method *AssignEvent*. The threshold, or the number of free elements at which the event is triggered, can be set with the property *Threshold*.

The application can wait for the occurrence of the event with *WaitOne* or *WaitAll* of the event object. Then data can be written in the FIFO. The following sequence diagram shows the time sequence for event-controlled writing of data in the FIFO.
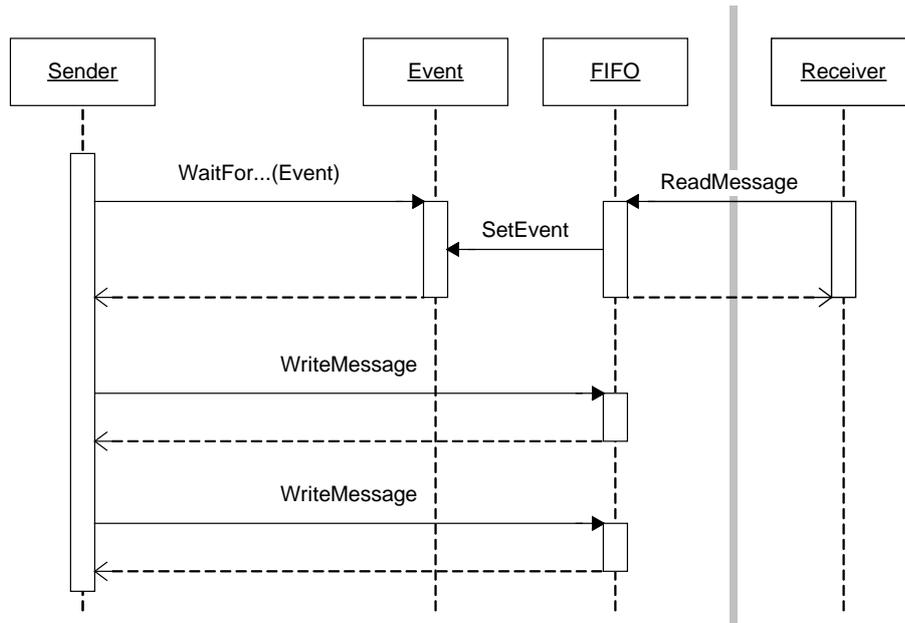

Fig. 3-5: Transmit frequency

4 Access to the fieldbus

4.1 Overview

The fieldbuses connected to the CAN interface board are accessed via the Bus Access Layer (BAL). The following diagram shows all components necessary or provided for the access and the functions to open the BAL of a CAN interface board.
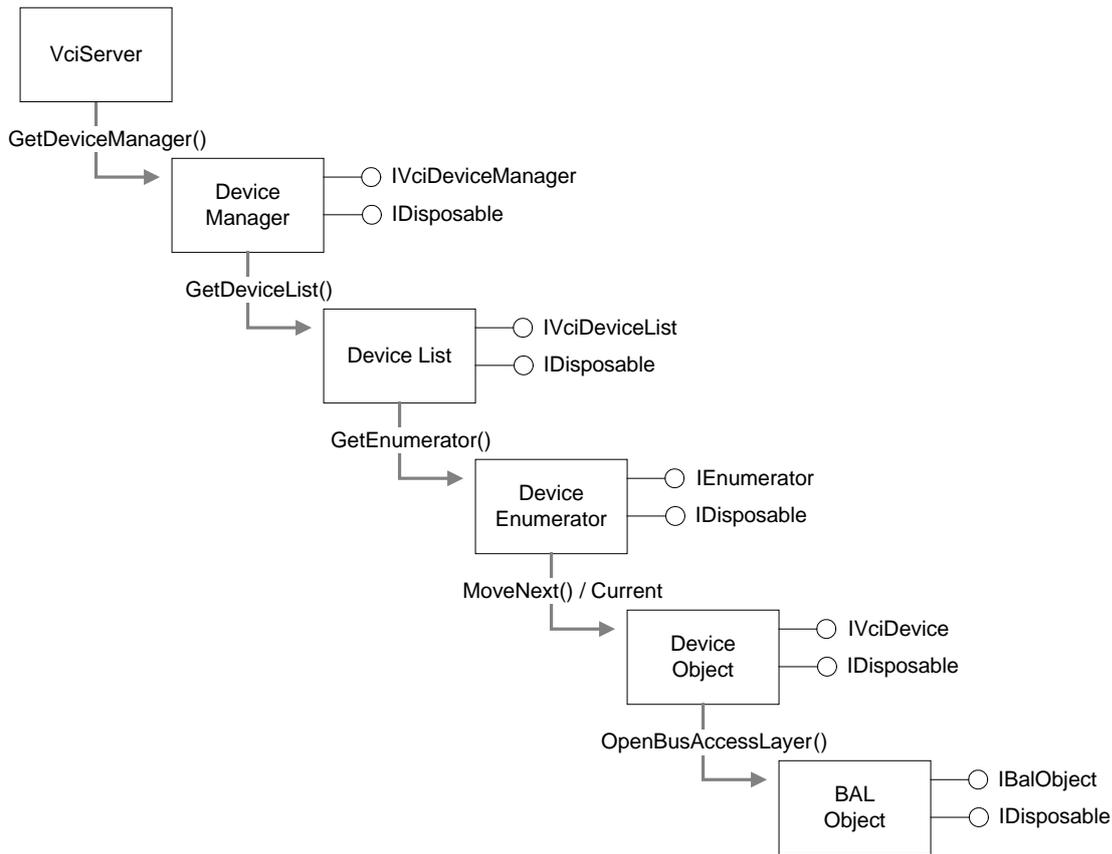
Fig. 4-1: Components for bus access.

In the first step, the required CAN interface board is searched for in the device list (see section 2.2). Then the BAL is opened by calling the method *IVciDevice.OpenBusAccessLayer*.

After the BAL is opened, the references to the device manager, the device list, the device enumerator and the device object are no longer required and can be released with the method *IDisposable.Dispose*. For further work, only the BAL object, or the interface *IBalObject* is required.

The BAL of a CAN interface board can be opened by more than one program at the same time. In addition, in principle it supports more than one and also different types of bus connections. The following diagram shows a CAN interface board with two connections.

