

# VCI Frame and Signal API

Software Version 1.0

## SOFTWARE DESIGN GUIDE

4.02.0250.20026 1.0 en-US ENGLISH

---

# Important User Information

## Disclaimer

The information in this document is for informational purposes only. Please inform HMS Industrial Networks of any inaccuracies or omissions found in this document. HMS Industrial Networks disclaims any responsibility or liability for any errors that may appear in this document.

HMS Industrial Networks reserves the right to modify its products in line with its policy of continuous product development. The information in this document shall therefore not be construed as a commitment on the part of HMS Industrial Networks and is subject to change without notice. HMS Industrial Networks makes no commitment to update or keep current the information in this document.

The data, examples and illustrations found in this document are included for illustrative purposes and are only intended to help improve understanding of the functionality and handling of the product. In view of the wide range of possible applications of the product, and because of the many variables and requirements associated with any particular implementation, HMS Industrial Networks cannot assume responsibility or liability for actual use based on the data, examples or illustrations included in this document nor for any damages incurred during installation of the product. Those responsible for the use of the product must acquire sufficient knowledge in order to ensure that the product is used correctly in their specific application and that the application meets all performance and safety requirements including any applicable laws, regulations, codes and standards. Further, HMS Industrial Networks will under no circumstances assume liability or responsibility for any problems that may arise as a result from the use of undocumented features or functional side effects found outside the documented scope of the product. The effects caused by any direct or indirect use of such aspects of the product are undefined and may include e.g. compatibility issues and stability issues.

---

# Table of Contents

Page

<b>1</b>	<b>User Guide .....</b>	<b>3</b>
1.1	Related Documents .....	3
1.2	Document History .....	3
1.3	Conventions .....	4
<b>2</b>	<b>System Overview .....</b>	<b>5</b>
2.1	VCI Components .....	5
2.2	Components of the Frame and Signal API .....	6
2.2.1	Message Based Clients .....	7
2.2.2	Signal Based Clients .....	8
2.2.3	CAN Specific Components .....	10
<b>3</b>	<b>Communication .....</b>	<b>13</b>
3.1	Signal Based Communication .....	13
3.1.1	Accessing and Initializing the Signal Set .....	13
3.1.2	Converting Signal Values .....	15
3.1.3	Reading Receive Signal Sets .....	15
3.1.4	Writing Transmit Signal Sets .....	16
3.1.5	Deactivating and Releasing the Signal Set .....	16
3.2	CAN Specific Communication .....	17
3.2.1	Creating a Message Switch .....	17
3.2.2	Initializing and Activating the Message Switch .....	18
3.2.3	Creating and Initializing Clients: Message Sinks .....	19
3.2.4	Creating and Initializing Clients: Message Sources .....	22
3.2.5	Disconnecting Clients .....	27

---

<b>4</b>	<b>API Functions .....</b>	<b>28</b>
4.1	Exported Functions.....	28
4.1.1	VciCreateCanMsgSwitch .....	28
4.2	Interface IUnknown .....	29
4.2.1	QueryInterface .....	29
4.2.2	AddRef.....	29
4.2.3	Release .....	30
4.3	Signal Specific Interfaces .....	31
4.3.1	ISignalSet.....	31
4.3.2	IRSignalSet .....	37
4.3.3	ITSignalSet .....	38
4.4	CAN Specific Interfaces .....	39
4.4.1	Message Switch: ICanMsgSwitch .....	39
4.4.2	Message Sink: ICanRMsgBuffer .....	43
4.4.3	Message Sink: ICanRMsgQueue .....	45
4.4.4	Message Sink: ICanRMsgSet .....	47
4.4.5	Message Source: ICanTMsgBuffer.....	49
4.4.6	Message Source: ICanTMsgQueue .....	51
4.4.7	Message Source: ICanTMsgSet .....	53
<b>5</b>	<b>Data Structures.....</b>	<b>55</b>
5.1	CAN Specific Data Types .....	55
5.1.1	CANMSGSWITCHSTATUS .....	55
5.2	Signal Specific Data Types.....	55
5.2.1	FSLVAR .....	55
5.2.2	FLSIGNAL.....	56

# 1 User Guide

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

## 1.1 Related Documents

Document	Author
VCI: C API Software Version 3/4 Software Design Guide	HMS
VCI Driver Installation Guide	HMS

## 1.2 Document History

Version	Date	Description
1.0	November 2019	First release

## 1.3 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. 4](#)

This is an external link (URL): [www.hms-networks.com](http://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.

## 2 System Overview

### 2.1 VCI Components

The VCI frame and signal library (VCIFSL) is an API extension of the VCI that provides components and functions to simplify accessing messages and processing signals and messages. In this guide the VCI frame and signal library VCIFSL.DLL is described.

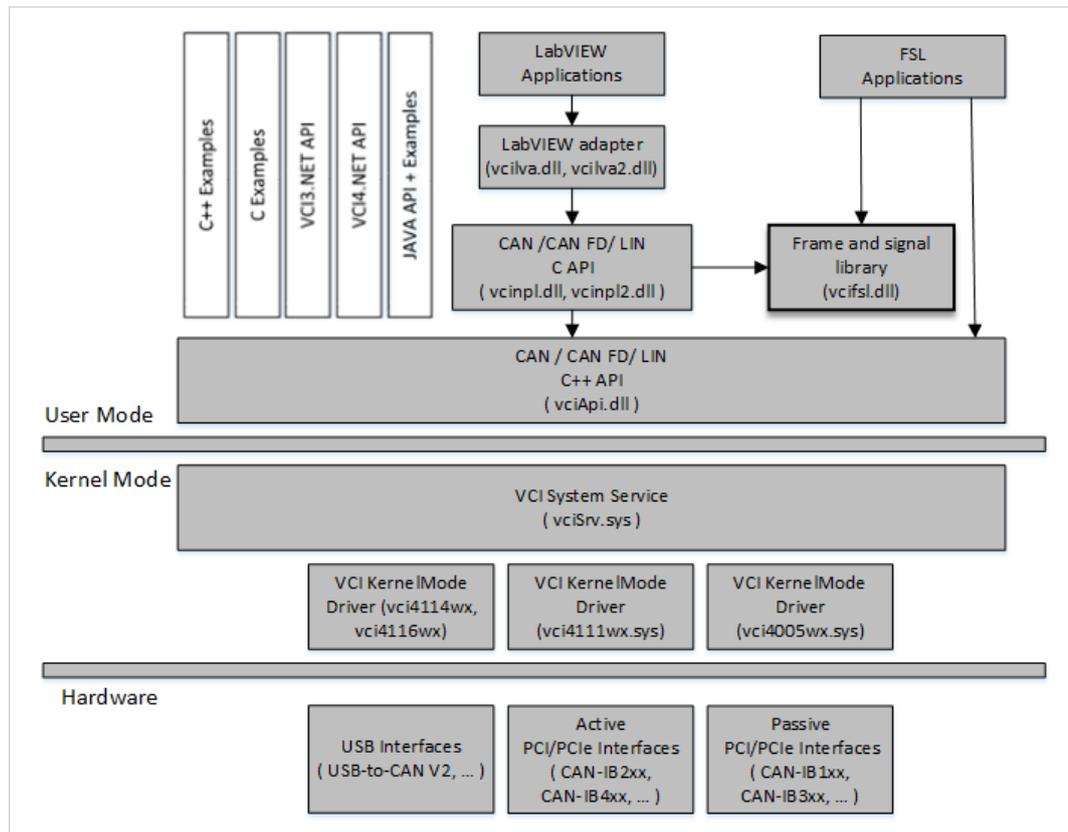


Fig. 1 System structure and components

The VCI system service manages the individual VCI device drivers, the access to the various interface boards and bus adapters, and provides mechanisms for the exchange of data and commands between user mode and kernel mode. The components of the User Mode provide the connection between the VCI System Service and the various application programs. The frame and signal library provides the API and programming interfaces via components that are designed according to the Microsoft Component Object Model (MS-COM).

All provided components implement the interface `IUnknown`, that is defined by MS-COM. The server functionality that is specified in MS-COM is not implemented. The components do not have a COM conform fabric or automation interface. Therefore the VCI specific components are not created with `IClassFactory` and do not have an `IDispatch` interface. They can not be used by script or .NET languages.

Regarding multi threading, simultaneous access to particular components from several threads is possible. Every thread has to open an own instance of the desired component or interface. The individual functions of an interface must not be called by different threads, because the implementation is not thread safe due to performance reasons.

The components of the frame and signal library VCIFSL.DLL do not have to be assigned to an apartment, as usual in COM. If the VCI specific components are used exclusively, without any other COM components the particular threads of an application do not have to be assigned to an apartment (MTA) nor create an apartment (STA) and therefore do not have to call the function

`CoInitialize()` or `CoInitializeEx()`. For more information see Microsoft Visual Studio Help in chapter *Processes, Threads and Apartments* or description of functions `CoInitialize()` and `CoInitializeEx()`.

## 2.2 Components of the Frame and Signal API

The VCI frame and signal API is an extension for VCI message channels. The central component is the message switch, to which all clients are connected.

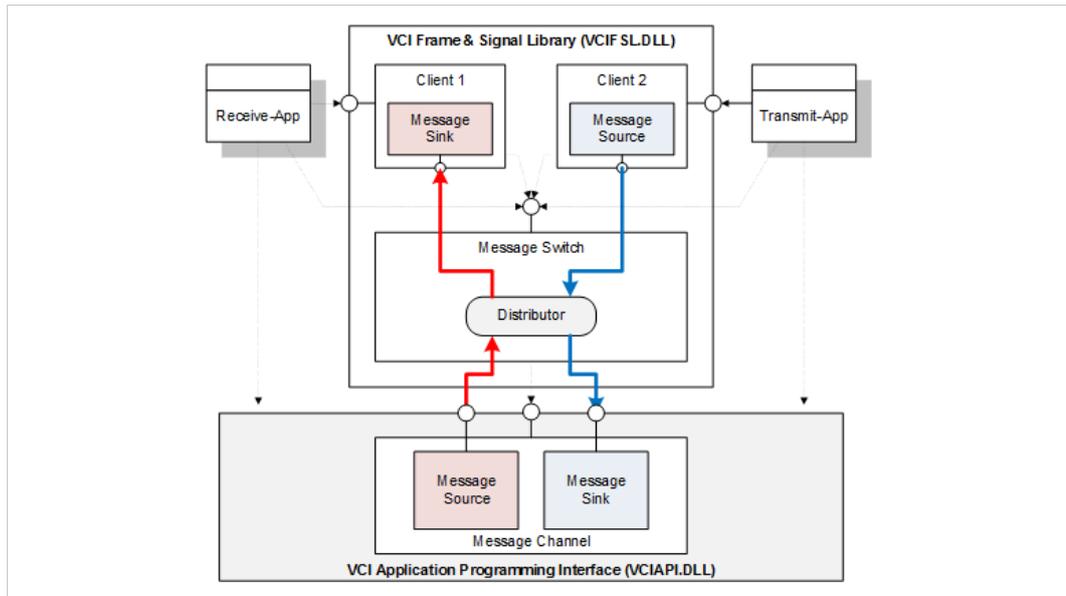


Fig. 2 Components

Two different client types are connected to the message switch:

- message sink: receiver, destination of a message
- message source: transmitter, source of a message

The time and event controlled distributor of the message switch transmits message from VCI internal message sources to the receiving client and vice versa from the transmitting client to VCI internal message sinks. Normally the distributor handles the clients in the same sequence as the clients were registered at the distributor. The processing time depends on the number of connected clients.

The clients can either be message based or signal based.

### 2.2.1 Message Based Clients

Message based clients provide interfaces for applications to transmit and to receive messages.

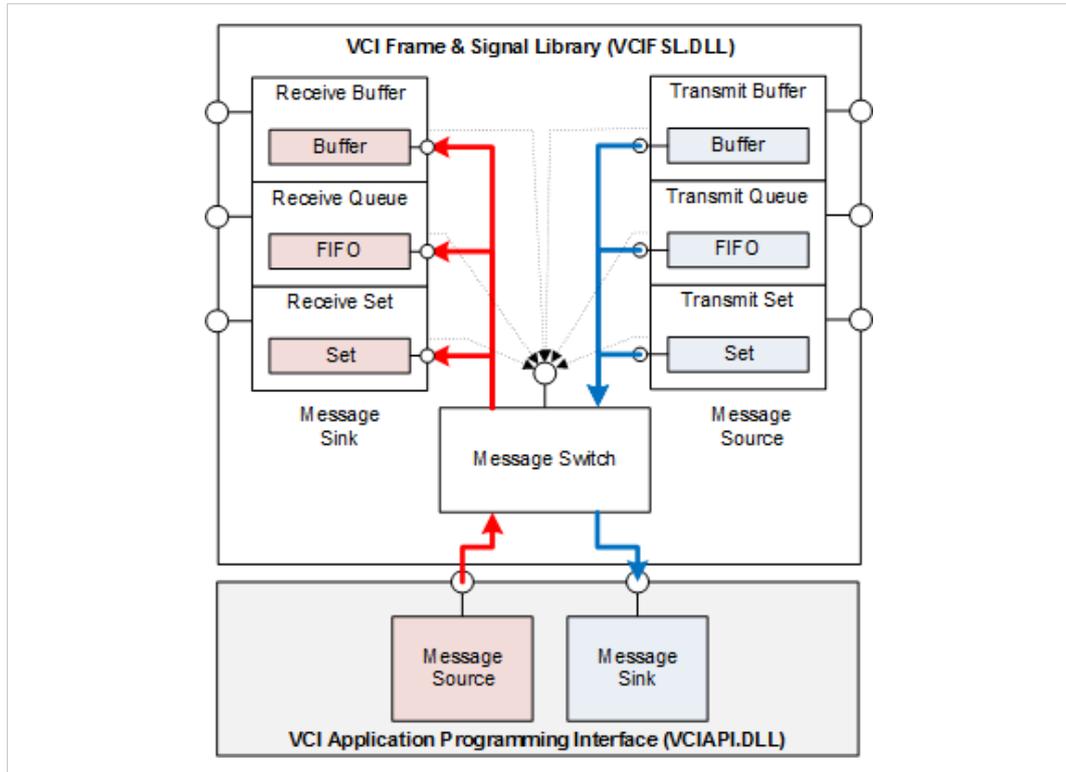


Fig. 3 Message based clients

The message sink client receives messages from a VCI internal source via the message switch. The received messages are buffered and provided via a destination specific interface to the application. The message source client buffers the messages received from the application and transmits the message via the message switch to a VCI internal sink. The message source client can transmit the messages directly, cyclically or delayed.

