Manual

VCI - Virtual CAN Interface

VCI-V2 Programmers Manual

IXXAT

The expert for industrial and automotive communication

IXXAT

Headquarter
IXXAT Automation GmbH
Leibnizstr. 15
D-88250 Weingarten

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

US Sales Office
IXXAT Inc.
120 Bedford Center Road
USA-Bedford, NH 03110

Phone: +1-603-471-0800
Fax: +1-603-471-0880
Internet: www.ixxat.com
e-Mail: sales@ixxat.com

Support

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

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

For customers from the USA/Canada
Fax: +1-603-471-0880
e-Mail: techsupport@ixxat.com

Copyright

Document No. VCIV2_Programmers Manual-E-V2.9

1 Introduction

The Virtual CAN Interfaces (**VCI**) is a powerful software package for the IXXAT-PC/CAN-Interfaces. It has been designed for software developers who wish to develop high-quality, hardware-independent CAN-Applications for PCs.

For this reason, particular importance was placed on easy application and good real-time behavior of the VCI.

1.1 Areas of Application

The aim of the VCI is to provide the user with a unified programming interface for the various PC/CAN-interface versions of the IXXAT company. For this, neither the design of the PC-connection (PCI(e), USB, TCP-IP ...) nor the CAN Controller of the interface used is important. In addition, the VCI makes it possible to operate several (even different) cards at the same time.

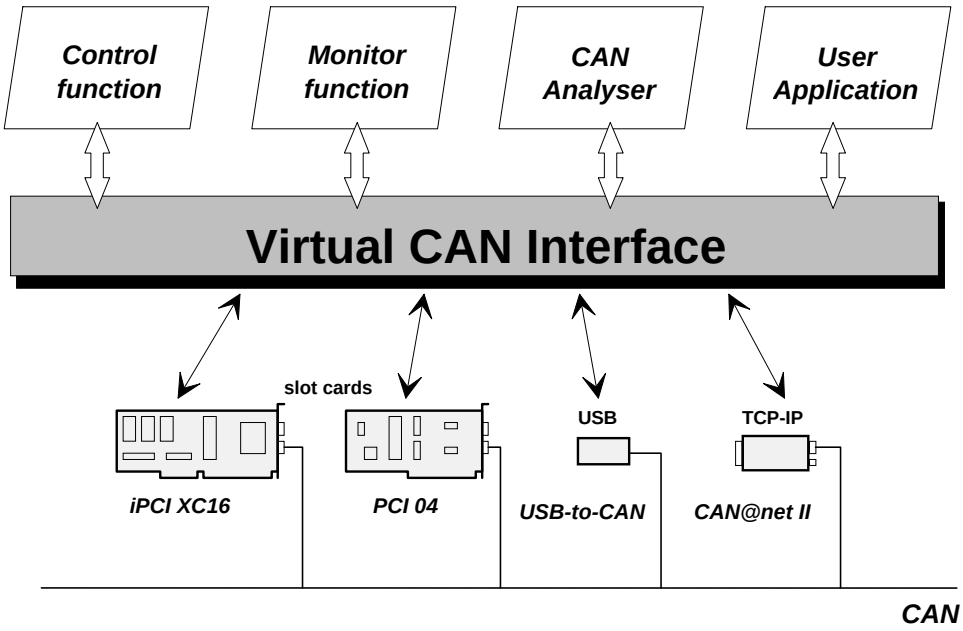

Fig. 1 - 1 Virtual CAN Interface

This concept enables realization of application programs independent of the PC/CAN-interface type used.

For this, a virtual CAN Controller was defined in the VCI, the structure of which corresponds to a Basic-CAN-Controller and which supports operation with 11-bit and 29-bit identifiers. Downstream from this virtual CAN-Controller a Firmware is installed which organizes the message administration. The virtual CAN-Controller

can be present on a PC/CAN-Interface up to 4 times, whereby simultaneous operation of up to 4 cards is possible.

Both intelligent PC/CAN-interfaces (with their own memory and CPU) and passive cards are supported by the VCI.