Fig. 4-2: BAL with two bus connections

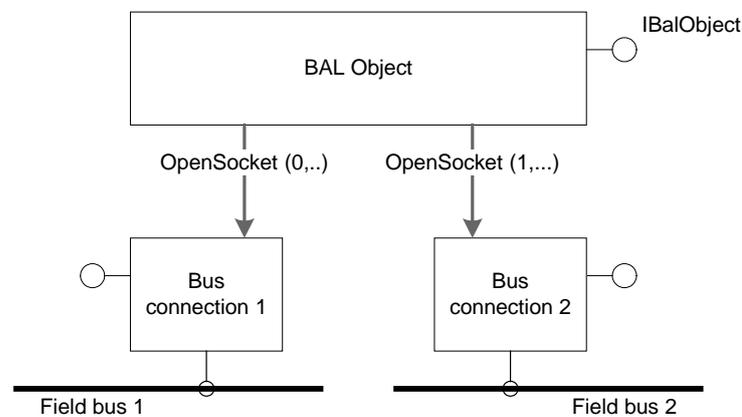The number and type of connections provided can be determined with the property *IBalObject.Resources*. The information is provided by the property in the form of a *BalResourceCollection*, which contains a BAL resource object for each available bus connection.
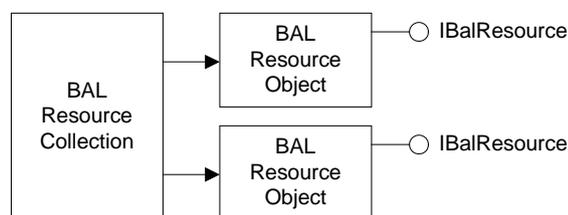
Fig. 4-3: BalResourceCollection with two bus connections.

The version number of the device firmware provides the BAL via the property *IBalObject.FirmwareVersion*.

Access to a connection, or to an interface of the connection, is obtained with the method *IBalObject.OpenSocket*. The method expects the number of the connection to be opened in the first parameter, the value of which must be in the range 0 to *IBalObject.Resources.Count*-1. To open connection 1, the value 0 is entered, for connection 2 the value 1 and so on. In the second parameter, the method expects the type of interface via which the connection is to be accessed. When run successfully, the method returns a reference to the required interface.

The possibilities of interfaces provided by a connection depend on the supported fieldbus. Finally, it should be mentioned that only one single program can access certain interfaces of a connection, whereas others can be accessed by any number of programs at the same time. The rules for access to the individual interfaces also depend on the type of connection and are described in more detail in the following sections.

4.2   CAN connection

4.2.1   Overview

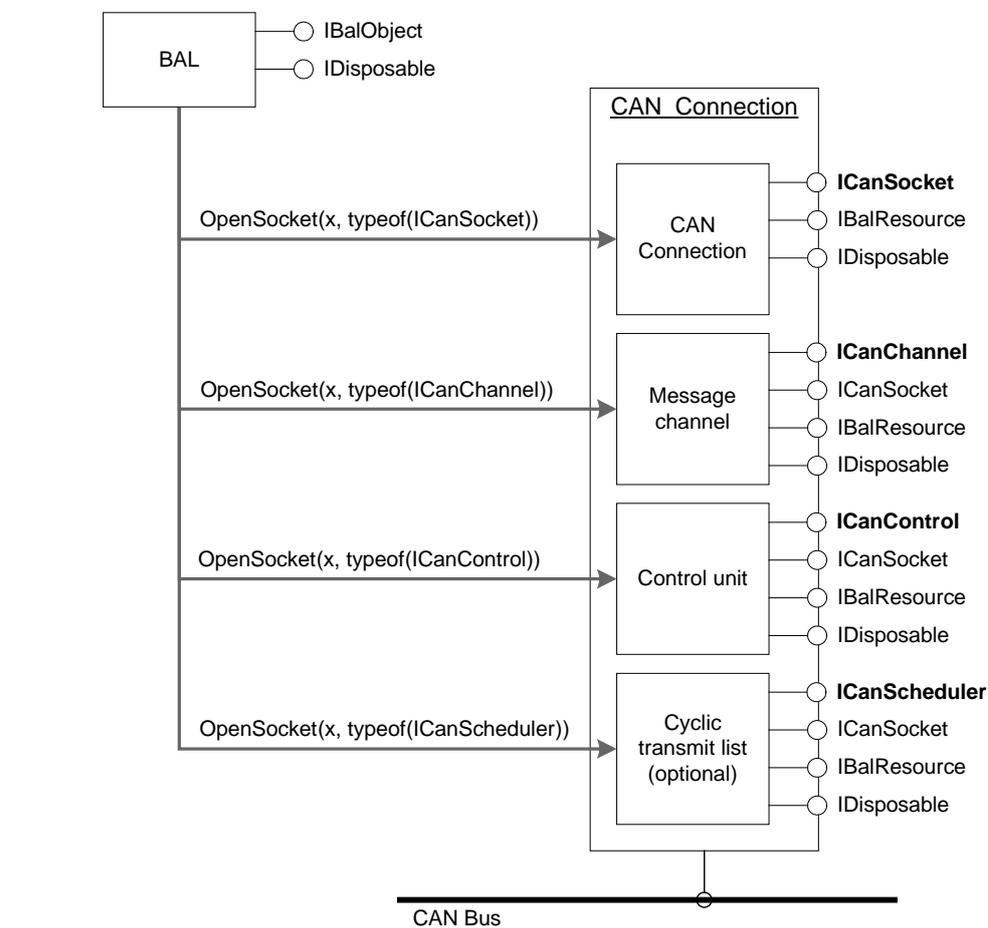Every CAN connection is made up of the sub-components shown in the following diagram.

Fig. 4-4: Components of a CAN connection

Access to the individual sub-components of a CAN connection takes place via the interfaces *ICanSocket*, *ICanChannel*, or *ICanControl*. The optional cyclic transmit list with the interface *ICanScheduler* is normally only available with CAN interface boards that have their own microprocessor.

The interface *ICanSocket* provides functions for requesting the properties of the CAN controller and of the current controller state.

The interface *ICanChannel* represents a message channel. One or more message channels can be set up for the same CAN connection. CAN messages are transmitted and received only via these message channels.

The control unit, or the interface *ICanControl*, provides functions for the configuration of the CAN controller, its transmission properties and functions for the configuration of CAN message filters and to request the current controller state.

With the optionally available cyclic transmit list, up to 16 message objects per connection can be transmitted cyclically, i.e. repeatedly at certain time intervals, via the interface *ICanScheduler*.

Access to the individual components is obtained via the method *IBalObject.OpenSocket*, as already described in section 4.1. Fig. 4-4 shows the interface types to be used for this.

4.2.2   Socket interface

The interface can be opened with the method *IBalObject.OpenSocket*. The type *ICanSocket* is to be entered in the parameter *socketType*. The interface is not subject to any access restrictions and can be opened as often as required and by more than one program at the same time.

The interface *ICanSocket* provides functions for requesting the properties of the CAN controller and the current controller state. However, it is not possible to control the connection.

The properties of a CAN connection as well as the type of the CAN controller, the type of bus coupling and the supported features are provided via numerous properties.