To transmit and receive messages three kinds of buffers can be used:

- simple buffer:
  - only the last received or written message is buffered and can be read
  - new message overwrites a not yet read message
- FIFO:
  - buffers in chronological sequence, no message is lost
  - must be read regularly to avoid overrun
- message set:
  - pools several messages
  - each message is allocated either to a simple buffer or to a FIFO

For a detailed description of the buffer types see [CAN Specific Components, p. 10](#).

## 2.2.2 Signal Based Clients

Signal based clients provide interfaces for applications to transmit and to receive signals and process values. Signal based clients act in the same way as message based clients and therefore they decouple the application data from the bus specific message packets. Signals are organized in signal sets. Signal sets can be accessed with the functions of interface *ISignalSet* or via the functions of the derived interfaces *IRSignalSet* or *ITSignalSet*.

The sink is a client that receives signals packed in messages from a VCI internal source via the message switch and extracts the contained signal. The source is a client that packs signals into messages and transmits the messages to a VCI internal sink via the message switch.

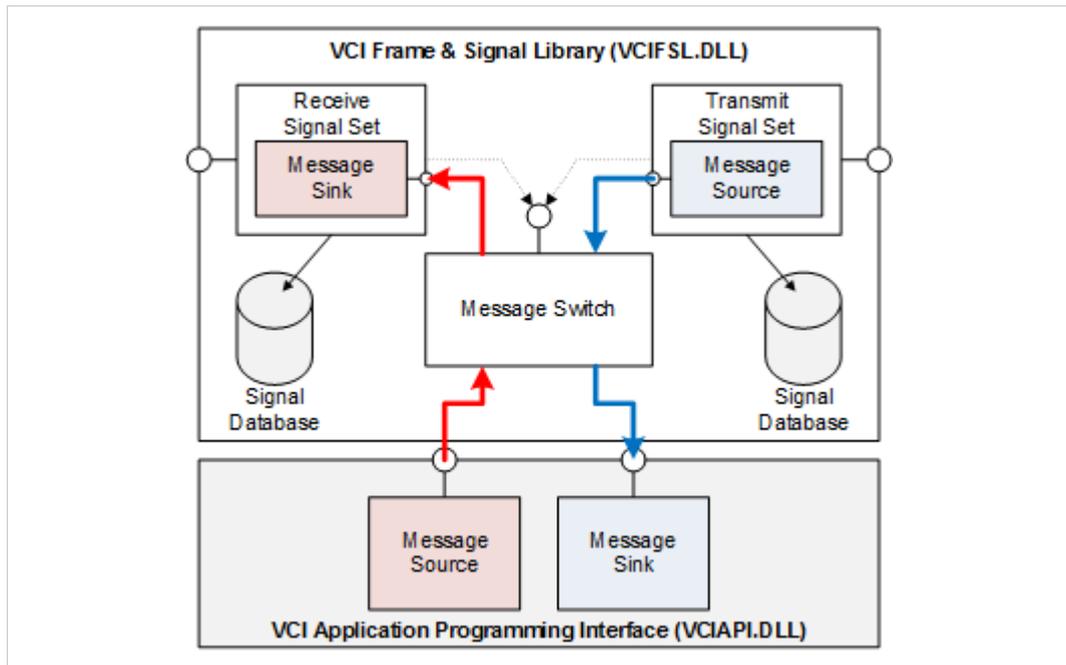


Fig. 4 Signal based clients

To transmit and receive signal sets two kinds of buffer can be used. Each signal is assigned to one kind of buffer.

The following kinds of buffer can be used:

- simple buffer:
  - only the last received or written signal is buffered and can be read
  - a new signal overwrites a not yet read signal
- FIFO:
  - buffers in chronological sequence, no signal is lost
  - must be read regularly to avoid overrun

The signals of a signal set are assigned to messages based on a description in a data base file.

Signals can be mapped in two ways:

- message based assignment: assigning individual signals to one or more messages
- process specific assignment: pooling of individual signals or process values in groups, so-called process data units (PDUs) and assigning of the PDUs to different messages (for example if, process-related, various signals must be transmitted together)

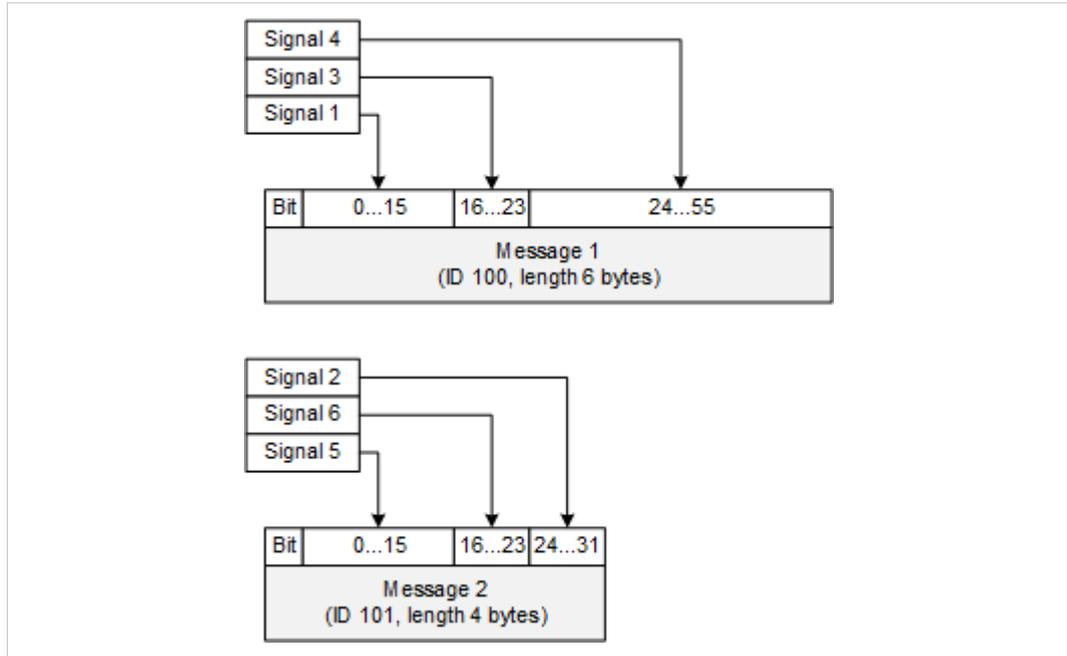


Fig. 5 Message based assignment

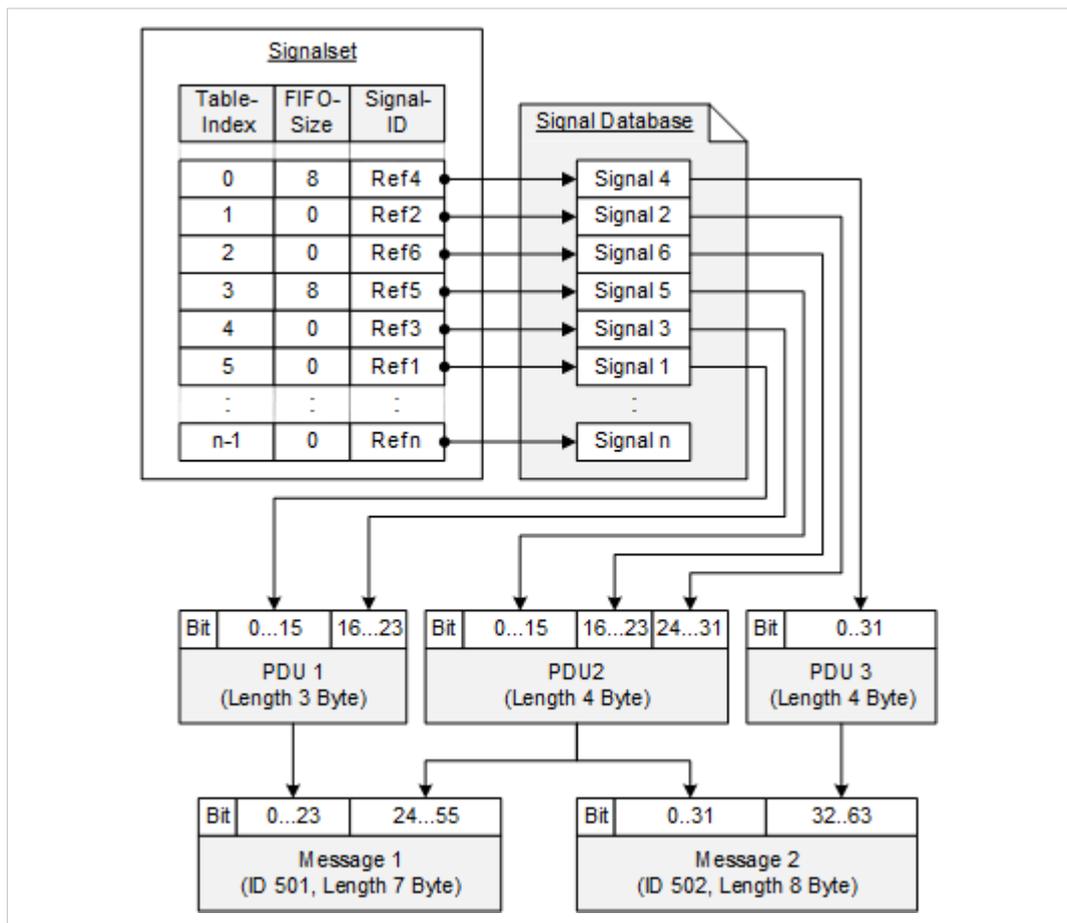


Fig. 6 Process specific assignment

The receive signal set (sink) receives signals packed in messages from a VCI internal source via the message switch. Based on the description in a data base file the client extracts the signals from the message and writes the extracted values and the receive time in the buffer. The signals can then be read with the function Read.

The transmit signal set (source) packs signals into messages and transmits the messages to a VCI internal sink via the message switch. The client reads the message from the buffer and packs, based on the description in a data base file, the signals into a message and transmits the message packets via the message switch. The signals can be transmitted with the function `Write`.

The rules how the signals (process data) are mapped to the messages are described in a data base file. The data base file can be generated with the `Ixxat DIM Editor` (contained in the VCI installation). A possible format is the FIBEX format (field bus exchange format), the standard of the "Association for Standardization of Automation and Measuring Systems" (ASAM) which is used mainly for the description of controller networks in the automotive industry (see MCD-2 NET on [www.asam.net](http://www.asam.net)). Another proprietary format is CANdb from Vector.

### 2.2.3 CAN Specific Components

CAN specific components are connected to the bus adapter via a CAN channel that is defined by the application. The messages received from the bus adapter are written into the receive FIFO of the CAN channel. The distributor transmits the messages via the message switch to the connected message sink clients. Messages that are provided by a message source client are transmitted by the distributor to the Transmit FIFO of the channel. The CAN specific message switch communicates with the VCI via a CAN channel with extended functionality (`ICanChannel2`). CAN channels are not prioritized. For the message switch the same conditions as for all other VCI specific applications apply. The message switch can be used as exclusive switch or as non-exclusive switch. For more information about CAN channels see *VCI: C++ Software Design Guide*.

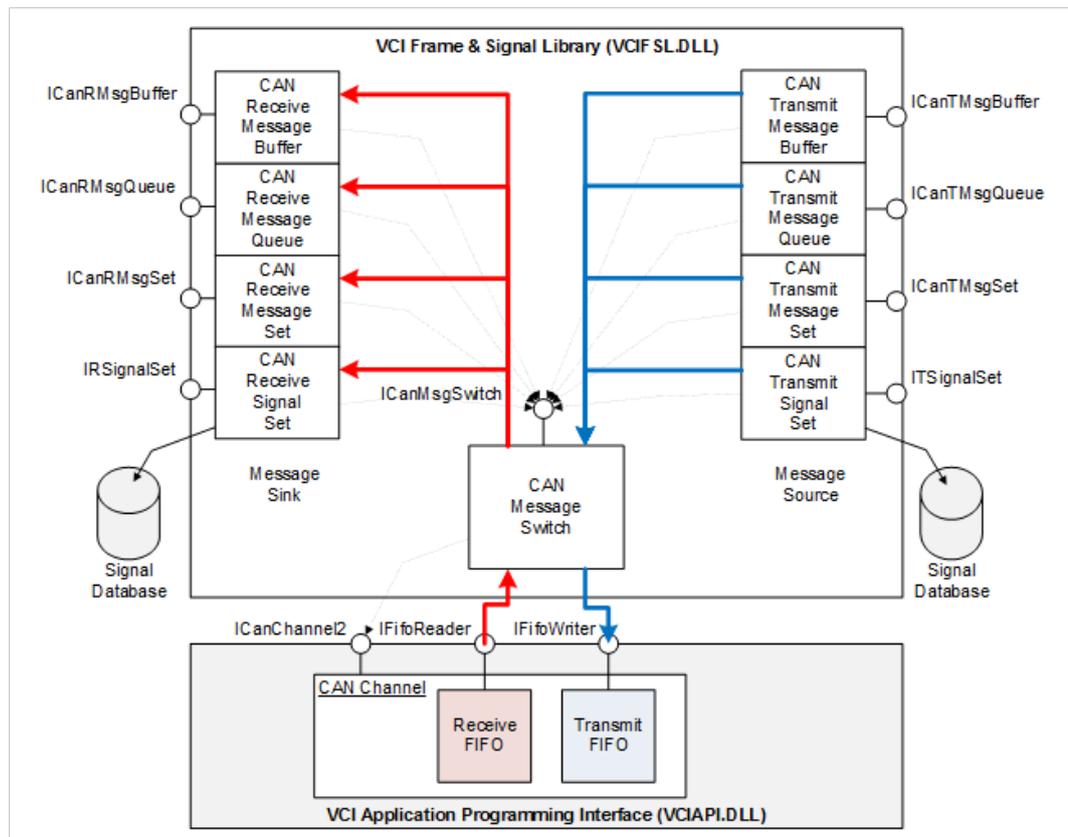


Fig. 7 CAN message sink and CAN message source

## Message Sink

The message sink is a client that receives CAN messages from the CAN channel via the message switch.