Active PC/CAN-interface cards support the PC in the pre-processing of the CAN-messages and in data management. This has a positive effect on the load of the processor of the PC.

With passive cards the processor of the PC is loaded considerably more by the interrupt routine of the CAN-Controller and the message administration. On the other hand, passive PC/CAN-interface cards make possible a reasonably priced connection of a PC to a CAN-network. However, high demands are made on the real-time behavior of the PC (under Windows only useful with low baud rates).

1.2 Notes on this Manual

The aim of this manual is to explain the way of functioning of the VCI and its functions.

This manual does **not** intend to describe the whole area of programming of CAN-applications nor to represent a reference for the functionality of individual CAN-Controllers.

This manual requires knowledge of programming under MS-Windows (Multi-Threading, event-controlled processing).

Before working with the VCI, it is **absolutely necessary** that you read through this manual completely at least once.

In order to keep this documentation as short as possible, the information it contains is given to a great extent without redundancy. It is therefore recommended that you work through the manual several times, since important information is often overlooked on the first reading.

In this connection we also strongly recommend studying the header files VCI.h.

1.3 Installation of the VCI

For information on the installation of the PC/CAN-interface, please see the "PC/CAN-interface hardware manual" supplied.

The procedure for the VCI-software installation is described in the "CAN-Driver VCI installation manual" .

1.4 Functional Scope of the VCI

The VCI supports:

- Standard and Extended Protocol (11 and 29-bit-Identifier)
- Several CAN-Controllers per interface (if supported by the hardware)
- Simultaneous operation of up to four interfaces by one or more applications
- Baud rates of up to 1000 Kbaud
- Reception of messages via configurable receive queues (FIFO) with time marker
- Reception of messages via configurable receive buffers with receive counter.
- Several queues and buffers can be assigned to each CAN-Controller.
- Sending of messages (via configurable send queues)
- Queues can be polled or read per interrupt (Timeout or ´High water mark´)
- Automatic, configurable response to request messages (Remote frames) (only in 11 Bit Standard protocol)

In addition, the VCI supplies statistic data to the CAN-Bus, to the CAN-Controller, via the data structures and the PC/CAN-interfaces.

1.5 Limitations

- Access to a PC/CAN-interface is only possible for one application. Therefore several applications cannot share one PC/CAN-interface.
- Depending on the CAN-Controller of the PC/CAN-interface used, there are limitations in the functional scope of the VCI:
  - Philips 82C200: No extended protocol possible
  - Intel 82527: No remote operation possible
  - Philips SJA1000: No limitations

  In order to support the corresponding functionality, your application must select the corresponding CAN-Controller.
- Remote Buffers only possible in 11Bit Standard Mode

1.6 Message Administration

The VCI possesses its own message administration. In this message administration, both incoming and outgoing messages are administered in different structures.

Intermediate storage of the received messages occurs in so-called Receive queues or in Receive buffers. In the case of a Queue, the messages are stored in order of the time of their occurrence including a time marker (FIFO-principle), whereby the messages can also have different identifiers. In contrast, a buffer contains only the last message received under a certain identifier (according to a current process diagram) as well as a counter for the number of the receive processes on this buffer.

The messages to be sent are recorded in Transmit queues. These are then serviced by a microcontroller (only with intelligent PC/CAN-interfaces) or by an interrupt routine of the PC. In addition, so-called Remote buffers can be created, in which messages are entered which are not sent directly but only on request (Remote-Frame) by another node.

The following section describes the elements provided by the VCI for the administration of the CAN-messages.

1.6.1 Receive buffers

Receive buffers are created for each identifier to be received. They always contain the last data received under the selected identifier. This means that data which have not yet been accepted by the application are overwritten. For flow control with repeated reception, the Receive buffers are equipped with a Receive counter. Receive buffers are specifically checked by the application for the presence of new data and the data are then accepted.

An event-controlled reading of Receive buffers is not possible, since Receive buffers generally come into use when the application has to check certain process data only sporadically and is only concerned with the most recent data in each case.

The maximum configurable number of buffers (Receive and Remote buffers together) is 2048 per CAN-Controller.