The current operating mode and the current state of the CAN controller can be determined via the property *LineStatus.*

4.2.3 Message channels

A message channel is generated or opened with the method *IBalObject.OpenSocket*. The type *ICanChannel* is to be entered in the parameter *socketType*. Every message channel must be initialized via the method *ICanChannel.Initialize* before it is used. The parameter e*xclusive* determines whether the connection is to be used exclusively. If the value 'true' is entered here, no further channels can be opened after the method is successfully run. If the CAN connection is not used exclusively, in principle any number of message channels can be set up.

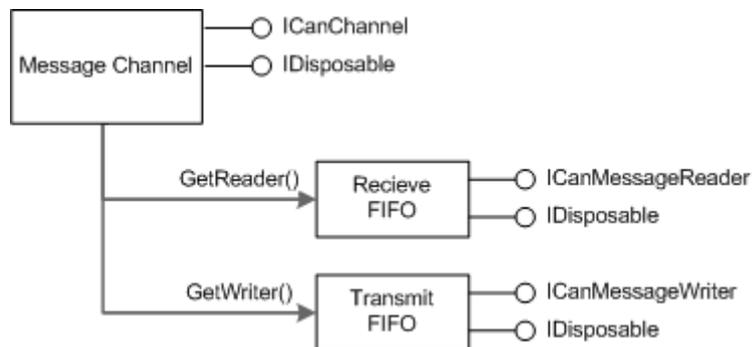A message channel consists of one receive and one transmit FIFO each, as described in section 3.1.

Fig. 4-5: CAN message channel

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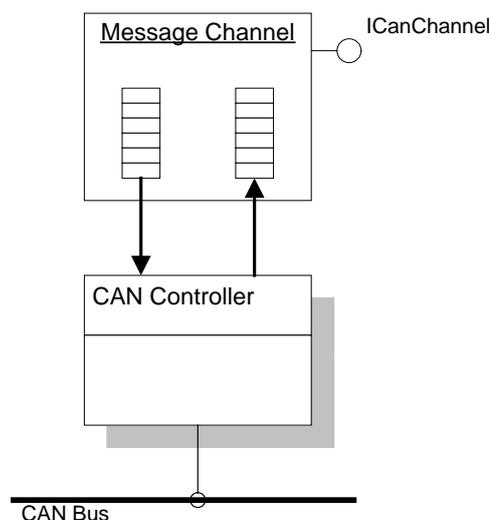

Fig. 4-6: Exclusive use of a CAN message channel

In the case of non-exclusive use of the connection (e*xclusive* = false), a splitter is connected between the controller and the message channels. The splitter re-routes incoming messages from the CAN controller to all message channels and sends the transmit messages of the channels to the controller. The messages are distributed in such a way that no channel receives preferential treatment. The following diagram shows a configuration with three channels on one CAN connection.
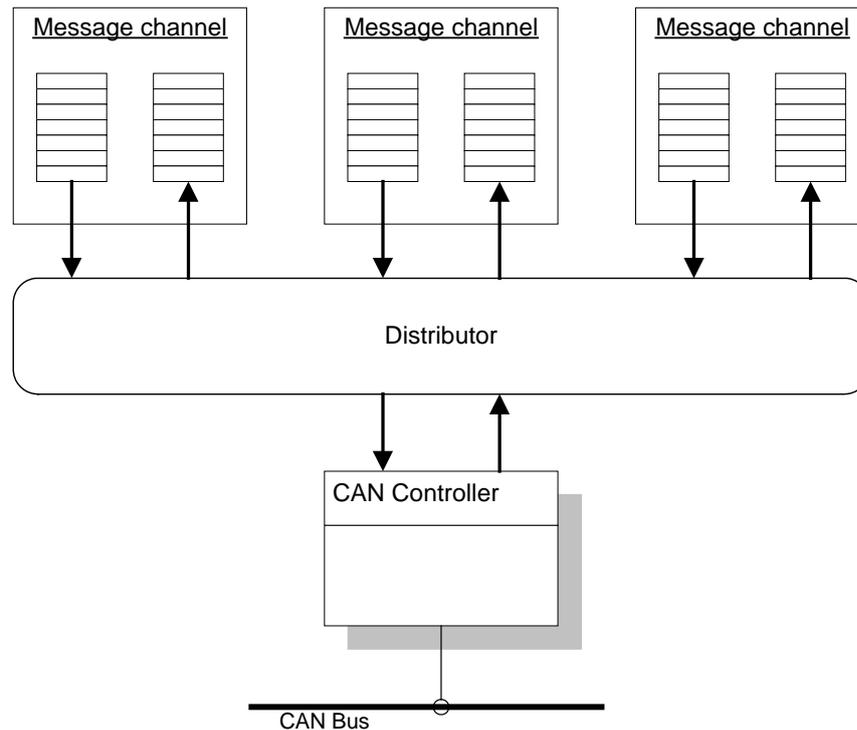
Fig. 4-7: CAN message splitter

At first, a message channel has no receive and transmit FIFOs. They must first be generated by calling the method *ICanChannel.Initialize*. The method expects the size of the individual FIFOs in number of CAN messages as the input parameter.

When the channel is set up, it can be activated with the method *ICanChannel.Activate* and deactivated again with the method *ICanChannel.Deactivate*. A message channel is deactivated after opening (default setting).

Messages are only received from the bus or sent to it when the channel is active and the CAN controller is started. The message filter of the CAN controller also has an influence on the messages received. Further information on the CAN controller is given in section 4.2.4.

4.2.3.1 Receiving CAN messages

The messages received and accepted by the filter are entered in the receive FIFO of a message channel. The interface *ICanMessageReader* is required to read the messages from the FIFO. This can be requested with the method *ICanChannel.GetMessageReader*.

The easiest way to read received messages from the receive FIFO is to call the method *ReadMessage*. The following code fragment shows one possible use of the method.

```
void DoMessages( ICanMessageReader reader )
{
  CanMessage message;

  while( reader.ReadMessage(out message) )
  {
    // Processing of the message
  }

}
```

Another way of reading messages from the receive FIFO, more optimized in terms of data throughput, is to use the method *ReadMessages*. The method is used to read out several CAN messages with one method call. The user creates a field of CAN messages and sends it to the method *ReadMessages*, which attempts to fill it with received messages. The number of messages actually read indicates the method via its return value.