CAN specific message switches support the following types of message sinks:

- receive message buffer (`IID_ICanRMsgBuffer`):
  - simple buffer that buffers a message with the CAN ID defined by application and counts the number of received messages
  - only the last received or written message is buffered and can be read
  - new message overwrites a not yet read message
  - by calling `Read` the counter is set to 0 and therefore the counter shows the number of overwritten messages
- receive message queue (`IID_ICanRMsgQueue`):
  - FIFO with set size
  - buffers messages with the CAN ID defined by application in chronological sequence, no message is lost
  - must be read regularly to avoid overrun
  - several messages can be read simultaneously with `Read`
- receive message set (`IID_ICanRMsgSet`):
  - combination of various message buffers with different IDs (simple buffers or FIFOs)
  - several messages can be read simultaneously with `Read`
- receive signal set (`IID_ICanRSignalSet`):
  - combination of various signals
  - signal values are extracted from the receive messages with use of the description in the data base

### Message Source

The message source is a client that transmits CAN messages to the CAN channel via the message switch.

The message switch checks all sources periodically and event-driven, dependent on the set clock frequency and the number of connected clients.

CAN specific message switches support the following types of message sources:

- transmit message buffer (IID\_ICanTMsgBuffer):
  - simple buffer that buffers one message from the application
  - with `Write` messages can be transmitted directly, delayed or cyclically
- transmit message queue (IID\_ICanTMsgQueue):
  - FIFO with set size
  - buffers messages from the application in chronological sequence, no message is lost
  - transmits the messages in the received order directly, delayed or cyclically
  - several messages can be transmitted simultaneously with `Write`
- transmit message set (IID\_ICanTMsgSet):
  - combination of various message buffers with different IDs (simple buffers or FIFOs)
  - several messages can be transmitted simultaneously with `Write`
- transmit signal set (IID\_ICanTSignalSet):
  - combination of various signals, transmission after a value change or timer-driven
  - signal values are packed to messages with use of the description in the data base

For more information see [CAN Specific Communication, p. 17](#).

## 3 Communication

### 3.1 Signal Based Communication

Signals are organized in signal sets. For more information about signal based clients see [Signal Based Clients, p. 8](#). For information how to create signal based clients see [Creating a Receive Signal Set, p. 22](#) and [Creating a Transmit Signal Set, p. 26](#).

#### 3.1.1 Accessing and Initializing the Signal Set

- ▶ Access the signal set with functions of the interface [ISignalSet](#) or via the functions of the derived interfaces [IRSignalSet](#) or [ITSignalSet](#).
- ▶ Initialize the signal set (sink and source) with function [LoadDB](#):
  - In parameter *pszFile* determine the absolute or relative file path including the name of the data base file as 0-terminated character string.
  - In parameter *pszPara* further data base specific parameter can be determined as value pair *keyword=value* (case sensitive). Separate value pairs with semicolon. Currently defined keywords are *cluster*, *channel* and *invalsignals* . For more information see [LoadDB](#) parameter *pszPara*.
  - In parameter *pszSigs* specify the signals that are received or transmitted from the signal set. Select only signals that refer to the selected network and if determined, the selected channel (in *pszPara*). Signals that refer to other networks or channels are ignored. For more information see [LoadDB](#) parameter *pszSigs* and [Signal Description, p. 13](#).
  - In parameter *awDepth* specify the type of buffer to be used for each signal. Value 0 or 1 defines a simple buffer. Value higher than 1 defines a FIFO. For more information see [LoadDB](#) parameter *awDepth* and [Buffer Type, p. 14](#).
  - In parameter *dwCount* define the capacity of the arrays resp. the number of elements in the arrays *awDepth* and *shSigId*. The value must match the number of character strings that are defined in *pszSigs* (without the empty terminating string, see [LoadDB](#) parameter *dwCount*).

If run successfully:

- Parameter *ahSigId* returns a pointer to the array that contains the handles and reference IDs of signal buffers.
- ▶ To request or change more features call [GetAttr](#) or [SetAttr](#).
- ▶ Activate the signal set with function [Enable](#).
  - Connection to message switch is established.
  - Messages and the contained signals can be received and transmitted.

#### Signal Description

Signals are described with the name of the message package or of the PDU that contains the signal in combination with the frame or PDU specific name of the signal. The description with the data base internal unique ID is also possible.

#### Via Frame

```
"FrameShortName/SignalShortName" or  
"$frm/FrameShortName/SignalShortName"
```

**Via PDU**

```
"$pdu/PDUshortName/SignalShortName"
```

**Via Data Base Internal ID**

```
"$id/SignalID"
```

**Example**

The example code shows the description of the signals 1–6 in [Fig. 6 Process specific assignment, p. 9](#) in the according tabular sequence.

```
static TCHAR szSignals[] =
    TEXT("$pdu/PDU3/Signal4\0") // Index 0
    TEXT("$pdu/PDU2/Signal2\0") // Index 1
    TEXT("$pdu/PDU2/Signal6\0") // Index 2
    TEXT("$pdu/PDU2/Signal5\0") // Index 3
    TEXT("$pdu/PDU1/Signal3\0") // Index 4
    TEXT("$pdu/PDU1/Signal1\0") // Index 5
    TEXT("\0"); // end of table
```

With a FIBEX data base it is possible to use the same names for different signals if the signals are contained in different messages or PDUs. For example the signal *S1* can be contained in the message *MSG1* and a second signal that is contained in the message *MSG2* can also be named *S1*. Therefore a signal must always be identified by the entire description, not only by the signal name.

The example shows the signal name *Air*, that is used in three different measured values of a weather station.

```
static TCHAR szSignals[] =
    TEXT("$frm/Pressure/Air\0") // air pressure
    TEXT("$frm/Temperature/Air\0") // air temperature
    TEXT("$pdu/Humidity_pdu/Air\0") // humidity
    TEXT("\0");
```

**Buffer Type**

To create a simple buffer for an individual signal that is defined in *pszSigs*, define value 0 or 1 in the array *awDepth*. To create a FIFO for an individual signal define a value higher than 1. The value in *awDepth[0]* defines the size of the buffer for the first signal defined in *pszSigs*, the value in *awDepth[1]* defines the size of the buffer for the second signal defined in *pszSigs*, etc.

```
static TCHAR szSignals[] = // buffer size | signal handle
    //-----+-----
    TEXT("$pdu/PDU3/Signal4\0") // awDepth[0] | ahSigId[0]
    TEXT("$pdu/PDU2/Signal2\0") // awDepth[1] | ahSigId[1]
    TEXT("$pdu/PDU2/Signal6\0") // awDepth[2] | ahSigId[2]
    TEXT("$pdu/PDU2/Signal5\0") // awDepth[3] | ahSigId[3]
    TEXT("$pdu/PDU1/Signal3\0") // awDepth[4] | ahSigId[4]
    TEXT("$pdu/PDU1/Signal1\0") // awDepth[5] | ahSigId[5]
    //-----+-----
    TEXT("\0"); // end of table
```

It is possible to create the same kind of buffer for all signals that are defined in *pszSigs*. Value `NULL` in *awDepth* creates a simple buffer for each signal. To create a FIFO of the same size for each signal, define a pointer value smaller 65536 in *awDepth*. The pointer value then defines the buffer capacity of the FIFO.

### 3.1.2 Converting Signal Values

Normally signal values are physical values. To be able to transfer the signals in messages and to display and visualize the signals in applications, the values must be converted before and after transmission. The rules how the signals are converted are set in the signal data base (FIBEX or CANdb format). For the message transfer raw values are necessary and to display mostly physical values are used.

- ▶ To convert physical values to raw values and vice versa call function [Convert](#).
  - In parameter *dwMode* define the converting mode.
  - In parameter *aInSig* define a pointer to the array that contains the values to be converted.
  - In parameter *dwCount* define the number of elements in both arrays.
  - Parameter *aOutSig* returns a pointer to the array that contains the converted values.
- ▶ For more information see parameter description in [Convert](#), p. 35.

### 3.1.3 Reading Receive Signal Sets

To access a receive signal set from the application the interface [IRSignalSet](#) is used.

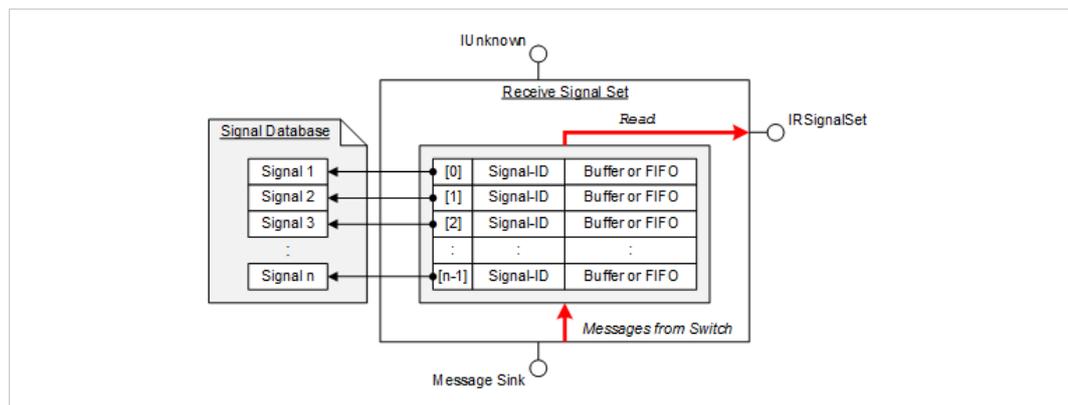


Fig. 8 Receive Signal Set

- ▶ Make sure that field *hSigId* of each element is initialized with the reference ID of the signal to be read.
- ▶ To read the last received signal values from the internal buffer call function [Read](#).
  - In parameter *fConvert* define if raw values are converted in physical values (TRUE) or if raw signals are delivered (FALSE).
  - In parameter *dwCount* define the number of receive buffers to be read.

If run successfully:

- Parameter *aSignal* points to the array where the read values are stored as elements of type [FSL SIGNAL](#). Field *qwTime* of each element contains the receive time. Field *sValue* of each element contains the received signal value of the respective element.
- Parameter *adwRxCnt* contains the number of each received signal value since the last call of [Read](#). If no signal was received the respective element is set to 0.
- If an overrun and therefore data loss occurs in one of the FIFOs of the signal set or in one of the upstream FIFOs, the receive counter of the respective signals is higher 1 and the bit `FSL_SIG_STAT_RXOVR` in field *dwStat* of structure [FSL SIGNAL](#) is set.

### 3.1.4 Writing Transmit Signal Sets

To access a transmit signal set from the application the interface *ITSignalSet* is used.

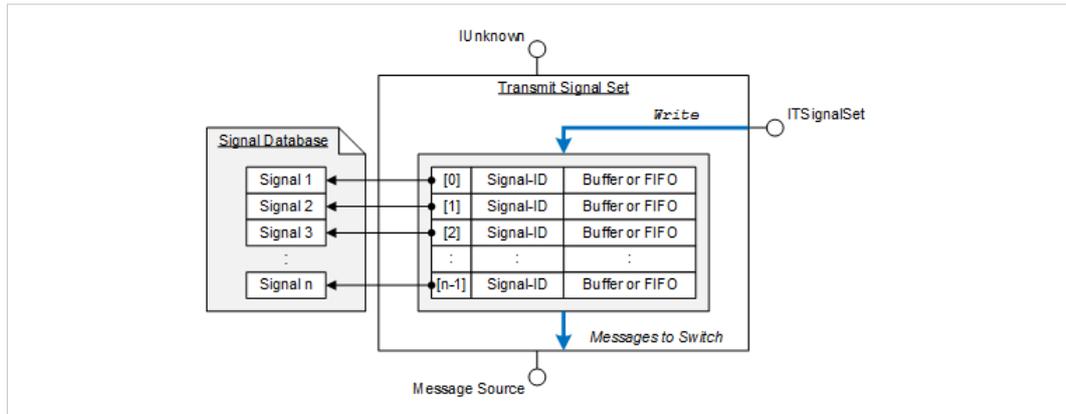


Fig. 9 Transmit signal set

- ▶ Make sure that field *hSigId* of each element in the array *aSignal* is initialized with the reference ID of the signal to be written.
- ▶ To write signal values call function *Write*.
- ▶ For each buffer only one signal value can be written. To write several values a signal buffer with a FIFO, call the function for each value.
  - In parameter *fConvert* define if physical values are converted in raw values (`TRUE`) or if the values to be transmitted are raw values (`FALSE`).
  - With parameter *afValid* define which signal value is valid. If *afValid[x]* is `TRUE`, the respective signal value is adopted in *aSignal[x]*, if *afValid[x]* is `FALSE` the signal value in *aSignal[x]* is ignored.
  - If a FIFO is used, check if a signal value is adopted or not (if the FIFO is full) in parameter *afDone* (`FALSE`: not transferred or invalid value).
  - In parameter *dwCount* define the number of elements in the arrays *aSignal*, *afValid*, and *afDone*.

If run successfully:

- Signal values in the array the parameter *aSignal* is pointing to are transmitted in the respective internal buffers.

### 3.1.5 Deactivating and Releasing the Signal Set

- ▶ To deactivate the signal set call function *Disable*.
  - Connection to message switch is interrupted.



Connection can be reestablished with function *Enable*.

- ▶ To deactivate and close the signal set call function *CloseDB*.
  - Signal set and the pointer to the interface are released. Pointer cannot be used anymore.