1.6.2 Receive queues

The use of Receive queues is recommended in particular for applications where all data are to be received which are transmitted under one or more identifiers and for which the application program is not able to react directly to the reception of a message.

The application program can determine how many messages a queue can accept and which identifiers should be assigned to a queue. Several queues can be

created, so that pre-sorting can already be carried out by the VCI. All messages recorded in a Receive queue are provided with a time stamp.

If due to the structure of the application program a regular polling of the Receive queue(s) is not useful or not possible, the application can be informed via a Call-back-function. The time of informing can be configured and depends on two events of which one must occur to trigger the callback:

- A certain number of entries is present in a queue ("high water mark" is reached)
- After a certain amount of time has expired (Timeout function).

The Call-back-function is called up from the Interrupt-thread of the VCI. This gives rise to several limitations:

- In the Call-back-function no time-critical calculations should be carried out, as otherwise CAN-messages may be lost.
- They are located in the Call-back-function in the context of the Interrupt-threads. An attempt to access data from its application may fail for this reason.
  One way to uncouple Call-back from its application is to start an application-thread for processing a queue. Incoming CAN-messages are signaled in their Call-back-function by the setting of an event. The application-thread waits for this event and carries out processing after the Event has been set. After the processing step it returns to wait mode.

The maximum number of Receive queues which can be configured is 16 per CAN-Controller.

1.6.3 Transmit queues

Messages (data and data requests) from the application are normally sent via Transmit queues. In this way, when making a request to send, the application does not need to wait until the CAN-Controller is ready to transmit. Servicing of the Transmit queue(s) is carried out by the microcontroller of the active PC/CAN-interfaces or with passive PC/CAN-interfaces by the Interrupt routine of the PC.

Several queues of different sizes (number of messages) and different priority can be created. The different priorities of the queues determine the order in which they are processed by the microcontroller.

The maximum number of Transmit queues per CAN-Controller which can be configured is 8.

1.6.4 Remote buffers

If data are to be kept available for requests by other nodes, they can be entered in so-called Remote buffers. If a request message is received (Remote frame) with appropriate identifier, the data are taken from the buffer and sent. The application only needs to update the data in the buffer. Processing of a request message is carried out with highest priority, which is before the processing of the Transmit queues.

Alternatively, request messages can also be received via a Receive queue. In this case the application initiates sending of the requested data itself, by entering a corresponding message in a Transmit queue.

The maximum number of buffers which can be configured (Receive and Remote buffers together) is 2048 per CAN-Controller.

1.6.5 Opening a PC-CAN Interface

VCI2_PrepareBoard or VCI2_PrepareBoardMsg are called to open an IXXAT PC-CAN Interface. The interface board you want to open is identified by an index number which is to be ascertained via one of four possible ways first:

- Manual selection of the interface in the Hardware Selection dialog (refer chapter 3.2.1).
- Query the attributes of the interface that is declared as "Default" in the IXXAT Interfaces Control Panel applet (refer chapter 3.2.6).
- Enumerate all installed IXXAT PC-CAN Interfaces (refer chapter 3.2.3).
- Search for an IXXAT PC-CAN Interface by means of specific board attributes (refer chapter 3.2.4).

2 Interface Description

The VCI-user interface provides the user with a collection of functions for the PC which access PC/CAN-Interface and handle communication via CAN. The interface distinguishes four different classes of functions:

- functions for the control and configuration of the PC/CAN interface
- functions for checking and configuration of the CAN-Controller
- functions to receive messages
- functions to send messages

The functions are described in the following section. Example programs supplied show the uses of the functions.

2.1 Pre-defined Return Codes of the VCI

In order to be able to support other PC/CAN-interface types in future, and as it is not possible to specify all errors and Return codes today which may occur in future implementations, all possible Return codes are described via the following Defines. Additional information (error string and further parameters) is provided by the Exception handler of the VCI (Call-back-function).