The following code fragment shows a possible use of the functions.

```
void DoMessages( ICanMessageReader reader )
{
  CanMessage[] messages = new CanMessage[10];

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

A detailed description of the FIFOs is given in section 3.1. The way receive FIFOs work is described in section 3.1.1.

4.2.3.2 Transmitting CAN messages

In order to transmit messages, the interface *ICanMessageWriter* of the transmit FIFO is required. This can be requested by the message channel with the method *ICanChannel.GetMessageWriter*.

The easiest way to transmit a message is to call the method *SendMessage*. For this, the method must be given the message to be sent of type *CanMessage* in the parameter *message*. The following code fragment shows one possible use of the method.

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

  // Initialize CAN message.
  message.TimeStamp            = 0;      // no delayed transmission
  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 databyte
  message[0]                   = data;

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

Please note that only messages of type *CanMsgFrameType.Data* can be sent. Other message types are not allowed, or are silently rejected by the CAN controller.

If a value not equal to 0 is given in *TimeStamp*, the message is transmitted to the bus with a delay. Further information on the delayed transmission is given in section 4.2.3.3.

Another way of transmitting messages is to use the method *WriteMessages*. With this method, a pre-defined sequence of messages can be transmitted via one method call. The message sequence is transmitted as a field of CAN messages in the parameter *messages*. The method return value acknowledges the number of messages that could actually be entered in the transmit FIFO.

The following code fragment shows one possible use of the functions.

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

  // Initialize CAN messages.
  message[0].Identifier = 0x100;
  message[0].DataLength = 0;

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

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

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

A detailed description of the FIFOs is given in section 3.1. The way transmit FIFOs work is described in section 3.1.2.

4.2.3.3   Delayed transmission of CAN messages

Connections for which the flag *ICanSocket.SupportsDelayedTransmission* is set support delayed transmission of CAN messages.

Using delayed transmission of messages, it is possible, for example, to prevent a device connected to the CAN bus from receiving too many data in too short a time, which in the case of 'slow' devices can lead to data loss.

To transmit a CAN message with a delay, the minimum time in ticks is given in the field *CanMessage.TimeStamp* which must pass before the message is forwarded to the CAN controller. The value 0 does not trigger delayed transmission, the maximum possible delay time is given in the field *ICanSocket.MaxDelayedTXTicks*. The increment of a tick in seconds is calculated from the values in the fields *ICanSocket.ClockFrequency* and *ICanSocket.DelayedTXTimerDivisor* according to the following formula:

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

The specified delay time only represents a minimum, as it cannot be guaranteed that the message can be transmitted to the bus after the time has expired. It must also be noted that when using more than one message channel on a CAN connection, the specified time values cannot generally be observed, as the splitter processes all channels in parallel. Applications that require an exact time sequence must therefore use the CAN connection exclusively.

4.2.4   Control unit

The control unit, or the interface *ICanControl*, provides methods for the configuration of the CAN controller, its transmission properties as well as functions to configure CAN message filters and to request the current controller state.

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

The interface is opened with the method *IBalObject.OpenSocket*. In the parameter *socketType*, the type *ICanControl* is to be entered. If the method call ends with an exception, the component is already being used by another program.

With the method *IDisposable.Dispose*, an opened control unit can be closed and thus released for other applications. If other interfaces of the connection are open when the control unit is closed, the current controller settings are retained.

4.2.4.1   Controller states

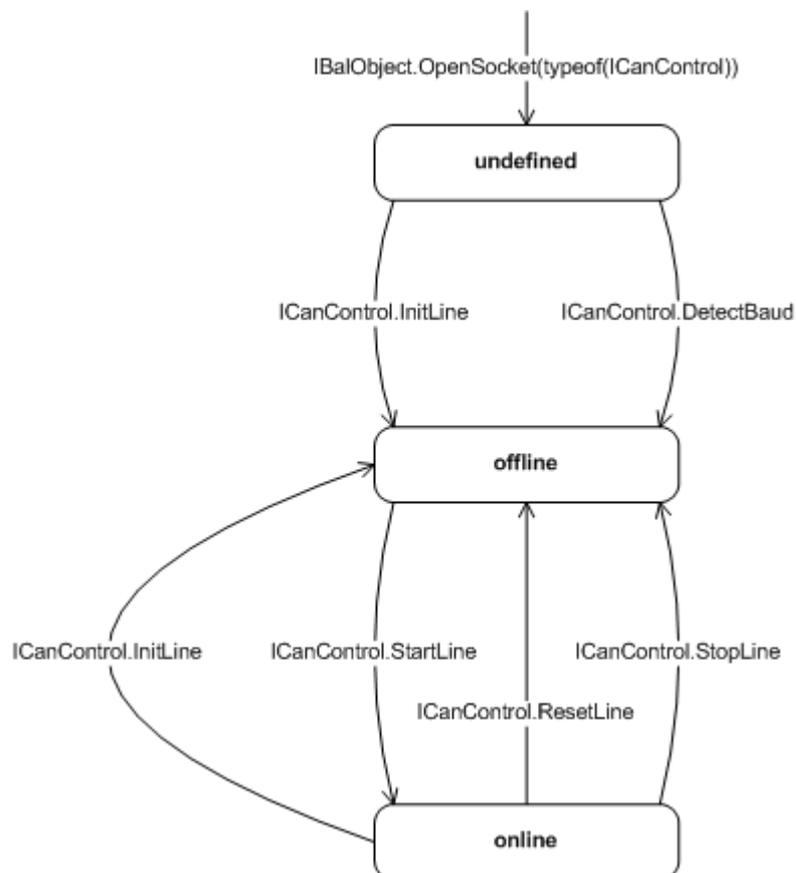The following diagram shows the various states of a CAN controller.

Fig. 4-8: Controller states

Access to the fieldbus

After opening the control unit, or the interface *ICanControl*, the controller is normally in an undefined state. This state is left by calling one of the functions *InitLine* or *DetectBaud*. Then the controller is in 'offline' state.