## 3.2 CAN Specific Communication

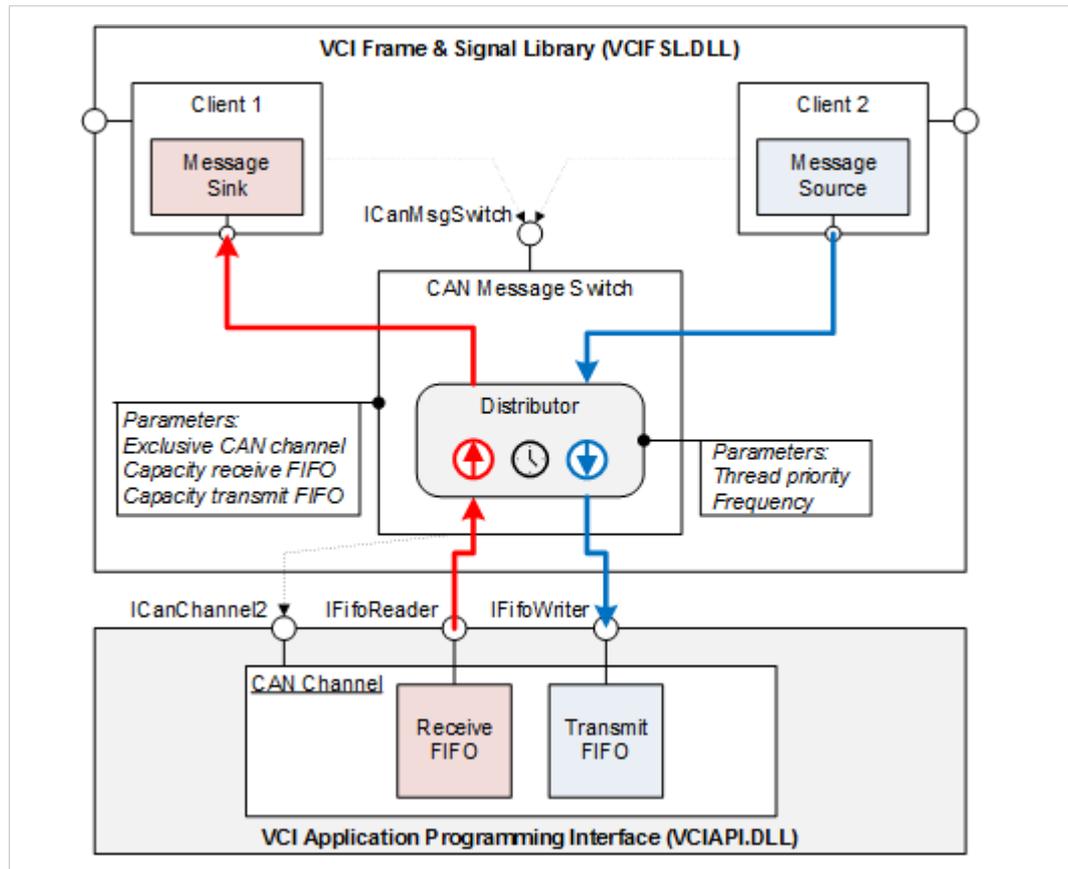


Fig. 10 Components of CAN message switch

### 3.2.1 Creating a Message Switch

- ▶ To create a message switch call function `VciCreateCanMsgSwitch`.
  - In parameter `pBalObj` determine for which adapter the switch is created.
  - In parameter `dwBusNo` determine the CAN connection to be used.

If run successfully:

- Variable that points to parameter `ppSwitch` returns a pointer to the interface [ICanMsgSwitch](#) of the message switch.

### 3.2.2 Initializing and Activating the Message Switch

- ▶ Initialize the message switch with function *Initialize*.
  - In parameter *dwTiming* determine the frequency of the distributor (how often the distributor checks the clients for messages). See *Temporal Accuracy, p. 18*.
  - In parameter *IPriority* determine the thread priority of the distributor (for more information see parameter *IPriority* in *Initialize*).
  - In parameter *fExclusive* determine if the CAN connection is used exclusively for the message switch to be opened (`TRUE`) or if further switches or channels can be created for the connection (`FALSE`) and the connection can be used by further applications.
  - To create a receive FIFO and set the capacity, determine in parameter *wRxFifoSize* the capacity of the receive FIFO in CAN messages of structure `CANMSG2`.
  - To create a transmit FIFO and set the capacity, determine in parameter *wTxFifoSize* the capacity of the receive FIFO in CAN messages of structure `CANMSG2`.



*HMS recommends a capacity between 32 and 128.*

---

→ Newly created message switch is initialized but inactive and not connected to the bus.

- ▶ Create clients for the message switch (see *Creating and Initializing Clients: Message Sinks, p. 19* and *Creating and Initializing Clients: Message Sources, p. 22*).

A new message switch is inactive and not connected to the bus. Messages are only received and transmitted if the switch is active.

- ▶ Make sure, that the CAN controller is in state *online*.
- ▶ Activate the message switch with function *Activate*.

If the switch is initialized and activated:

- Distributor checks the clients for messages according to the defined frequency.
- Messages received from the receive FIFO are transmitted event driven to clients with active message sink.
- If a free entry in the transmit FIFO is available, the distributor checks the clients for messages to be transmitted.

#### Temporal Accuracy

How accurate the clients are served is dependent on the defined frequency and on the number of active clients. The more clients have to be served, the longer is the cycle. The number of events on the CAN channel also influence the accuracy. The more messages are transferred, the more receive messages must be transmitted from the distributor to the active message sinks. A higher load leads to a delay in the transmission of messages and therefore to a delay in the checking of the message sources.

### 3.2.3 Creating and Initializing Clients: Message Sinks

After initializing the message switch any number of clients can be created for the message switch. Clients can be sinks, that receive CAN messages of a CAN channel via the message switch. CAN specific message switches support different types of sinks: simple buffer, message queues, message sets and signal sets. For more information see [CAN Specific Components, p. 10](#).

#### Creating a Receive Message Buffer

For communication the interface `ICanRMsgBuffer` is used.

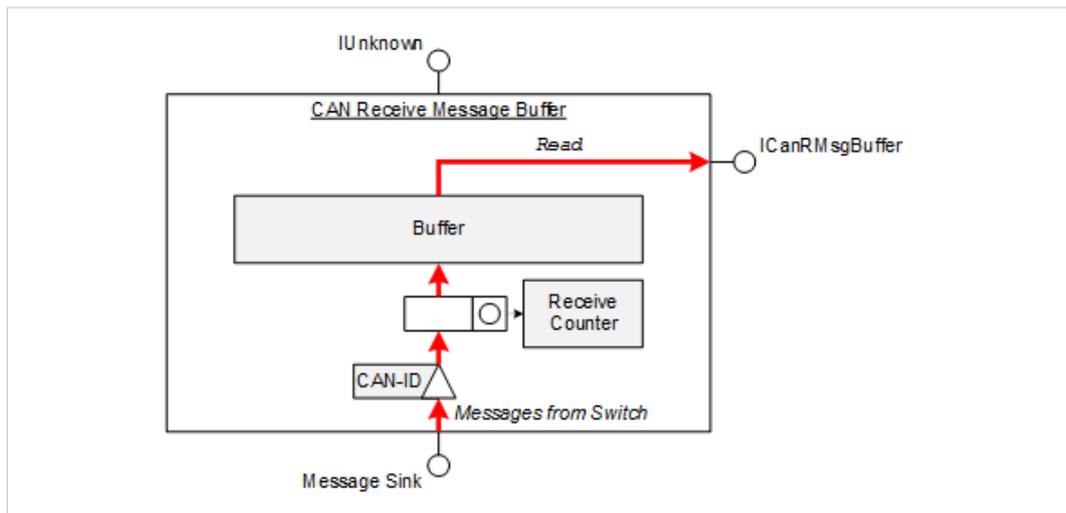


Fig. 11 CAN receive message buffer

- ▶ To create a receive message buffer, call function `CreateClient` with value `IID_ICanRMsgBuffer` in parameter `riid`.
    - Parameter `ppv` returns a pointer to the interface `ICanRMsgBuffer`.
    - Newly created clients are not connected to the distributor of the message switch and cannot receive messages.
  - ▶ To connect the client to the distributor, initialize the client with function `ICanRMsgBuffer::Enable`.
    - In parameter `dwCanId` determine the CAN ID of the message to be filtered from the received message stream.
- If run successfully:
- Message buffer is connected to the distributor of the message switch and receives messages.
- ▶ Make sure that the message switch is initialized and activated (see [Initializing and Activating the Message Switch, p. 18](#)).
  - ▶ To read the messages from the receive buffer, call function `Read`.
    - Parameter `pCanMsg` points to the variable that stores the content of the buffer.
    - Parameter `pdwRxCnt` points to the variable that stores the current value of the receive counter.
    - Receive counter is set to 0 with each call of `Read` and therefore shows the number of overwritten messages since the last call.
    - If the overrun bit is set in a message, an overrun occurred in one of the upstream FIFOs.

### Creating a Receive Message Queue

For communication the interface `ICanRMsgQueue` is used.

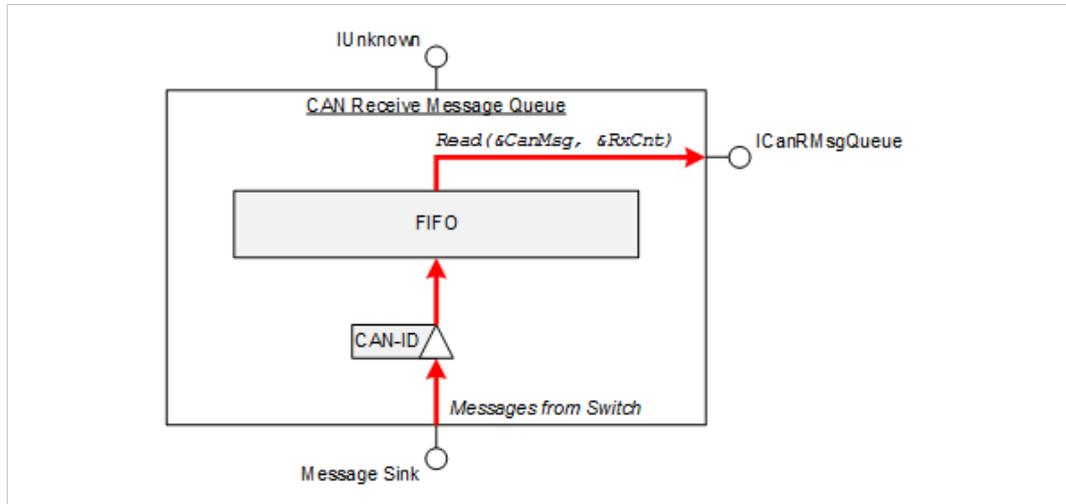


Fig. 12 CAN receive message queue

- ▶ To create a receive message queue, call function `CreateClient` with value `IID_ICanRMsgQueue` in parameter `riid`.
  - Parameter `ppv` returns a pointer to the interface `ICanRMsgQueue`.
  - Newly created clients are not connected to the distributor of the message switch and cannot receive messages.
- ▶ To connect the client to the distributor, initialize the client with function `ICanRMsgQueue::Enable`.
  - In parameter `dwCanId` determine the CAN ID of the message to be filtered from the received message stream.
  - In parameter `wDepth` determine the size of the FIFO in number of CAN messages.
 If run successfully:
  - Message queue is connected to the distributor of the message switch and receives messages.
- ▶ Make sure that the message switch is initialized and activated (see [Initializing and Activating the Message Switch, p. 18](#)).
- ▶ To read the messages from the receive buffer, call function `Read`.
  - In parameter `dwCount` determine the number of CAN messages to be read and stored.
 If run successfully:
  - Parameter `aCanMsg` points to the variable that stores the content of the buffer.
  - Parameter `pdwDone` points to the variable that stores the number of actually read messages.
  - If the overrun bit is set in a message, an overrun occurred in the queue or in one of the upstream FIFOs and data is lost.

### Creating a Receive Message Set

For communication the interface `ICanRMsgSet` is used.

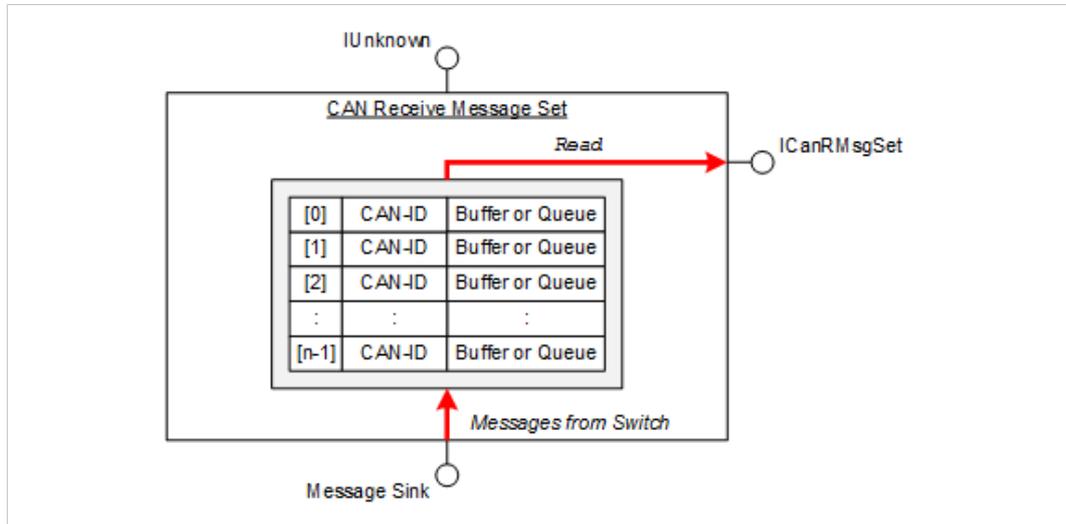


Fig. 13 CAN receive message set

- ▶ To create a receive message set, call function `CreateClient` with value `IID_ICanRMsgSet` in parameter `riid`.
  - Parameter `ppv` returns a pointer to the interface `ICanRMsgSet`.
  - Newly created clients are not connected to the distributor of the message switch and cannot receive messages.
- ▶ To connect the client to the distributor, initialize the client with function `ICanRMsgSet::Enable`.
  - In the array the parameter `adwCanId` points to, determine the CAN IDs of the messages to be filtered from the received message stream.
  - In the element the parameter `awDepth` point to, determine the capacity of the respective buffer (for more information see parameter description in `Enable`).
  - In parameter `dwCount` determine the number of elements in both arrays `dwCanId` and `wDepth`.

If run successfully:

  - Message set is connected to the distributor of the message switch and receives messages.
- ▶ Make sure that the message switch is initialized and activated (see [Initializing and Activating the Message Switch, p. 18](#)).
- ▶ To read the messages from the receive buffer, call function `Read`.
  - In parameter `dwFirst` determine the 0 based start index and in parameter `dwCount` the number of buffers to be read. Value must be equal or smaller than the number of elements in the arrays `aCanMsg` or `adwRxCnt`.

If run successfully:

  - Parameter `aCanMsg` points to the variable that stores the last received messages from the buffers.

- Parameter *adwRxCnt* points to the array that stores the number of received messages in the respective buffer since the last call (with a queue the value is maximally 1, if higher an overrun occurred).
- If the overrun bit is set in a message, an overrun occurred in the queue or in one of the upstream FIFOs and data is lost.

### Creating a Receive Signal Set

For communication the interface *IRSignalSet* is used.

- ▶ To create a receive message set, call function *CreateClient* with value *IID\_IRSignalSet* in parameter *riid*.
  - Parameter *ppv* returns a pointer to the interface *IRSignalSet*.
  - Newly created clients are not connected to the distributor of the message switch and cannot transmit or receive messages.
- ▶ To connect the client to the distributor, initialize the client with function *LoadDB* and activate the signal set with function *Enable*. For more information see [Accessing and Initializing the Signal Set, p. 13](#).
- ▶ Make sure that the message switch is initialized and activated (see [Initializing and Activating the Message Switch, p. 18](#)).