Define	*Value*	*Error description*
VCI_OK	1	Successful, not further specified message for functions carried out correctly
VCI_ERR	0	Standard error message, further specification provided by the Exception handler
VCI_QUE_EMPTY	0	The Receive queue is empty, no messages can be read
VCI_QUE_FULL	0	The Transmit queue is already full, no further entries can be made at the moment
VCI_OLD	0	There are no new data in the Receive buffer, old data are read if applicable
VCI_HWSW_ERR	-1	Function could not be carried out due to hardware or software errors; check function of the PC/CAN-interface
VCI_SUPP_ERR	-2	Function is not supported in this form (support error); check your error with the implementation overview of your platform
VCI_PARA_ERR	-3	Parameter(s) transferred is/are faulty or outside the permitted range; check the parameters transferred
VCI_RES_ERR	-4	Resource error; during creation of a queue etc. the resource limits (memory, max. number of queues, etc.) has been exceeded; check your error with the implementation overview of your platform
VCI_QUE_ERR	-5	Receive queue overrun: One or more objects couldn't be inserted into the queue and were lost. The last inserted object was marked with the 'Receive-Queue-Overrun' bit.
VCI_TX_ERR	-6	It was not possible to send a message via CAN over a long period (several seconds), which indicates a missing device, missing bus terminator or wrong baud rate; check your CAN-connection and cabling

If a 'CciReqData-Error' is signaled with a VCI_ERR as error string of the Exception handler, this means an error in communication between PC and PC/CAN-interface. Possible errors are given in the following list:

Value	*Significance*
0	Command could not be transmitted to the PC/CAN-interface
1	An Error was returned from the PC/CAN-interface as a response and not an OK
2	The wrong response to the commando was returned
3	While waiting for the response, a Timeout has occurred
4	Response is too short (wrong length)
5	When handing off a command to the PC/CAN-interface, a Timeout has occurred

The errors listed here can normally be traced to installation problems, such as:

- Memory range of the PC/CAN-interface is not displayed correctly in the address space of the PC (error number 0, 1 or 5).
- Interrupt of the PC/CAN-interface is not passed on correctly to the PC or is occupied by other plug-in cards (error number 3).
- Communication to the PC/CAN-interface (e.g. interfaces with LPT-interface) is interrupted.

2.2 Type Definitions of the Call-back Handler

Call-back handlers are functions coded by the user and called up (in this case by the VCI) when certain events occur.

In this case they are used for error display and error handling, processing of interrupt messages or for issuing test or initialization protocols.

In order that the VCI can recognize and carry out these Call-back handlers, these functions must correspond to the set type definitions and introduce them to the VCI via 'VCI2_PrepareBoard'.

If, for example, an interrupt is triggered by a Receive queue, a corresponding function (Call-back handler) must be coded by the user. This function must be coded for each installed PC/CAN-interface which should trigger interrupts.

The user decides whether to use the possibilities of Call-back-handling or to do without and just transfer 'VCI2_PrepareBoard' to a NULL-Pointer instead of to a function pointer.

2.2.1 Receive-Interrupt-Handler

The queue messages (Timeout or "High water mark") received via the interrupts are transferred to this function, provided this was specified via the VCI_ConfigQueue.

This Call-back handler is used for 2 different interrupt mechanisms:

- Transmission of messages (max. 13 messages simultaneously)
- Signal of a Receive queue for the application

In the first case the messages are given to the Interrupt-Callback function by parameter. This mode should be used only at low message rates.

Within the Receive-Interrupt-Callback function you should pay attention to the following points:

- Avoid time consuming calculations because the Interrupt-Thread is blocked while you are in the Callback function and no more messages could be handled during this time.
- Sometimes it could be difficult to access application data within the Receive-Interrupt-Callback function because you are in the context of the Interrupt-Thread.

In the second mechanism the call to the Callback function is only a signal to the application (count = 0) and means that messages are in the receive queue that should be read using the VCI_ReadQueObj function.

This could be used for example to set a worker thread in the running state (by setting an event) which could process the messages.

Type definition: typedef void (***VCI_t_UsrRxIntHdlr**)
(UINT16 que_hdl, UINT16 count, VCI_CAN_OBJ far * p_obj);

Parameter:

que_hdl (in)
Handle of the queue which has triggered the interrupt.

count (in)
Number of messages received.

p_obj (in)
FAR-Pointer on the received message(s) of type VCI_CAN_OBJ.