With *InitLine* the operating mode and bitrate of the CAN controller are set. For this, the method expects values for the parameter *operatingMode* and *bitrate*.

The bitrate is entered in the fields *CanBitrate.Btr0* and *CanBitrate.Btr1*. The values correspond to the values for the registers BTR0 and BTR1 of Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz. Further information on this is given in the data sheet of SJA 1000 in section 6.5. The bus timing values with all CiA or CANopen-compliant bitrates are given in the following table.

Bitrate (KBit)	Pre-defined CiA bitrates	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

If the CAN connection is connected to a running system with an unknown bitrate, the current bitrate of the system can be determined with the method *DetectBaud*. The bus timing values determined by the method can then be transmitted to the method *InitLine* .

The method *DetectBaud* requires a field with pre-defined bus timing values.

The following example shows the use of the method for automatic initialization of a CAN connection on a CANopen system.

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

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

The CAN controller is started by calling the method *StartLine*. After the method is run successfully, the CAN controller is in 'online' state. In this state the CAN controller is actively connected to the bus. Incoming CAN messages are forwarded to all open and active message channels, or transmit messages are transmitted from these to the bus.

The method *StopLine* resets the CAN controller to 'offline' state. Message transport is thus interrupted and the controller deactivated. Calling the method does not alter the set acceptance filters and filter lists not. The method does not simply interrupt an active transmission process of the controller either, but waits until the message has been completely transmitted to the bus.

The method *ResetLine* also sets the CAN controller to the 'offline' state. Unlike *StopLine*, the method resets the controller hardware and deletes all message filters. Please note that resetting the controller hardware leads to faulty message telegrams on the bus if a transmission process is aborted during transmission when the method is called.

The methods *ResetLine* and *StopLine* do not delete the contents of the transmit and receive FIFOs of the message channels.

4.2.4.2   Message filter

Every control unit has a two-stage message filter. The received messages are only filtered based on their ID (CAN-ID), databytes are not considered.

If the 'Self reception request' bit on a transmit message is set, the message is entered in the receive buffer as soon as it has been transmitted on the bus. In this case the message filter is bypassed.
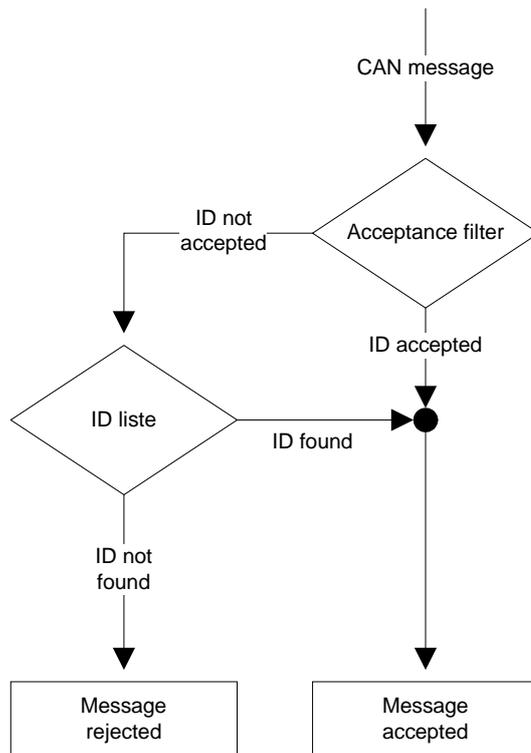

Fig. 4-9: Filter mechanism

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

The CAN controller has separate, mutually independent filters for 11-bit and 29-bit IDs. When the controller is reset or initialized, the filters are set in such a way that all messages are accepted.

The filter settings can be altered with the methods *SetAccFilter*, *AddFilterIds*, and *RemFilterIds*. As input values, the functions expect two bit patterns in the parameters *code* and *mask* which define the ID, or the group of IDs, which are accepted by the filter. A call of the functions is only successful, however, if the controller is in 'offline' state.

The bit patterns in the parameters *code* and *mask* determine which IDs are accepted by the filter. The value of *code* defines the bit pattern of the ID, whereas *mask* defines which bits in *code* are used for the comparison. If a bit in *mask* has the value 0, the corresponding bit in *code* is not for the used for the comparison. If on the other hand it has the value 1, it is relevant for the comparison.

With the 11-bit filter, only the lower 12 bits are relevant. With the 29-bit filter, the bits 0 to 29 are used. All other bits should be set to 0 before calling the method.

The following tables show the connection between the bits in the parameters *code* and *mask*, as well as the bits of the message ID (CAN- ID):

Meaning of the bits of the 11-bit filter:

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

Meaning of the bits of the 29-bit filter:

Bit	29	28	27	26	25	...	5	4	3	2	1	0
	ID28	ID27	ID26	ID25	ID24	...	ID4	ID3	ID2	ID1	ID0	RTR

Bits 11 to 1 or 29 to 1 correspond to the ID bits 10 to 0 and 28 to 0 respectively. Bit 0 always corresponds to the Remote Transmission Request bit (RTR) of a message.

The following example shows the values for the parameters *code* and *mask*, in order to accept only the messages in the range 100h to 103h, for which the RTR bit is simultaneously 0:

code:	**001 0000 0000 0**
mask:	**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**