## 3.2.4 Creating and Initializing Clients: Message Sources

After initializing the message switch any number of clients can be created for the message switch. Clients can be sources that transmit CAN messages to the message switch. CAN specific message switches support different types of sources: simple buffer, message queues, message sets and signal sets. For more information see [CAN Specific Components, p. 10](#).

### Creating a Transmit Message Buffer

For communication the interface *ICanTMsgBuffer* is used.

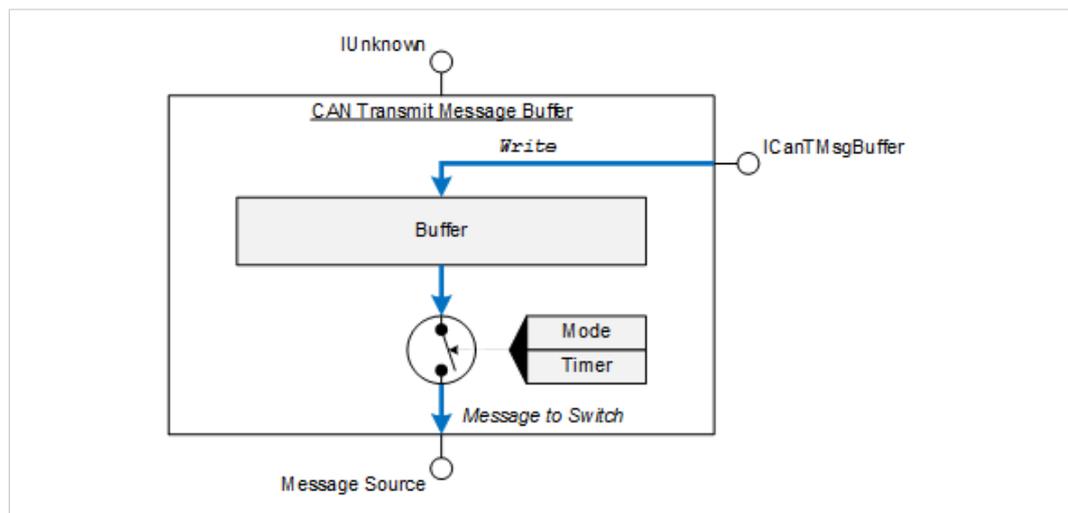


Fig. 14 CAN transmit message buffer

- ▶ To create a transmit message buffer, call function *CreateClient* with value *IID\_ICanTMsgBuffer* in parameter *riid*.
  - Parameter *ppv* returns a pointer to the interface *ICanTMsgBuffer*.
  - Newly created clients are not connected to the distributor of the message switch and cannot transmit messages.

- ▶ To connect the client to the distributor, initialize the client with function `ICanTMsgBuffer::Enable`.
  - In parameter `dwCanId` determine the CAN ID of the message to be accepted. Message must be of type `CAN_MSGTYPE_DATA` and must have a valid CAN ID.
  - In parameter `dwMode` determine the operation mode (direct, cyclically, delayed). For more information see [Cyclic and Delayed Transmission, p. 23](#).
  - In parameter `dwTime` determine the cycle or delay time.
- If run successfully:
  - Message buffer is connected to the distributor of the message switch and transmits messages.
- ▶ Make sure that the message switch is initialized and activated (see [Initializing and Activating the Message Switch, p. 18](#)).
- ▶ To transmit messages, call function `Write` with pointer of the message to be written in parameter `pCanMsg`.
  - Message is written in the internal buffer.
  - In operation mode `CAN_TX_DIRECT` message is transmitted once directly to the message switch.
  - In operation mode `CAN_TX_DELAYED` message is transmitted once to the message switch when the delay time (`dwTime`) is expired.
  - In operation mode `CAN_TX_CYCLIC` message is transmitted cyclically (cycle time in `dwTime`).

### Cyclic and Delayed Transmission

With cyclic transmission it is possible to change the content of the buffer by calling `Write` without changing the current cycle. The cyclic transmission is stopped when the client is deactivated with `Disable`. The cyclic transmission can be disabled by writing a CAN message with valid ID but with invalid value for field `uMsgInfo.bType` in the transmit buffer, e.g. 255 or 0xFF. Messages with invalid value are not of type `CAN_MSG_DATA` and are therefore ignored.

In cyclic transmission the first message is transmitted without delay, the next message is transmitted after the time defined in `dwTime`. The cycle timer value is calculated by the difference of current time ( $T_C$ ) and time of the last transmitted message ( $T_P$ ). The message is transmitted when  $(T_C - T_P) \geq dwTime$ .

The delay timer value is calculated by the difference of current time ( $T_C$ ) and calling time of `Write` ( $T_W$ ). The message is transmitted when  $(T_C - T_W) \geq dwTime$ .

The calculation is done by the distributor during the query of the message source and therefore the accuracy is dependent on the frequency of the query. The defined frequency of the distributor and the number of active clients are influencing factors. Another factor is the bus load: the higher the bus load, the more inaccurate is the transmit time. Furthermore the query interval is influenced by the performance of the computer and the priority of process and thread of the message distributor. With a thread with normal or low priority the query interval is smaller than with a thread with higher priority. To increase the query interval and reaction time of the distributor, use a higher prioritized process and thread. The exact calculation of the processing time is not possible, and therefore should be determined experimentally.

### Creating a Transmit Message Queue

For communication the interface `ICanTMsgQueue` is used.

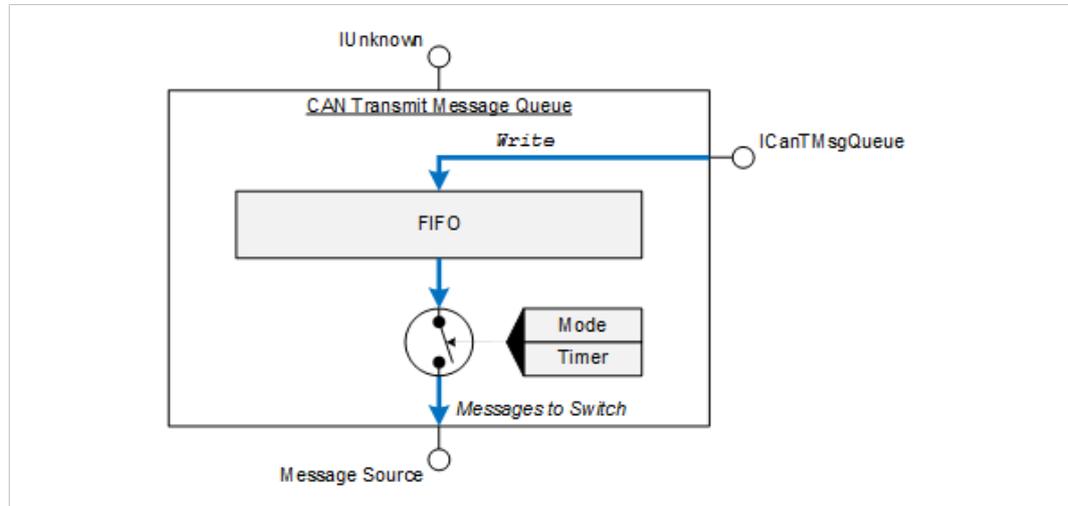


Fig. 15 CAN transmit message queue

- ▶ To create a receive message queue, call function `CreateClient` with value `IID_ICanTMsgQueue` in parameter `riid`.
  - Parameter `ppv` returns a pointer to the interface `ICanTMsgQueue`.
  - Newly created clients are not connected to the distributor of the message switch and cannot transmit messages.
- ▶ To connect the client to the distributor, initialize the client with function `ICanTMsgQueue::Enable`.
  - In parameter `dwCanId` determine the CAN ID of the message to be accepted. Message must be of type `CAN_MSGTYPE_DATA` and must have a valid CAN ID.
  - In parameter `wDepth` determine the size of the FIFO in number of CAN messages.
  - In parameter `dwMode` determine the operation mode (direct, cyclically, delayed). For more information see [Cyclic and Delayed Transmission, p. 23](#).
  - In parameter `dwTime` determine the cycle or delay time.

If run successfully:

  - Message queue is connected to the distributor of the message switch and transmits messages.
- ▶ Make sure that the message switch is initialized and activated (see [Initializing and Activating the Message Switch, p. 18](#)).
- ▶ To transmit messages, call function `Write`.
  - In parameter `aCanMsg` define pointer to array with the CAN messages to be written.
  - In parameter `dwCount` determine the number of CAN messages to be transmitted.

If run successfully:

- Messages of the array are written in the internal FIFO.
- Parameter *pdwDone* points to the variable that stores the number of actually written messages.
- In operation mode `CAN_TX_DIRECT` message is transmitted once directly to the message switch.
- In operation mode `CAN_TX_DELAYED` message is transmitted once to the message switch when the delay time (*dwTime*) is expired. If the FIFO contains further messages the timer is started again for the next message.
- In operation mode `CAN_TX_CYCLIC` message is transmitted cyclically (cycle time in *dwTime*).

### Creating a Transmit Message Set

For communication the interface `ICanTMsgSet` is used.

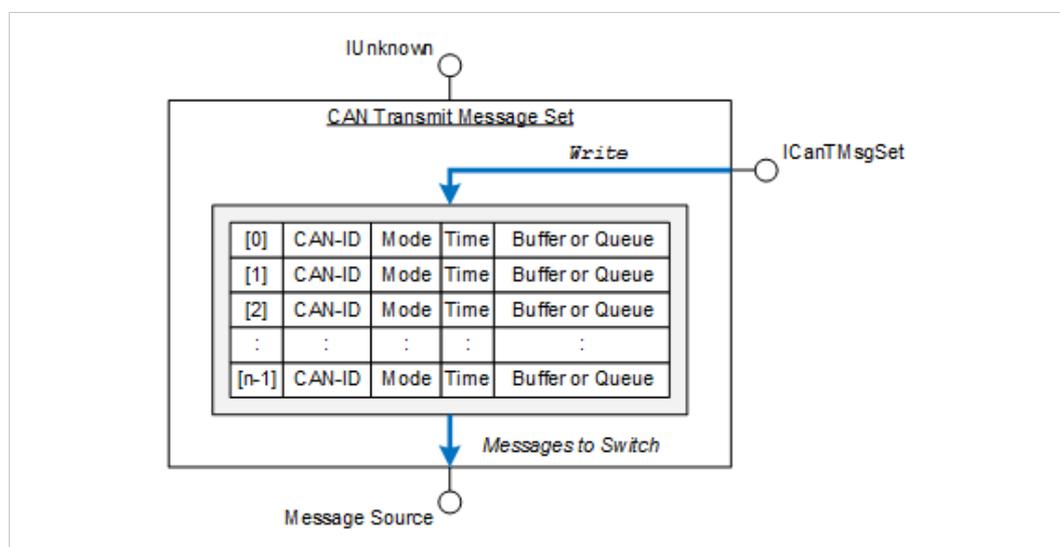


Fig. 16 CAN transmit message set

- ▶ To create a receive message set, call function `CreateClient` with value `IID_ICanTMsgSet` in parameter *riid*.
  - Parameter *ppv* returns a pointer to the interface `ICanTMsgSet`.
  - Newly created clients are not connected to the distributor of the message switch and cannot transmit messages.
- ▶ To connect the client to the distributor, initialize the client with function `ICanTMsgSet::Enable`.
  - In parameter *adwCanId* determine the CAN ID of the message to be accepted by the different buffers. Message must be of type `CAN_MSGTYPE_DATA` and must have a valid CAN ID.
  - In the element the parameter *awDepth* point to, determine the capacity of the respective buffer (for more information see parameter description in `Enable`).
  - In parameter *adwMode* determine the operation mode (direct, cyclically, delayed) of the respective buffer. For more information see `Cyclic and Delayed Transmission`, p. 23.
  - In parameter *adwTime* determine the cycle or delay time.

- In parameter *dwCount* determine the number of elements in the arrays *adwCanId*, *awDepth*, *adwMode*, and *adwTime*.

If run successfully:

- Message set is connected to the distributor of the message switch and transmits messages.
- ▶ Make sure that the message switch is initialized and activated (see [Initializing and Activating the Message Switch, p. 18](#)).
- ▶ To transmit messages, call function *Write*.
- ▶ For each buffer only one message can be transmitted. To write several messages to a queue, call the function for each message to be transmitted.
  - In parameter *aCanMsg* define pointer to array with the CAN messages to be written.
  - With parameter *afValid* define which message is valid. If *afValid[x]* is TRUE, the respective message is adopted in *aCanMsg[x]*, if *afValid[x]* is FALSE the message in *aCanMsg[x]* is ignored.
  - If a FIFO is used, check if a message is adopted or not (if the FIFO is full) in parameter *afDone* (FALSE: not transferred or invalid value).
  - In parameter *dwFirst* determine the 0 based start index and in parameter *dwCount* the number of messages to be written. Value must be equal or smaller than the number of elements in the array *aCanMsg*.

If run successfully:

- Messages of the array are written in the respective internal buffers.
- In operation mode `CAN_TX_DIRECT` message is transmitted once directly to the message switch.
- In operation mode `CAN_TX_DELAYED` message is transmitted once to the message switch when the delay time (*dwTime*) is expired. If the FIFO contains further messages the timer is started again for the next message.
- In operation mode `CAN_TX_CYCLIC` message is transmitted cyclically (cycle time in *dwTime*).

### Creating a Transmit Signal Set

For communication the interface `ITSignalSet` is used.

- ▶ To create a receive message set, call function *CreateClient* with value `IID_ITSignalSet` in parameter *riid*.
  - Parameter *ppv* returns a pointer to the interface `ITSignalSet`.
  - Newly created clients are not connected to the distributor of the message switch and cannot transmit messages.
- ▶ To connect the client to the distributor, initialize the client with function *LoadDB* and activate the signal set with function *Enable*. For more information see [Accessing and Initializing the Signal Set, p. 13](#).

### 3.2.5 Disconnecting Clients

- ▶ To deactivate and deregister a client, call function `Disable` of the respective interface. The client can be reconnected with `Enable`.

or

- ▶ To release the client, call function `Release`. The client is disconnected and the pointer is invalid and cannot be used anymore.
- ▶ To disconnect a client only temporarily (stays registered), call function `DetachClient`.
- ▶ To connect the client again, call function `AttachClient`.

A client is not exclusively assigned to the message switch it was created with, but can also be assigned to another message switch. A client can only be assigned to one message switch, not to several message switches simultaneously. To be able to reassign a client, the client must be connected to the message switch it was created with.

To reassign a client from message switch A to message switch B:

- ▶ Disconnect the client from message switch A: `DetachClient` at A.
- ▶ Connect the client to message switch B: `AttachClient` at B.