Return values: none

2.2.2 Exception-Handlers

This function is always called up when an error has occurred in a system function. In this case this error is not only displayed via the Return value, but is also handed on to the Exception handler. Thus the user has two ways to handle errors, whereby the one via the Exception handler provides a clearer program code.

Strings with a more exact error specification are transferred to the Exception handler which can be output in an error window or written in a file.

These Null-terminated strings (without control character) with a max. length of 60 characters state the function name of the function in which the error has occurred and the error is specified more precisely.

For each PC/CAN-interface a separate Exception handler must be coded.

Type definition: typedef void (***VCI_t_UsrExcHdlr**)(VCI_FUNC_NUM func_num,
int err_code, UINT16 ext_err, char * s);

Parameter:

func_num (in)
Type name from the list type VCI_FUNC_NUM, via which the function is specified in which the error has occurred.

err_code (in)
Standard-Error-Codes (VCI_SUPP_ERR, VCI_PARA_ERR, ...) specified via defines.

ext_err (in)
Further error specifications with Standard-Error-Code VCI_ERR (see below).

s (in)
Error string (max. 60 characters) stating the function name and further error specification.

Return values: none

2.2.3 Handler for String Output

For the functions VCI2_TestBoard or VCI2_PrepareBoard it is possible to specify an output function via which a test or initialization protocol can be output.

Null-terminated strings (without control character) with a max length of 60 characters are transferred to this function.

Type definition: typedef void (***VCI_t_PutS**)(char * s);

Parameter:

s (in)
Error string (max. 60 characters) stating function name and further error specification.