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

With the method *AddFilterIds*, individual IDs or groups of IDs can be entered in the list and removed from the list again with the method *RemFilterIds*. The parameters *code* and *mask* have the same format as with the acceptance filter.

If the method *AddFilterIds* is called, for example, with the values from the previous example, the method enters the IDs 100h to 103h in the list. If only one single ID is to be entered when the method is called, the required ID (including RTR bit) is entered in *code* and *mask* is set to the value FFFh or 3FFFFFFFh.

The acceptance filter can be completely blocked by calling the method *SetAccFilter* if the value *CanAccCode.None* is entered for *code* and the value *CanAccMask.None* for *mask*. Further filtering is then only carried out based on the ID list. Calling the method with the values *CanAccCode.All* and *CanAccMask.All* on the other hand opens the acceptance filter completely. The ID list is therefore without effect in this case.

4.2.5   Cyclic transmit list

With the optionally available cyclic transmit list, up to 16 messages per CAN connection can be transmitted cyclically, i.e. repeatedly at certain time intervals. It is possible here for a certain part of a CAN message to be automatically incremented after every transmission cycle.

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

The interface is opened with the method *IBalObject.OpenSocket*. In the parameter *socketType* the type *ICanScheduler* must be entered. If the method ends with a *VciException*, the transmit list is already being used by another program. If the CAN connection does not support a cyclic transmit list, *IBalObject.OpenSocket* issues a *NotImplementedException*. With the method *IDisposable.Dispose* an open transmit list is closed and released for other applications.

With the method *ICanScheduler.AddMessage* a message object is added to the list. The method expects a reference to a *CanCyclicTXMsg* object, which specifies the message object that is to be added to the list.

The cycle time of a transmit object is given in number of ticks in the field *CanCyclicTXMsg.CycleTicks*. The value in this field must be more than 0 and must not exceed the value in the field *ICanSocket.MaxCyclicMsgTicks*.

The duration of a tick, or the cycle time t_z of the transmit list can be calculated with the fields *ICanSocket.ClockFrequency* and *ICanSocket.CyclicMessage-TimeDivisor* according to the following formula.

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

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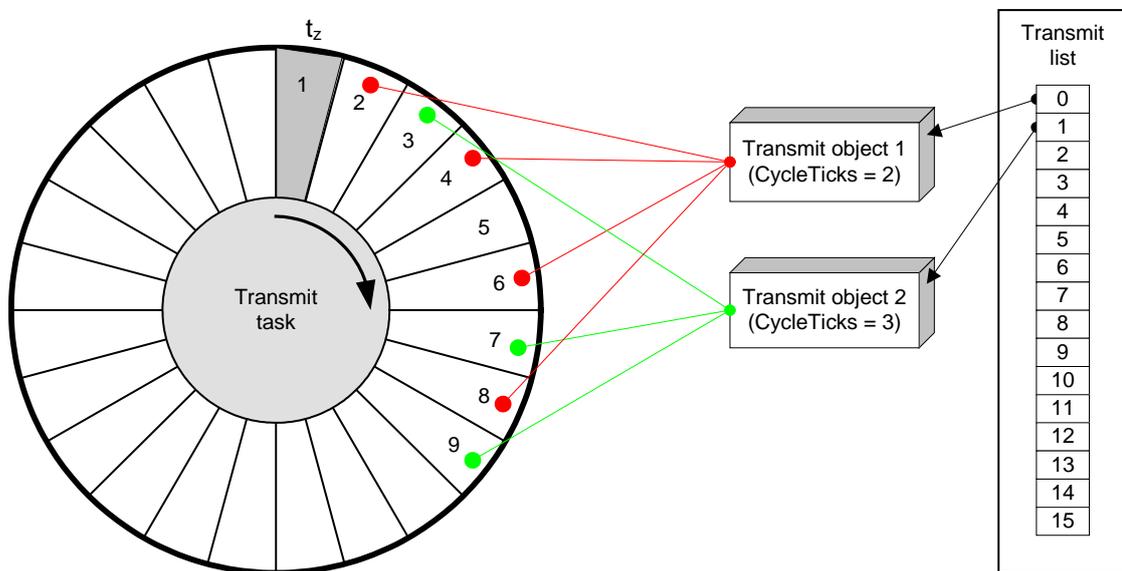

Fig. 4-10: Transmit task of the cyclic transmit list

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

The example in Fig. 4-10 shows two transmit objects with the cycle times 2 and 3. When creating transmit object 1, the time slots 2, 4, 6, 8 etc. are occupied. When subsequently creating the second object (cycle time = 3), collisions occur in the time slots 6, 12, 18 etc., as these time slots are already occupied by object 1.

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

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

The field *CanCyclicTXMsg.AutoIncrementMode* determines whether part of the message is automatically incremented after every transmit cycle. If the value *CanCyclicTXIncMode.NoInc* is entered here, the content remains unchanged. With the value *CanCyclicTXIncMode.IncId*, the *Identifier* field of the message is automatically incremented by one after every transmit cycle. If the *Identifier* filed reaches the value 2048 (11-bit ID) or 536.870.912 (29-bit ID), an automatic overrun to 0 occurs.

With the value *CanCyclicTXIncMode.Inc8* or *CanCyclicTXIncMode.Inc16* in the field *CanCyclicTXMsg.AutoIncrementMode*, an individual 8-bit or 16-bit value is incremented in the data field of the message. The field *AutoIncrementIndex* defines the index of the data field. With 16-bit values, the least significant byte (LSB) is in the data field *Data[AutoIncrementIndex]* and the most significant byte (MSB) in the field *Data[AutoIncrementIndex +1]*. If the value 255 (8-bit) or 65535 (16-bit) is reached, an overrun to 0 occurs.
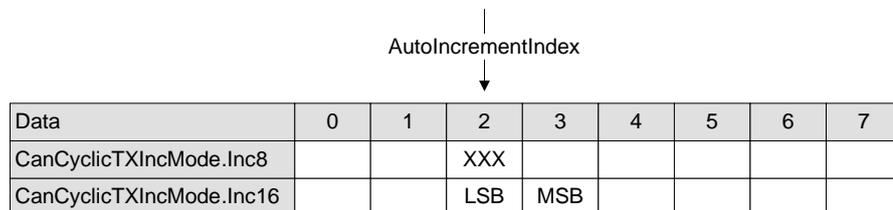
AutoIncrementIndex

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

Fig. 4-11: Auto-increment of data fields

With the method *RemMessage,* a transmit object can be removed from the list again. For this, the method expects a reference of a message object added via the method *AddMessage*.

A newly created transmit object is at first in standby state and is not transmitted by the transmit task until it is started by calling the method *StartMessage*. The transmit process for an object can be stopped with the method *StopMessage*.

The state of an individual transmit object can be requested via its *Status* property. However, the transmit object status must be updated manually via the method *UpdateStatus* of the associated transmit list.