Functions `Enable` and `Disable` are not possible to use after reassigning of a client, because they are internally linked to the message switch the client was created with.

## 4 API Functions

### 4.1 Exported Functions

The declaration of the exported interfaces and functions are in the file *vcifsl.h*.

#### 4.1.1 VciCreateCanMsgSwitch

Creates a message switch for a CAN connection.

```
HRESULT VCI_API VciCreateCanMsgSwitch (
    IBalObject*    pBalObj
    UINT32         dwBusNo
    ICanMsgSwitch* ppSwitch);
```

##### Parameter

Parameter	Dir.	Description
<i>pBalObj</i>	[in]	Pointer to bus access layer (BAL) component of the CAN connection
<i>dwBusNo</i>	[in]	Number of bus connection to be opened. Value 0 selects bus connection 1, value 1 selects bus connection 2 etc. Entered value must match a CAN connection. See description of data structure <i>BALFEATURES</i> in <i>VCI: C++ Software Design Guide</i> .
<i>ppSwitch</i>	[out]	Address of a pointer variable. If run successfully the parameter saves the pointer to the interface <i>ICanMsgSwitch</i> of the newly created message switch. In case of an error the variable is set to <i>NULL</i> .

##### Return Value

Return value	Description
<i>VCI_OK</i>	Function succeeded
<i>!=VCI_OK</i>	Error, more information about error code provides the function <i>VciFormatError</i>

##### Remark

If run successfully the function increments the reference counter of the bus connection automatically by 1. When the application does not need the message switch anymore, the pointer returned in *ppSwitch* must be released with function [Release](#).

## 4.2 Interface IUnknown

All components provided by the VCI implement the interface `IUnknown` that is specified in the Component Object Model of Microsoft (MS-COM). The interface provides the function `QueryInterface` to request further interfaces of the component, and additionally the functions `AddRef` resp. `Release` to control the lifespan of the component.

### 4.2.1 QueryInterface

Calls a particular interface of a component.

```
ULONG QueryInterface ( REFIID riid, PVOID *ppv );
```

#### Parameter

Parameter	Dir.	Description
<i>riid</i>	[in]	Reference to the ID of the interface to access the component.
<i>ppv</i>	[out]	Address of a pointer variable. If run successfully the pointer is stored in the in <i>riid</i> requested interface. In case of an error the variable is set to NULL.

#### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

#### Remark

If run successfully the function increments the reference counter of the component automatically by 1. When the application does not need the interfaces resp. the components anymore, the pointer returned in *ppv* must be released with [Release](#).

### 4.2.2 AddRef

Increments the reference counter of the component by 1.

```
ULONG AddRef ( void );
```

#### Return Value

Function returns the current value of the reference counter.

#### Remark

The function always must be called, if the application stores a copy of the interface pointer. This ensures that the component exists as long as the last reference to it is released. An interface resp. the connected component is released by the call of the function [Release](#).

### 4.2.3 Release

Decrements the reference counter of the component by 1. If the reference count falls to 0, the component is released.

```
ULONG Release ( void );
```

#### Return Value

Function returns the current value of the reference counter.

#### Remark

After calling the function the pointer to the interface used by the application is not valid anymore and must not be used anymore. This also applies if the function returns a value larger than 0, i. e. the component itself is not released by this call.

## 4.3 Signal Specific Interfaces

### 4.3.1 ISignalSet

The interface defines the common functions to access the signal sets. The interface can only be opened in combination with one of the interfaces `IRSignalSet` or `ITSignalSet`.

#### LoadDB

Opens a signal data base (ANSI version or WideChar version) and initializes the signal set with the defined signals.

```
HRESULT LoadDBA (
    PCHAR pszFile,
    PCHAR pszPara,
    PCHAR pszSigs,
    UINT16 awDepth[],
    HANDLE ahSigId[],
    UINT32 dwCount );

HRESULT LoadDBW (
    PWCHAR pszFile,
    PWCHAR pszPara,
    PWCHAR pszSigs,
    UINT16 awDepth[],
    HANDLE ahSigId[],
    UINT32 dwCount );
```

#### Parameter

Parameter	Dir.	Description
<i>pszFile</i>	[in]	Pointer to 0-terminated character string that contains file name and optional path of the signal data base to be opened
<i>pszPara</i>	[in]	Pointer to 0-terminated character string with additional parameters, optional, can be <code>NULL</code> . Specify the data base specific parameters as value pair <i>keyword=value</i> (case sensitive). Separate value pairs with semicolon. The following keywords are currently defined: <i>cluster</i> : select network, as value enter the name of the cluster that is defined in the data base, the name is a short name of the cluster that is called SHORT-NAME in the FIBEX file, e.g. <i>cluster=CAN1</i> selects the network named CAN1 <i>channel</i> : select transmission channel, as value enter the name of the channel that is defined in the data base, defining a transmission channel is only necessary with FlexRay, with CAN and LIN <i>channel</i> is optional, because there is only one transmission channel <i>invalsignals</i> : define if invalid signal values are accepted or if not, <i>invalsignal=1</i> accepts invalid signals, <i>invalsignal=0</i> ignores invalid signals
<i>pszSigs</i>	[in]	Pointer to buffer that contains one or more 0-terminated character strings with the names of the signals to be loaded. Last entry in the buffer must be an empty character string (“\0”). Select only signals that refer to the selected network and if determined, the selected channel (in <i>pszPara</i> ). Signals that refer to other networks or channels are ignored.
<i>awDepth</i>	[in]	Array that defines the capacity of the signal buffer in number of entries of type <code>FSL SIGNAL</code> . To create a simple buffer for an individual signal that is defined in <i>pszSigs</i> , define value 0 or 1 in the array. To create a FIFO for an individual signal define a value higher than 1. The value in <i>awDepth[0]</i> defines the size of the buffer for the first signal defined in <i>pszSigs</i> , the value in <i>awDepth[1]</i> defines the size of the buffer for the second signal defined in <i>pszSigs</i> , etc. It is possible to create the same kind of buffer for all signals that are defined in <i>pszSigs</i> . Value <code>NULL</code> creates a simple buffer for each signal. To create a FIFO of the same size for each signal, define a pointer value smaller 65536. The pointer value then defines the buffer capacity of the FIFO.

Parameter	Dir.	Description
<i>ahSigId</i>	[out]	Pointer to an array for the reference IDs of the loaded signals. If a signal name that is defined in <i>pszSigs</i> is missing in the data base, the corresponding array element is set to NULL.
<i>dwCount</i>	[in]	Number of elements in the array <i>ahSigId</i> and if defined in the array <i>awDepth</i> . The value must match the number of character strings that are defined in <i>pszSig</i> (without the empty terminating string). If the value is smaller the function cannot generate buffer for all signals.

### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

### CloseDB

Deactivates the signal set and closes the currently opened signal data base.

```
HRESULT CloseDB ( void );
```

### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

### Remark

The function deregisters the signal set from the distributor of the message switch.

**GetAttr**

Retrieves the current value of a signal attribute (ANSI version or WideChar version).

```
HRESULT GetAttrA (
    HANDLE hSigId,
    UINT32 dwAttr,
    PVOID pvData,
    UINT32 dwSize,
    PUINT32 pdwOut );

HRESULT GetAttrW (
    HANDLE hSigId,
    UINT32 dwAttr,
    PVOID pvData,
    UINT32 dwSize,
    PUINT32 pdwOut );
```

**Parameter**

Parameter	Dir.	Description	
<i>hSigId</i>	[in]	Reference ID of the signal	
<i>dwAttr</i>	[in]	Typ of attribute to be retrieved, the following constants are possible:	
		FSL_SIG_ATTR_NAME	Name of the signal
		FSL_SIG_ATTR_UNIT	Unit of the signal value (not implemented)
		FSL_SIG_ATTR_DLID	ID of the default language (signal specific implementation is not supported)
FSL_SIG_ATTR_PLID	ID of the preferred language (signal specific implementation is not supported)		
<i>pvData</i>	[out]	Pointer to buffer area where the function stores the data of the requested attribute. If value NULL is defined, parameter <i>pdwOut</i> returns the necessary capacity of the buffer area in number of bytes.	
<i>dwSize</i>	[in]	Capacity of the buffer area in Byte to which the parameter <i>pvData</i> points. The size is only relevant, if in <i>pvData</i> a value unequal NULL is defined.	
<i>pdwOut</i>	[out]	If the function is run successfully, a pointer to a variable is returned that stores the number of data bytes that are copied to the buffer area to which the parameter <i>pvData</i> points.	

**Return Value**

Return value	Description
VCI_OK	Function succeeded
VCI_E_NOT_IMPLEMENTED	Attribute not supported
!=VCI_OK	Error, more information about error code provides the function VciFormatError

**Remark**

To determine what size is necessary for the data of the attribute, call the function with value NULL in *pvData*. The variable to which the parameter *pdwOut* points, returns the necessary capacity in number of bytes.

If value NULL is determined for *hSigId* and either FSL\_SIG\_ATTR\_DLID or FSL\_SIG\_ATTR\_PLID for *dwAttr* the function returns the current language ID of the data base.

**SetAttr**

Updates the value of a signal attribute (ANSI version or WideChar version).

```

HRESULT SetAttrA (
    HANDLE hSigId,
    UINT32 dwAttr,
    PVOID pvData,
    UINT32 dwSize );

HRESULT SetAttrW (
    HANDLE hSigId,
    UINT32 dwAttr,
    PVOID pvData,
    UINT32 dwSize );

```

**Parameter**

Parameter	Dir.	Description										
<i>hSigId</i>	[in]	Reference ID of the signal										
<i>dwAttr</i>	[in]	Typ of attribute to be updated, the following constants are possible: <table border="1" data-bbox="598 824 1401 1064"> <thead> <tr> <th>Constant</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>FSL_SIG_ATTR_NAME</td> <td>Name of the signal (not implemented)</td> </tr> <tr> <td>FSL_SIG_ATTR_UNIT</td> <td>Unit of the signal value (not implemented)</td> </tr> <tr> <td>FSL_SIG_ATTR_DLID</td> <td>ID of the default language (signal specific implementation is not supported)</td> </tr> <tr> <td>FSL_SIG_ATTR_PLID</td> <td>ID of the preferred language (signal specific implementation is not supported)</td> </tr> </tbody> </table>	Constant	Description	FSL_SIG_ATTR_NAME	Name of the signal (not implemented)	FSL_SIG_ATTR_UNIT	Unit of the signal value (not implemented)	FSL_SIG_ATTR_DLID	ID of the default language (signal specific implementation is not supported)	FSL_SIG_ATTR_PLID	ID of the preferred language (signal specific implementation is not supported)
Constant	Description											
FSL_SIG_ATTR_NAME	Name of the signal (not implemented)											
FSL_SIG_ATTR_UNIT	Unit of the signal value (not implemented)											
FSL_SIG_ATTR_DLID	ID of the default language (signal specific implementation is not supported)											
FSL_SIG_ATTR_PLID	ID of the preferred language (signal specific implementation is not supported)											
<i>pvData</i>	[in]	Pointer to buffer area with the new data values										
<i>dwSize</i>	[in]	Capacity of the buffer area in byte to which the parameter <i>pvData</i> points. For the attributes FSL_SIG_ATTR_NAME and FSL_SIG_ATTR_UNIT the value can be calculated as follows: (length of character string+1)* size of one character.										

**Return Value**

Return value	Description
VCI_OK	Function succeeded
VCI_E_NOT_IMPLEMENTED	Attribute not supported
!=VCI_OK	Error, more information about error code provides the function VciFormatError

**Remark**

If *hSigId* is set to NULL and either FSL\_SIG\_ATTR\_DLID or FSL\_SIG\_ATTR\_PLID is set for *dwAttr*, the language ID is set for all signals in the set.

## Convert

Converts the defined signal values from raw values to physical values or from physical values to raw values.

```
HRESULT Convert (
    UINT32      dwMode,
    FLSIGNAL    aInSig[],
    FLSIGNAL    aOutSig[],
    UINT32      dwCount );
```

## Parameter

Parameter	Dir.	Description	
<i>dwMode</i>	[in]	Conversion mode, the following constants are possible:	
		FSL_SIG_CONV_RAWTOPHYS	Converts raw values to physical values.
		FSL_SIG_CONV_PHYSTORAW	Converts physical values to raw values.
<i>aInSig</i>	[in]	Pointer to array with elements of type <code>FLSIGNAL</code> . Before calling, the fields <i>hSigId</i> of the elements must be initialized with the reference ID of the signal and the fields <i>sValue</i> must be initialized with valid signal values according to the set mode.	
<i>aOutSig</i>	[out]	Pointer to an array for the converted values. If the values in array <i>aInSig</i> are not needed anymore after calling the function, it is possible that <i>aOutSig</i> points to the same array as <i>aInSig</i> . Then the values are converted directly there.	
<i>dwCount</i>	[in]	Number of elements in arrays <i>aInSig</i> and <i>aOutSig</i>	

## Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

## Remark

If the conversion is not successful the bit `SSL_SIG_STAT_GFAIL` in field *dwStat* in structure [FLSIGNAL](#) is set. If the conversion is successful the bit is deleted.

## Enable

Activates the signal set.

```
HRESULT Enable ( void );
```

## Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

## Remark

The functions registers the signal set at the distributor of the message switch.

**Disable**

Deactivates the signal set.

```
HRESULT Disable ( void );
```

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

The functions deregisters the signal set from the distributor of the message switch.

### 4.3.2 IRSignalSet

The interface extends the basic interface *ISignalSet* with functions to access the receive signal set. *VciCreateCanMsgSwitch* returns a pointer to the CAN specific implementation of the interface. The ID `IID_IRSignalSet` must be used.

#### Read

Reads the received signal values from the buffers of the receive signal set.

```
HRESULT Read (
    BOOL          fConvert,
    FLSIGNAL     aSignal[],
    UINT32        adwRxCnt[],
    UINT32        dwCount );
```

#### Parameter

Parameter	Dir.	Description
<i>fConvert</i>	[in]	TRUE: function converts raw signal values in physical values when reading. FALSE: function delivers raw signal values (e.g. for data logging). Values are only buffered, not interpreted. Reading process is faster without converting. The physical values can be calculated with <b>Convert</b> from the buffered raw values any time.
<i>aSignal</i>	[in/out]	Pointer to array with elements of type <code>FLSIGNAL</code> . Before calling, the fields <i>hSigId</i> of the elements must be initialized with the reference ID of the signal to be read. If run successfully the functions stores the receiving time in field <i>qwTime</i> and the received signal value in field <i>sValue</i> of the respective element.
<i>adwRxCnt</i>	[out]	Pointer to array of type <code>UINT32</code> . If run successfully the function stores the number of each received signal value since the last call of the function. If no signal was received the respective element is set to 0. The array must have at least the capacity of <i>dwCount</i> . If the information is not needed, define value <code>NULL</code> .
<i>dwCount</i>	[in]	Number of receive buffers to be read. Value must be equal or smaller than the number of elements in the arrays <i>aSignal</i> or <i>adwRxCnt</i> .