Return values: none

2.3 State diagram for Board Initialization

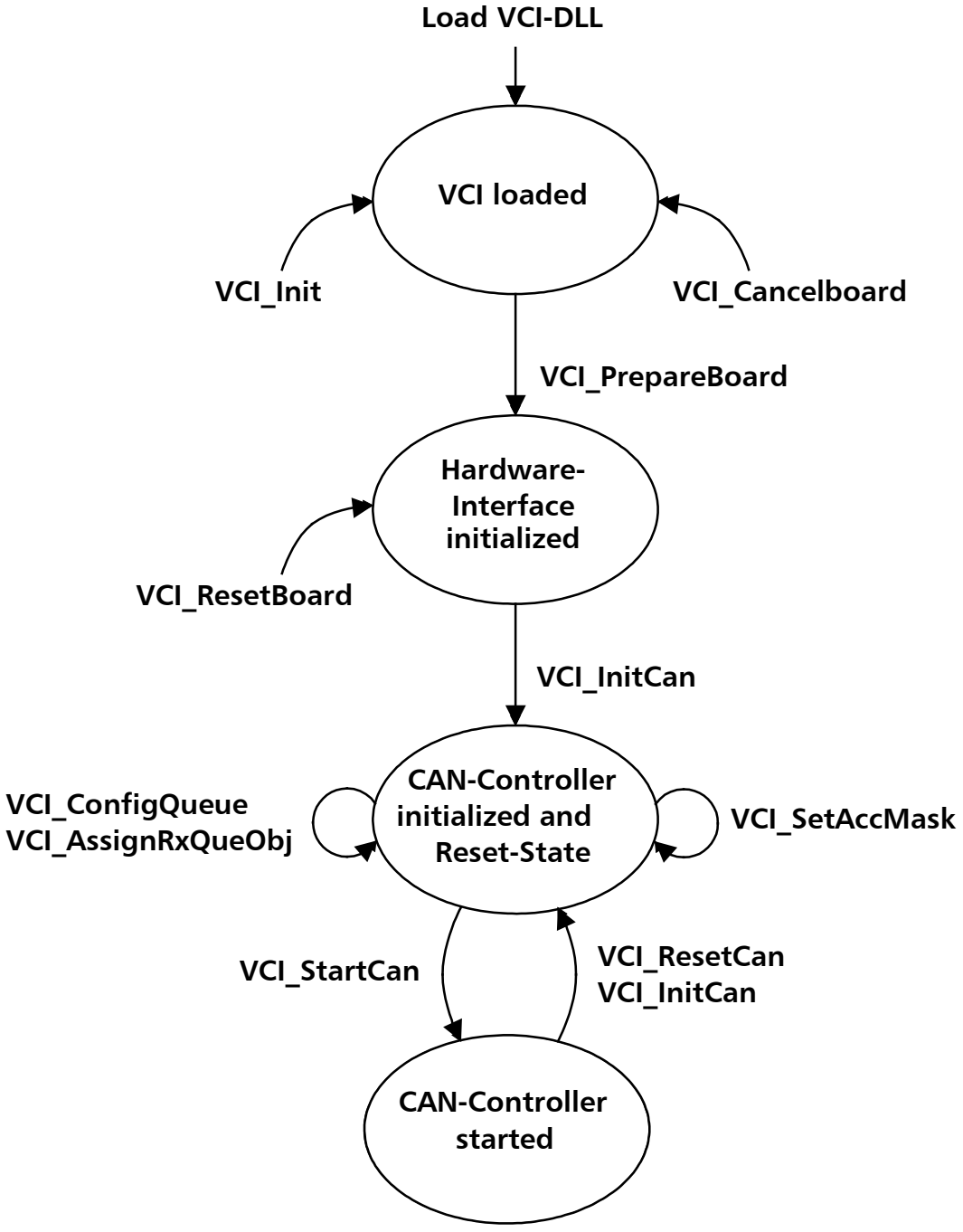

Fig. 2 - 1 State Diagram for Board Initialization

2.4 Table of VCI functions

In this table all necessary VCI functions are sorted by usage. For every VCI function the states are given in which a call is possible. (see state diagram of VCI in chapter 2.3)

VCI – State / VCI functions	VCI-DLL loaded	Board initialized	Controller initialized and in Reset	Queue/Buffer created	Controller started	Remark
VCI_Init	x	x	x	x	x	Should be used only for development! See description.
VCI_Get_LibType	x	x	x	x	x	For compatibility reasons only
VCI_GetBrdTypeInfo	x	x	x	x	x	For compatibility reasons only
VCI_GetBrdNameByType	x	x	x	x	x	For compatibility reasons only
VCI_GetBrdTypeByName	x	x	x	x	x	For compatibility reasons only
Board handling						
VCI_Prepareboard VCI_PrepareboardMsg	x					For compatibility reasons only
VCI2_PrepareBoard VCI2_PrepareBoardMsg	x					
VCI_ResetBoard		x	x	x	x	
VCI_CancelBoard		x	x	x	x	
VCI_ReadBoardInfo		x	x	x	x	
VCI_ReadBoardStatus		x	x	x	x	
VCI_ResetTimeStamp		x	x	x	x	

VCI – State / VCI functions	VCI-DLL loaded	Board initialized	Controller initialized and in Reset	Queue/Buffer created	Controller started	Remark
CAN-controller handling						
VCI_ReadCanInfo		x	x	x	x	
VCI_ReadCanStatus		x	x	x	x	
VCI_InitCan		x	x	x	x	
VCI_SetAccMask			x			To set the acceptance mask the CAN-Controller should be in reset mode!
VCI_ResetCan			x	x	x	
VCI_StartCan			x	x	x	Before starting the CAN controller all necessary initializations like creating of queues/buffers should have been done.
Queue handling						
VCI_ConfigQueue			x			
VCI_AssignRxQueObj				x		
VCI_ReadQueObj				x	x	
VCI_ReadQueStatus				x	x	
VCI_TransmitObj					x	
Buffer handling						
VCI_ConfigBuffer			x			
VCI_ReConfigBuffer				x		
VCI_ReadBufStatus				x	x	
VCI_ReadBufData				x	x	

VCI – State / VCI functions	VCI-DLL loaded	Board initialized	Controller initialized and in Reset	Queue/Buffer created	Controller started	Remark
Remote Buffer handling						
VCI_RequestObj					x	
VCI_UpdateBufObj				x	x	

2.5 Initialization of the VCI

2.5.1 VCI_Init

Function: void VCI_Init(void);

Description: Initialization of the VCI-structures (without Board initialization). Already initialized boards are replaced and logged off (all handles are discarded!).