The transmit task is normally deactivated after the transmit list is opened. In principle, the transmit task does not transmit any messages in deactivated state, even if the list contains created and started transmit objects.

The transmit task of a transmit list can be activated or deactivated by calling the method *Resume*.

The method can be used to start all transmit objects simultaneously, by first starting all transmit objects via *StartMessage* and only then activating the transmit task. It is also possible to stop all objects simultaneously. For this, the transmit task must be deactivated via the method *Suspend.*

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

4.3   LIN connection

4.3.1   Overview

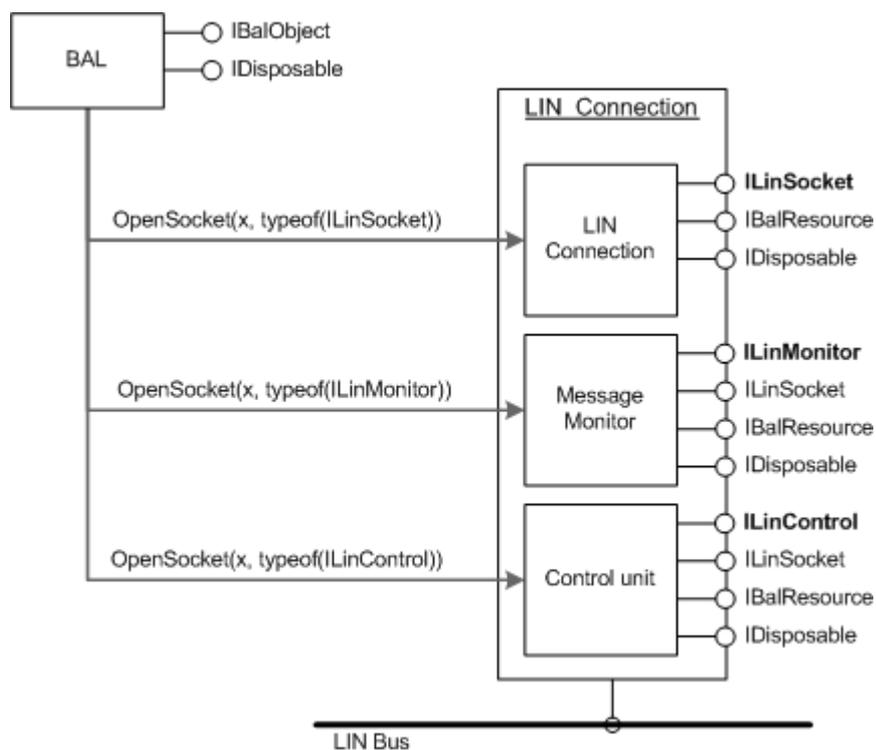Every LIN connection is made up of the sub-components shown in the following diagram.

Fig. 4-12: Components of a LIN connection

Access to the individual sub-components of a LIN connection takes place via the interfaces *ILinSocket*, *ILinMonitor*, or *ILinControl*.

The interface *ILinSocket* provides methods for requesting the properties of the LIN controller and of the current controller state.

The interface *ILinMonitor* represents a message monitor. One or more message monitors can be set up for the same LIN connection. LIN messages are received only via these message monitors.

The control unit, or the interface *ILinControl*, provides methods for the configuration of the LIN controller, its transmission properties and properties to request the current controller state.

Access to the individual components is obtained via the method *IBalObject.OpenSocket*, as already described in section 4.1. Fig. 4-12 shows the interface types to be used for this.

4.3.2   Socket interface

The interface *ILinSocket* can be opened with the method *IBalObject.OpenSocket*. The type *ILinSocket* is to be entered in the parameter *socketType*. The interface is not subject to any access restrictions and can be opened as often as required and by more than one program at the same time.

The interface *ILinSocket* provides functions for requesting the properties of the LIN controller and the current controller state. However, it is not possible to control the connection.

The properties of a LIN connection such as the supported features are provided via properties.

The current operating mode and the current state of the LIN controller can be determined via the property *LineStatus.*

4.3.3   Message monitors

A message monitor is generated or opened with the method *IBalObject.OpenSocket*. The type *ILinMonitor* is to be entered in the parameter *socketType*. Every message monitor must be initialized via the method *ILinMonitor.Initialize* before it is used. The parameter *exclusive* determines whether the connection is to be used exclusively. If the value *true* is entered here, no further monitors can be opened after the method is successfully run. If the LIN connection is not used exclusively, in principle any number of message monitors can be set up.

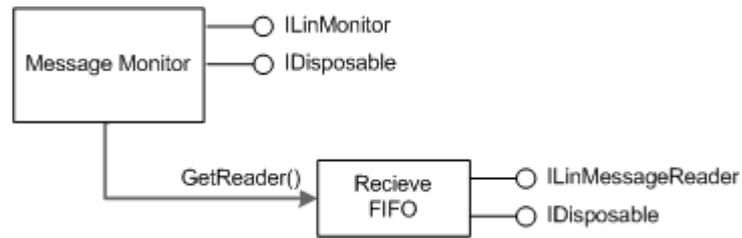A message monitor consists of one receive FIFO, as described in section 3.1 for CAN messages.

Fig. 4-13: LIN message monitor

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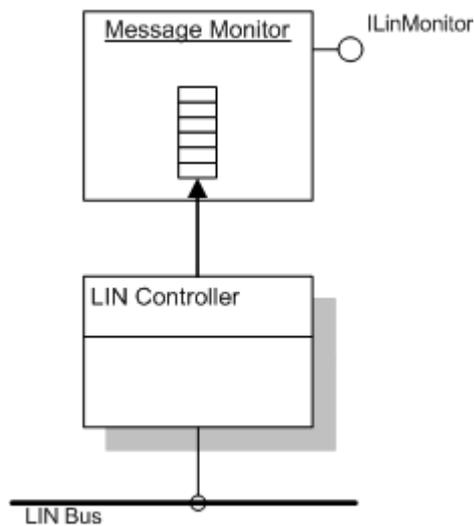

Fig. 4-14: Exclusive use of a LIN message monitor

In the case of non-exclusive use of the connection (e*xclusive* = false), a splitter is connected between the controller and the message monitors. The splitter re-routes incoming messages from the LIN controller to all message monitors. The messages are distributed in such a way that no monitor receives preferential treatment. The following diagram shows a configuration with three monitors on one LIN connection.
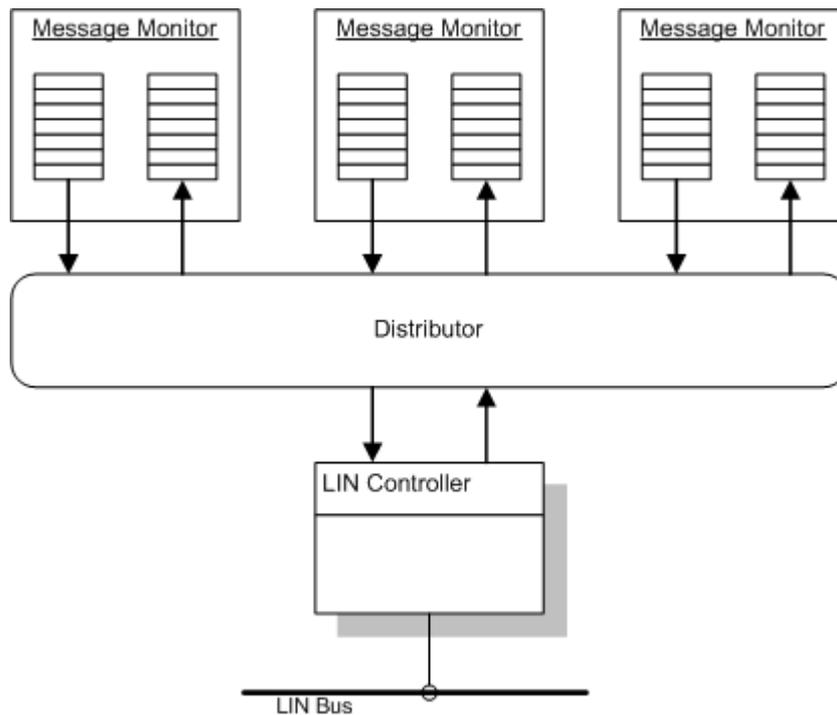
Fig. 4-15: LIN message splitter

At first, a message monitor has no receive FIFO. It first must be generated by calling the method *ILinMonitor.Initialize*. The method expects the size of the FIFO in number of LIN messages as input parameter.

When the monitor is set up, it can be activated with the method *ILinMonitor.Activate* and deactivated again with the method *ILinMonitor.Deactivate*. A message monitor is deactivated after opening (default setting).

Messages are only received from the bus when the monitor is active and the LIN controller is started. Further information on the LIN controller is given in section 4.3.4.

4.3.3.1   Receiving LIN messages

The received messages are entered in the receive FIFO of a message monitor. The interface *ILinMessageReader* is required to read the messages from the FIFO. This can be requested with the method *ILinMonitor.GetMessageReader*.

The easiest way to read received messages from the receive FIFO is to call the method *ReadMessage*. The following code fragment shows one possible use of the method.