#### Return Value

Return value	Description
<code>VCI_OK</code>	Function succeeded
<code>!=VCI_OK</code>	Error, more information about error code provides the function <code>VciFormatError</code>

#### Remark

If no message is received in the receive buffer until the first call or between two subsequent calls of the function, the function returns the value 0 in the respective array element of *adwRxCnt*. The receive time of a signal is the receive time of the message that contains the signal. Therefore the format of the timestamp is the same format as in the message.

If an overrun occurs in one of the FIFOs of the signal set or in one of the upstream FIFOs, the receive counter of the respective signals is higher 1 and the bit `FSL_SIG_STAT_RXOVR` in field *dwStat* of structure *FLSIGNAL* is set.

### 4.3.3 ITSignalSet

#### Write

Writes the signal values to the buffers of the transmit signal set.

```
HRESULT Write (
    BOLL      fConvert
    FLSIGNAL  aSignal[],
    BOOL8     afValid[],
    BOOL8     afDone[],
    UINT32    dwCount );
```

#### Parameter

Parameter	Dir.	Description
<i>fConvert</i>	[in]	TRUE: function converts physical values in raw signal values before transmitting. FALSE: the values to be transmitted are raw values. Writing process is faster without converting. The raw values can be calculated with <b>Convert</b> before calling <b>Write</b> .
<i>aSignal</i>	[out]	Pointer to array with the signals to be written. Before calling, the fields <i>hSigid</i> of the elements must be initialized with the reference ID of the signal to be written. Field <i>qwTime</i> is ignored. Dependent on the settings in <i>fConvert</i> (TRUE or FALSE) field <i>sValue</i> must contain the physical value or the raw value.
<i>afValid</i>	[in]	Pointer to array that defines if the value of a signal in the array <i>aSignal</i> is valid and ready for transmission. Element <i>afValid[x]</i> must be TRUE to adopt the signal value in signal <i>aSignal[x]</i> . If <i>afValid[x]</i> is FALSE the signal value is not adopted.
<i>afDone</i>	[out]	Pointer to array of type BOOL8. If run successfully the function sets the individual elements to TRUE or FALSE, depending if the respective signal value is written in the corresponding transmit buffer or not. If the information is not needed, define value NULL.
<i>dwCount</i>	[in]	Number of signal values to be written. Value must be smaller than the number of elements in the arrays <i>aSignal</i> , <i>afValid</i> and optionally <i>afDone</i> .

#### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

## 4.4 CAN Specific Interfaces

### 4.4.1 Message Switch: ICanMsgSwitch

The interface is used to access the message switch. *VciCreateCanMsgSwitch* returns a pointer to the interface. The ID is IID\_ICanMsgSwitch.

#### Initialize

Initializes the distributor thread and CAN message channel of the message switch with the defined parameters.

```
HRESULT Initialize (
    UINT32 dwTiming
    INT32 lPriority
    BOOL fExclusiv
    UINT16 wRxFifoSize
    UINT16 wTxFifoSize );
```

#### Parameter

Parameter	Dir.	Description
<i>dwTiming</i>	[in]	Cycle time of the message distributor in milliseconds. Defines the minimal cycle time of the individual message sources.
<i>lPriority</i>	[in]	Priority of the message distributor, possible values: <code>THREAD_PRIORITY_NORMAL</code> (for non time critical applications), <code>THREAD_PRIORITY_ABOVE_NORMAL</code> , <code>THREAD_PRIORITY_HIGHEST</code> (for time critical applications), <code>THREAD_PRIORITY_TIME_CRITICAL</code> (highest priority) Observe the priority class of the process that is created by the message switch. Create this priority class with Windows API function <code>SetPriorityClass</code> . For more information see Windows API function documentation.
<i>fExclusive</i>	[in]	Defines if the CAN connection is used exclusively by the message switch to be opened. If <code>TRUE</code> is defined no other message channels can be opened after successful call of the function until the channel is released again. If <code>FALSE</code> is defined further message channels can be opened for the CAN connection.
<i>wRxFifoSize</i>	[in]	Capacity of receive FIFO in number of CAN messages of structure <code>CANMSG2</code>
<i>wTxFifoSize</i>	[in]	Capacity of transmit FIFO in number of CAN messages of structure <code>CANMSG2</code>

#### Return Value

Return value	Description
<code>VCI_OK</code>	Function succeeded
<code>VCI_E_ACCESSDENIED</code>	Connection can not be used, because another application uses the connection exclusively.
<code>!=VCI_OK</code>	Error, more information about error code provides the function <code>VciFormatError</code>

**Activate**

Activates the message switch and starts the message distributor.

```
HRESULT Activate ( void );
```

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

**Remark**

After creating or initializing the message switch is per default deactivated and disconnected from the bus. To connect the message channel of the switch with the bus, the switch must be activated. Then messages can be transmitted to the bus and received from the bus. The CAN controller must be in status *online*. For more information see *VCI: C++ Software Design Guide* in chapter *CAN Controller*.

**Deactivate**

Stops the message distributor and deactivates the message channel to the CAN connection.

```
HRESULT Activate ( void );
```

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

**Remark**

When the message distributor is deactivated, it is not possible to transmit messages to the CAN bus and the CAN bus does not receive any messages.

**ForceReceive**

Triggers the message distributor as if a message is received in the receive FIFO of the message channel.

```
HRESULT ForceReceive ( void );
```

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

**ForceTransmit**

Triggers the message distributor as if a message is transmitted from the transmit FIFO of the message channel.

```
HRESULT ForceTransmit ( void );
```

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**GetStatus**

Gets the status of the message switch, message channel and CAN controller.

```
HRESULT GetStatus ( PCANMSGSWITCHSTATUS pStatus );
```

**Parameter**

Parameter	Dir.	Description
<i>pStatus</i>	[out]	Pointer to a buffer area of type <a href="#">CANMSGSWITCHSTATUS</a> . If run successfully the function stores the current state of the message switch, message channel and CAN controller in the buffer area.

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

The function can be called anytime, even before calling *Initialize*. For more information see description of structure [CANMSGSWITCHSTATUS](#).

**GetControl**

Opens the control unit of the connection the message switch is connected to.

```
HRESULT GetControl ( PCANCONTROL2* ppCanCtrl );
```

**Parameter**

Parameter	Dir.	Description
<i>ppCanCtrl</i>	[out]	Address of variable that gets a pointer to the interface <code>ICanControl2</code> if run successfully. Pointer is allocated by the opened control unit. In case of an error the variable is set to <code>NULL</code> .

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

The control unit of a connection can exclusively be opened once by one application at a time. If the control unit is not needed anymore, the pointer delivered in *ppCanCtrl* must be released by calling *Release*.

**CreateClient**

Creates a client for the message switch.

```
HRESULT CreateClient ( REFIID riid, PVOID* ppv );
```

**Parameter**

Parameter	Dir.	Description
<i>riid</i>	[in]	ID of the interface of the client that is to be created. The following IDs are possible: IID_ICanRMsgBuffer, IID_ICanRMsgQueue, IID_ICanRMsgSet, IID_ICanTMsgBuffer, IID_ICanTMsgQueue, IID_ICanTMsgSet, IID_ICanRSignalSet, IID_ICanTSignalSet
<i>ppv</i>	[out]	Address of the variable to which the pointer to the desired interface of the newly created client is allocated if the function is run successfully. In case of an error the variable is set to NULL.

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <i>VciFormatError</i>

**Remark**

A newly created client must be initialized and activated before being able to receive messages or to transmit messages.

**AttachClient**

Registers a client at the distributor of the message switch.

```
HRESULT AttachClient ( IUnknown* pClient );
```

**Parameter**

Parameter	Dir.	Description
<i>pClient</i>	[in]	Pointer to interface <i>IUnknown</i> of the client that is to be registered at the distributor

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <i>VciFormatError</i>

**Remark**

The client must be registered at the distributor before being able to receive or transmit messages. The client registers automatically at the distributor when function *Enable* of the client is called. For more information see [<\(2.1.4 link\)>](#).

**DetachClient**

Deregisters a client from the distributor of the message switch.

```
HRESULT DetachClient ( IUnknown* pClient );
```

**Parameter**

Parameter	Dir.	Description
<i>pClient</i>	[in]	Pointer to interface <code>IUnknown</code> of the client that is to be deregistered from the distributor

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

The client deregisters automatically from the distributor when function `Disable` of the client is called or if the client is removed by calling `Release`. For more information see [<\(2.1.4 link\)>](#).

**4.4.2 Message Sink: ICanRMsgBuffer**

The interface is a supported sink for CAN messages that are received by the bus. The ID of the interface is `IID_ICanRMsgBuffer`.

**Enable**

Initializes and activates the receive buffer.

```
HRESULT Enable ( UINT32 dwCanId );
```

**Parameter**

Parameter	Dir.	Description
<i>dwCanId</i>	[in]	ID of the CAN message the receive buffer is intended for

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

The function registers the receive buffer at the distributor of the message switch with an internal call of `ICanMsgSwitch::AttachClient`.

**Disable**

Deactivates the receive buffer.

```
HRESULT Disable ( void );
```

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

The function deregisters the receive buffer at the distributor of the message switch with an internal call of `ICanMsgSwitch::DetachClient`.

**Read**

Reads the last received CAN messages from the receive buffer.

```
HRESULT Read ( PCANMSG2 pCanMsg, PUINT32 pdwRxCnt );
```

**Parameter**

Parameter	Dir.	Description
<i>pCanMsg</i>	[out]	Pointer to buffer area of type <code>CANMSG2</code> . If run successfully the last received message is buffered in the defined area.
<i>pdwRxCnt</i>	[out]	Pointer to variable of type <code>UINT32</code> . If run successfully number of messages received since the last call of the function are buffered here.

**Return Value**

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALID_STATE	No message with defined CAN ID received
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

### 4.4.3 Message Sink: ICanRMsgQueue

The interface is a supported sink for CAN messages that are received by the bus. The ID of the interface is IID\_ICanRMsgQueue.

#### Enable

Initializes and activates the receive queue.

```
HRESULT Enable ( UINT32 dwCanId, UINT16 wDepth );
```

#### Parameter

Parameter	Dir.	Description
<i>dwCanId</i>	[in]	ID of the CAN message the receive queue is intended for
<i>wDepth</i>	[in]	Capacity of receive queue in number of CAN messages

#### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

#### Remark

The function registers the receive queue at the distributor of the message switch with an internal call of ICanMsgSwitch:: *AttachClient*.

#### Disable

Deactivates the receive queue.

```
HRESULT Disable ( void );
```

#### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

#### Remark

The function deregisters the receive buffer at the distributor of the message switch with an internal call of ICanMsgSwitch:: *DetachClient*.

**Read**

Reads one or more received CAN messages from the receive queue.

```
HRESULT Read (
    CANMSG2 aCanMsg[],
    UINT32 dwCount
    PUINT32 pdwDone );
```

**Parameter**

Parameter	Dir.	Description
<i>aCanMsg</i>	[out]	Pointer to buffer area of type CANMSG2. If run successfully received messages are buffered in the defined area.
<i>dwCount</i>	[in]	Capacity of array <i>aCanMsg</i> in number of CAN messages
<i>pdwDone</i>	[out]	Pointer to variable of type UINT32. If run successfully number of read messages is buffered.

**Return Value**

Return value	Description
VCI_OK	Function succeeded
VCI_E_RXQUEUE_EMPTY	No message with defined CAN ID received
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

#### 4.4.4 Message Sink: ICanRMsgSet

The interface is a supported sink for CAN messages that are received by the bus. The ID of the interface is IID\_ICanRMsgSet.

##### Enable

Initializes and activates the receive message set.

```
HRESULT Enable (
    UINT32 adwCanId[],
    UINT16 awDepth[],
    UINT32 dwCount );
```

##### Parameter

Parameter	Dir.	Description
<i>adwCanId</i>	[in]	Array of CAN IDs
<i>awDepth</i>	[in]	Array with capacity of the buffer in number of CAN messages. If an element of the array is higher 1, a FIFO is created for the message with the respective number of messages. If the element is smaller 1 a simple buffer is created. The value in <i>awDepth[0]</i> defines the size of the buffer for the CAN ID defined in <i>adwCanId[0]</i> , the value in <i>awDepth[1]</i> defines the size of the buffer for the CAN ID defined in <i>adwCanId[1]</i> , etc. Value NULL creates a simple buffer for each message that is defined in <i>adwCanId</i> . If the pointer value is smaller 65536 a FIFO of the same size is created for each defined message. The pointer value then defines the buffer capacity of the FIFO.
<i>dwCount</i>	[in]	Number of elements in the arrays that are specified in <i>adwCanId</i> and <i>awDepth</i>

##### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

##### Remark

The function registers the receive message set at the distributor of the message switch with an internal call of `ICanMsgSwitch::AttachClient`.

##### Disable

Deactivates the receive message set.

```
HRESULT Disable ( void );
```

##### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

##### Remark

The function deregisters the receive message set at the distributor of the message switch with an internal call of `ICanMsgSwitch::DetachClient`.

**Read**

Reads the last received message of each individual buffer of the receive message set.

```
HRESULT Read (
    CANMSG2 aCanMsg[],
    UINT32 adwRxCnt[],
    UINT32 dwFirst,
    UINT32 dwCount );
```

**Parameter**

Parameter	Dir.	Description
<i>aCanMsg</i>	[out]	Pointer to buffer area of type CANMSG2. If run successfully the last received message is buffered in the defined area. If value NULL is defined, the next message in the receive buffers is removed and the receive counter is reset.
<i>adwRxCnt</i>	[out]	Pointer to array of type UINT32. If run successfully the last received messages are buffered in the defined area. If the information is not needed, define value NULL.
<i>dwFirst</i>	[in]	Index of first receive buffer in the message set. Value must be smaller than the value of <i>dwCount</i> when calling <code>Enable</code> .
<i>dwCount</i>	[in]	Number of receive buffers to be read. Value must be equal or smaller than the number of elements in the arrays <i>aCanMsg</i> or <i>adwRxCnt</i> .

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

If no message is received until the first call or between two subsequent calls, the function returns 0 in the array element of *adwRxCnt*.