```
void DoMessages( ILinMessageReader reader )
{
  LinMessage message;

  while( reader.ReadMessage(out message) )
  {
    // Processing of the message
  }

}
```

Another way of reading messages from the receive FIFO, more optimized in terms of data throughput, is to use the method *ReadMessages*. This method is used to read out several LIN messages with one method call. The user creates a field of LIN messages and sends it to the method *ReadMessages*, which attempts to fill it with received messages. The number of messages actually read indicates the method via its return value.

The following code fragment shows a possible use of the functions.

```
void DoMessages( ILinMessageReader reader )
{
  LinMessage[] messages = new LinMessage[10];

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

A detailed description of the FIFOs is given in section 3.1. The way receive FIFOs work is described in section 3.1.1 exemplarily for CAN messages.

4.3.4   Control unit

The control unit, or the interface *ILinControl*, provides methods for the configuration of the LIN controller, its transmission properties and to request the current controller state.

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

The interface is opened with the method *IBalObject.OpenSocket*. In the parameter *socketType*, the type *ILinControl* is to be entered. If the method call ends with an exception, the component is already being used by another program.

With the method *IDisposable.Dispose*, an opened control unit can be closed and thus released for other applications. If other interfaces of the connection are open when the control unit is closed, the current controller settings are retained.

4.3.4.1 Controller states

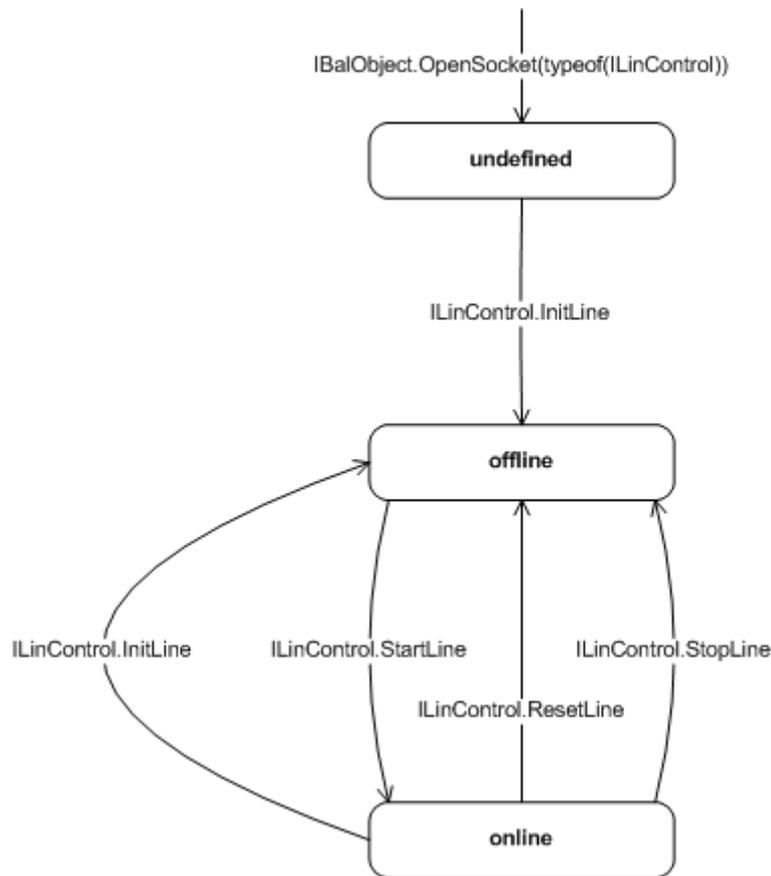The following diagram shows the various states of a LIN controller.

Fig. 4-16: Controller states

After opening the control unit, or the interface *ILinControl*, the controller is nor-mally in an undefined state. This state is left by calling the method *InitLine.* Then the controller is in 'offline' state.

With *InitLine* the operating mode and bitrate of the LIN controller are set. For this, the method expects a *LinInitLine* structure containing values for operating mode and bitrate.

The bitrate in bits per second is defined in the field *LinInitLine.Bitrate*. Valid values for the bitrate are between 1000 and 20000, or between *LinBitrate.MinBitrate* and *LinBitrate.MaxBitrate*. If the connection supports automatic bitrate detection, this can be activated by using *LinBitrate.AutoRate*. The following table shows some recommended bitrates:

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

The LIN controller is started by calling the method *StartLine*. After the method is run successfully, the LIN controller is in 'online' state. In this state the LIN controller is actively connected to the bus. Incoming LIN messages are forwarded to all open and active message monitors.

The method *StopLine* resets the LIN controller to 'offline' state. Message transport is thus interrupted and the controller deactivated. The method does not simply interrupt an active transmission process of the controller either, but waits until the message has been completely transmitted to the bus.

The method *ResetLine* also sets the LIN controller to the 'offline' state. Unlike *StopLine*, the method resets the controller hardware. Please note that resetting the controller hardware leads to faulty message telegrams on the bus if a transmission process is aborted during transmission when the method is called.

The methods *ResetLine* and *StopLine* do not delete the contents of the receive FIFO of the message monitors.

4.3.4.2   Transmission of LIN messages

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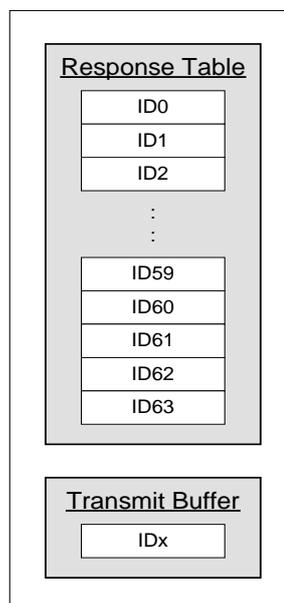

Fig. 4-17: Internal structure of the control unit

The control unit contains an internal response table with the response data for the IDs transmitted by the master. If the controller detects an ID on the bus which is assigned to it and has been send by the master, it transmits the response data entered in the table at the corresponding position. The contents of the table can be changed or updated with the method *ILinControl.WriteMessage* by entering the value *false* in the parameter *send*. The message with the response data in the data field of the structure *LinMessage* is transferred to the method in the parameter *message*. Note that the message is of type *LinMessageType.Data* and contains a valid ID in the range 0 to 63. The table must be initialized before the controller is started, irrespective of the operating mode (master or slave), but can be updated at any time without stopping the controller. The response table is emptied when the method *ILinControl.ResetLine* is called.

With the method *ILinControl.WriteMessage* messages can also be transmitted directly to the bus. For this, parameter *send* must be set to the value *true*. In this case the message is not entered in the response table but instead in the transmit buffer and transmitted to the bus, as soon as it is free, by the controller.

If the connection is operated as a master, in addition to the control messages *LinMessageType.Sleep* and *LinMessageType.Wakeup*, data messages of type *LinMessageType.Data* can also be transmitted directly.

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

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

In the master mode, the method *ILinControl.WriteMessage* is also used to send IDs. For this, a *LinMessageType.Data* message with a valid ID and data length is transmitted, where the flag *IdOnly* simultaneously has the value *true*.

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

5 Description of the interface

A detailed description of the VCI V3 .NET 2.0/4.0 interfaces and classes is given in the online reference **vcinet2.chm/ vcinet4.chm** also installed in the sub-directory **dotnet**.