#### 4.4.5 Message Source: ICanTMsgBuffer

The interface is a supported sink for CAN messages that are transmitted by the bus. The ID of the interface is IID\_ICanTMsgBuffer.

##### Enable

Initializes and activates the transmit buffer.

```
HRESULT Enable (
    UINT32 dwCanId,
    UINT32 dwMode,
    UINT32 dwTime );
```

##### Parameter

Parameter	Dir.	Description						
<i>dwCanId</i>	[in]	ID of the CAN message accepted by the buffer. Value 0xFFFFFFFF selects the CAN ID of the message that is currently stored in the internal buffer.						
<i>dwMode</i>	[in]	Operation mode of transmit buffer: <table border="1" data-bbox="730 801 1401 936"> <tr> <td>CAN_TX_DIRECT:</td> <td>Messages are directly transmitted during writing.</td> </tr> <tr> <td>CAN_TX_CYCLIC:</td> <td>Message in transmit buffer is transmitted cyclically.</td> </tr> <tr> <td>CAN_TX_DELAYED:</td> <td>Messages are transmitted delayed.</td> </tr> </table>	CAN_TX_DIRECT:	Messages are directly transmitted during writing.	CAN_TX_CYCLIC:	Message in transmit buffer is transmitted cyclically.	CAN_TX_DELAYED:	Messages are transmitted delayed.
CAN_TX_DIRECT:	Messages are directly transmitted during writing.							
CAN_TX_CYCLIC:	Message in transmit buffer is transmitted cyclically.							
CAN_TX_DELAYED:	Messages are transmitted delayed.							
<i>dwTime</i>	[in]	Cycle time or delay time of transmit buffer in milliseconds						

##### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

##### Remark

The function registers the transmit buffer at the distributor of the message switch with an internal call of ICanMsgSwitch:: [AttachClient](#).

##### Disable

Deactivates the transmit buffer.

```
HRESULT Disable ( void );
```

##### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

##### Remark

The function deregisters the transmit buffer at the distributor of the message switch with an internal call of ICanMsgSwitch:: [DetachClient](#).

**Write**

Writes a CAN message to the transmit buffer.

```
HRESULT Enable ( PCANMSG2 pCanMsg );
```

**Parameter**

Parameter	Dir.	Description
<i>pCanMsg</i>	[in]	Pointer to the message to be written

**Return Value**

Return value	Description
VCI_OK	Function succeeded
VCI_E_INVALID_STATE	No message with defined CAN ID received
!=VCI_OK	Error, more information about error code provides the function <i>VciFormatError</i>

#### 4.4.6 Message Source: ICanTMsgQueue

The interface is a supported sink for CAN messages that are transmitted by the bus. The ID of the interface is IID\_ICanTMsgQueue.

##### Enable

Initializes and activates the transmit queue.

```
HRESULT Enable (
    UINT32 dwCanId,
    UINT16 wDepth
    UINT32 dwMode,
    UINT32 dwTime );
```

##### Parameter

Parameter	Dir.	Description						
<i>dwCanId</i>	[in]	ID of the CAN message the transmit queue is intended for. Value 0xFFFFFFFF selects the CAN ID of the messages that are currently in the FIFO.						
<i>wDepth</i>	[in]	Capacity of transmit queue in number of CAN messages						
<i>dwMode</i>	[in]	Operation mode of transmit queue: <table border="1" data-bbox="683 869 1401 1025"> <tr> <td>CAN_TX_DIRECT:</td> <td>Messages are transmitted directly during writing.</td> </tr> <tr> <td>CAN_TX_CYCLIC:</td> <td>Messages in the queue are transmitted cyclically.</td> </tr> <tr> <td>CAN_TX_DELAYED:</td> <td>Messages in the queue are transmitted delayed.</td> </tr> </table>	CAN_TX_DIRECT:	Messages are transmitted directly during writing.	CAN_TX_CYCLIC:	Messages in the queue are transmitted cyclically.	CAN_TX_DELAYED:	Messages in the queue are transmitted delayed.
CAN_TX_DIRECT:	Messages are transmitted directly during writing.							
CAN_TX_CYCLIC:	Messages in the queue are transmitted cyclically.							
CAN_TX_DELAYED:	Messages in the queue are transmitted delayed.							
<i>dwTime</i>	[in]	Cycle or delay time in milliseconds						

##### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

##### Remark

The function registers the receive queue at the distributor of the message switch with an internal call of ICanMsgSwitch::AttachClient.

##### Disable

Deactivates the transmit queue.

```
HRESULT Disable ( void );
```

##### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function VciFormatError

##### Remark

The function deregisters the receive buffer at the distributor of the message switch with an internal call of ICanMsgSwitch::DetachClient.

**Write**

Writes one or more CAN messages in the transmit queue.

```
HRESULT Write (
    CANMSG2 aCanMsg[],
    UINT32 dwCount
    PUINT32 pdwDone );
```

**Parameter**

Parameter	Dir.	Description
<i>aCanMsg</i>	[in]	Pointer to array with the CAN messages to be transmitted
<i>dwCount</i>	[in]	Number of the CAN messages to be transmitted
<i>pdwDone</i>	[out]	Pointer to variable of type <code>UINT32</code> . If run successfully number of transmitted messages is buffered.

**Return Value**

Return value	Description
<code>VCI_OK</code>	Function succeeded
<code>VCI_E_TXQUEUE_FULL</code>	Not all messages written
<code>!=VCI_OK</code>	Error, more information about error code provides the function <code>VciFormatError</code>

#### 4.4.7 Message Source: ICanTMsgSet

The interface is a supported sink for CAN messages that are transmitted by the bus. The ID of the interface is IID\_ICanTMsgSet.

##### Enable

Initializes and activates the transmit message set.

```
HRESULT Enable (
    UINT32 adwCanId[],
    UINT16 awDepth[],
    UINT32 adwMode[],
    UINT32 adwTime[],
    UINT32 dwCount );
```

##### Parameter

Parameter	Dir.	Description						
<i>adwCanId</i>	[in]	Array of CAN IDs. Value 0xFFFFFFFF selects the CAN ID of the messages that are currently in the buffer or in the queue.						
<i>awDepth</i>	[in]	Array with capacity of the buffer in number of CAN messages. If an element of the array is higher 1, a FIFO is created for the message with the respective number of messages. If the element is smaller 1 a simple buffer is created. The value in <i>awDepth[0]</i> defines the size of the buffer for the CAN ID defined in <i>adwCanId[0]</i> , the value in <i>awDepth[1]</i> defines the size of the buffer for the CAN ID defined in <i>adwCanId[1]</i> , etc. Value NULL creates a simple buffer for each message that is defined in <i>adwCanId</i> . If the pointer value is smaller 65536 a FIFO of the same size is created for each defined message. The pointer value then defines the buffer capacity of the FIFO.						
<i>adwMode</i>	[in]	Array with operation mode of the buffers. For each buffer one of the following operation modes is possible: <table border="1" data-bbox="671 1106 1402 1263"> <tr> <td>CAN_TX_DIRECT:</td> <td>Messages in the buffer are directly transmitted during writing.</td> </tr> <tr> <td>CAN_TX_CYCLIC:</td> <td>Messages in the buffer are transmitted cyclically.</td> </tr> <tr> <td>CAN_TX_DELAYED:</td> <td>Messages in the buffer are transmitted delayed.</td> </tr> </table>	CAN_TX_DIRECT:	Messages in the buffer are directly transmitted during writing.	CAN_TX_CYCLIC:	Messages in the buffer are transmitted cyclically.	CAN_TX_DELAYED:	Messages in the buffer are transmitted delayed.
CAN_TX_DIRECT:	Messages in the buffer are directly transmitted during writing.							
CAN_TX_CYCLIC:	Messages in the buffer are transmitted cyclically.							
CAN_TX_DELAYED:	Messages in the buffer are transmitted delayed.							
<i>adwTime</i>	[in]	Array with cycle or delay time of the buffers milliseconds						
<i>dwCount</i>	[in]	Capacity of message set, number of elements that are specified in the arrays <i>adwCanId</i> , <i>awDepth</i> , <i>adwMode</i> , and <i>adwTime</i>						

##### Return Value

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

##### Remark

The function registers the transmit message set at the distributor of the message switch with an internal call of `ICanMsgSwitch::AttachClient`.

**Disable**

Deactivates the transmit message set.

```
HRESULT Disable ( void );
```

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

**Remark**

The function deregisters the transmit message set at the distributor of the message switch with an internal call of `ICanMsgSwitch::DetachClient`.

**Write**

Writes CAN messages in the transmit buffers of the message set.

```
HRESULT Write (
    CANMSG2 aCanMsg[],
    BOOL8 afValid[],
    BOOL8 afDone[],
    UINT32 dwFirst,
    UINT32 dwCount );
```

**Parameter**

Parameter	Dir.	Description
<i>aCanMsg</i>	[in]	Pointer to array with CAN messages to be written
<i>afValid</i>	[in]	Pointer to array that shows if a message in the array <i>aCanMsg</i> is valid or not. If <i>afValid[x]</i> is <code>TRUE</code> the message in element <i>aCanMsg[x]</i> is written in the corresponding transmit buffer. If <i>afValid[x]</i> is <code>FALSE</code> the message is not adopted.
<i>afDone</i>	[out]	Pointer to array of type <code>BOOL8</code> . If run successfully each element is set to either <code>TRUE</code> or <code>FALSE</code> depending if the message is written in the internal transmit buffer or not. If the information is not needed, define value <code>NULL</code> .
<i>dwFirst</i>	[in]	Index of first transmit buffer in the message set. The value must be smaller than the value of <i>dwCount</i> when calling <code>Enable</code> .
<i>dwCount</i>	[in]	Number of messages to be written. Value must be smaller than the number of elements in the arrays <i>aCanMsg</i> , <i>afValid</i> , and optionally <i>afDone</i> .

**Return Value**

Return value	Description
VCI_OK	Function succeeded
!=VCI_OK	Error, more information about error code provides the function <code>VciFormatError</code>

## 5 Data Structures

### 5.1 CAN Specific Data Types

#### 5.1.1 CANMSGSWITCHSTATUS

The data type describes the current state of a CAN message switch.

```
typedef struct _CANMSGSWITCHSTATUS
{
    CANCHANSTATUS2 sChanStatus;
    UINT32          dwRxClients;
    UINT32          dwTxClients;
    UINT8           bWorkLoad;
} CANMSGSWITCHSTATUS, *PCANMSGSWITCHSTATUS;
```

Member	Dir.	Description
<i>sChanStatus</i>	[out]	Current state of CAN message channel, for more descriptions see <i>VCI: C++ Software Design Guide</i> .
<i>dwRxClients</i>	[out]	Current number of attached message sinks (RX clients)
<i>dwTxClients</i>	[out]	Current number of attached message sources (TX clients)
<i>bWorkLoad</i>	[out]	Current load of message distributor in percent (0 to 100)

### 5.2 Signal Specific Data Types

#### 5.2.1 FSLVAR

The data type describes the structure of a signal variable. The buffer layout of the structure is binary compatible to the 32 bit version of the Windows data type `VARIANT`. The 64 bit version of the Windows data type `VARIANT` is 8 byte bigger. This version reserves space for 2 pointers with each 8 byte in the union and therefore includes 24 bytes instead 16 bytes.

```
typedef struct _FSLVAR
{
    UINT16 wVarType;
    UINT16 _rsvd1_;
    UINT16 _rsvd2_;
    UINT16 _rsvd3_;
    union
    {
        FSL_BOOL    asBool;
        FSL_INT8    asInt8;
        FSL_UINT8   asUInt8;
        FSL_INT16   asInt16;
        FSL_UINT16  asUInt16;
        FSL_INT32   asInt32;
        FSL_UINT32  asUInt32;
        FSL_INT64   asInt64;
        FSL_UINT64  asUInt64;
        FSL_INT32   asInt;
        FSL_UINT32  asUInt;
        FSL_SINGLE  asSingle;
        FSL_DOUBLE  asDouble;
        FSL_BSTR    asBStr;
    };
} FSLVAR, *PFSLVAR;
```

Member	Dir.	Description
<i>wVarType</i>	[out]	Data type of the variant, the following values are possible

Member	Dir.	Description	
		FSL_VT_EMPTY	Empty, no data
		FSL_VT_BOOL	Boolean (0 = FALSE, 1 = TRUE)
		FSL_VT_INT8, FSL_VT_UINT8	Signed or unsigned 8 bit integer
		FSL_VT_INT16, FSL_VT_UINT16	Signed or unsigned 16 bit integer
		FSL_VT_INT, FSL_VT_INT32, FSL_VT_UINT, FSL_VT_UINT32	Signed or unsigned 32 bit integer
		FSL_VT_INT64, FSL_VT_UINT64	Signed or unsigned 64 bit integer
		FSL_VT_SINGLE	32 bit floating point
		FSL_VT_DOUBLE	64 bit floating point
		FSL_VT_BSTR	Pointer to a <i>BSTR</i> whose buffer is reserved with one of the Windows API functions <i>SysAllocString</i> , <i>SysAllocStringByteLen</i> , <i>SysAllocStringLen</i> , <i>SysReAllocString</i> or <i>SysReAllocStringLen</i> and released with <i>SysFreeString</i> .
<i>_rsvd1_</i> , <i>_rsvd2_</i> , <i>_rsvd3_</i>	[out]	Reserved, not used	
<i>asBool</i> , <i>asInt8</i> , <i>asUInt8</i> , <i>asInt16</i> , <i>asUInt16</i> , <i>asInt32</i> , <i>asUInt32</i> , <i>asInt64</i> , <i>asUInt64</i> , <i>asInt</i> , <i>asUInt</i> , <i>asSingle</i> , <i>asDouble</i> , <i>asBStr</i>	[in/out]	Value of the variant, value range is determined by data type that is defined in field <i>wVarType</i>	

## 5.2.2 FLSIGNAL

The data type describes the structure of a signal.

```
typedef struct _FLSIGNAL
{
    HANDLE hSigId;
    UINT64 qwTime;
    UINT32 dwStat;
    FSLVAR sValue;
} FLSIGNAL, *PFLSIGNAL;
```

Member	Dir.	Description
<i>hSigId</i>	[in]	Signal ID, the ID of a signal is returned by <a href="#">LoadDB</a> .
<i>qwTime</i>	[out]	Receive time of signal
<i>dwStat</i>	[out]	Signal status flags, value is a logical combination of one or several of the following constants: FSL_SIG_STAT_GFAIL: error in converting the signal value FSL_SIG_STAT_GFAIL: overrun in one of the receive buffers
<i>sValue</i>	[in/out]	Value of signal

**This page intentionally left blank**