This function should only be used during the program development of VCI-applications. If during a test a VCI-application has crashed and has left the VCI in an undefined state, the internal data structures can be replaced with this function.

In this way the VCI can be set up again without having to re-boot the computer.

Note: Calling up the function VCI_Init() causes all handles of the VCI-applications currently running to become invalid. This can lead to crashes if at the time of the call-up a VCI-application has already initialized its board and then attempts to access it. **Therefore:** In release versions of a VCI-application, do not execute VCI-Init!

Return value: none

2.6 Functions for VCI Support Information

These functions had formerly been used to get additional information about different PC/CAN interface types. They have been replaced by the functions from the XATxxReg.DLL (see chapter 3).

2.6.1 VCI_Get_LibType

Note: This function is only still included for reasons of compatibility. It should no longer be used.

2.6.2 VCI_GetBrdNameByType

Note: This function is only still included for reasons of compatibility. It should no longer be used.

2.6.3 VCI_GetBrdTypeByName

Note: This function is only still included for reasons of compatibility. It should no longer be used.

2.7 Functions for Board Initialization

2.7.1 VCI_SearchBoard

Note: This function is only still included for reasons of compatibility. It should no longer be used.

2.7.2 VCI_SetDownloadState

Note: This function is only still included for reasons of compatibility. In the current version of the VCI, this function is without any action. It should therefore no longer be used.

2.7.3 VCI2_PrepareBoard and VCI2_PrepareBoardMsg

There are several different versions of the function VCI_PrepareBoard, which differ in the following properties:

- Parameter transfer according to **VCI V1** or conform with **VCI V2**.
  The parameters for specification of the PC/CAN-interface must be known to the user for the **VCI V1** version and explicitly specified.
  For the **VCI V2** version, the parameters for specification of the PC/CAN-interface are acquired via the XAT-registration functions (see Section 3).
- Use of a **Call-back function** or of a **Message handler** for the interrupt-controlled processing of messages received.
  (see Section 2.2 and Section 2.9.1).

Overview of the versions:

Function	*Use*
VCI_PrepareBoard	VCI V1 with Call-back function
VCI2_PrepareBoard	VCI V2 with Call-back function
VCI_PrepareBoardMsg	VCI V1 with Message handler
VCI2_PrepareBoardMsg	VCI V2 with Message handler

Description:
The function requests the use of a PC/CAN-interface from the VCI. This involves resetting the interface, Firmware download and start of the Firmware on intelligent interface as well as registration of the Call-back functions. The CAN-Controllers of the interface are set to Init-mode.

A Handle is returned to the PC/CAN-interface as a Return value, under which the interface can be addressed. Handles are assigned as ascending numbers from zero onwards (0, 1, 2, ...n).

The function must be executed before the interface is accessed. Interfaces already logged in and thus occupied by a program **cannot** be logged in again. (If the PC/CAN-interface is to be used by another application, the interface must first be released by the VCI_CancelBoard.)

The Call-back handlers are also set by VCI2_PrepareBoard.

- Put string for screen issue with PrepareBoard.
- Exception handler for error handling.
- Receive interrupt handler **or** Message handler for Interrupt operation, depending on version.

For this, see type definitions Call-back handler.

2.7.3.1 VCI_PrepareBoard

Note: This function is only still included for reasons of compatibility. It should no longer be used. Please use the VCI2_PrepareBoard function instead.

2.7.3.2 VCI2_PrepareBoard

Function: int **VCI2_PrepareBoard**(VCI_BOARD_TYPE board_type, UINT16 board_index, char* s_addinfo, UINT8 b_addLength, VCI_t_PutS fp_puts, VCI_t_UsrIntHdlr fp_int_hdlr, VCI_t_UsrExHdlr fp_exc_hdlr);

Description: see under 2.7.3

Parameter: **board_type (in)**
An integer value which marks the type of PC/CAN-interface used. Is

only used for checking consistency, as the board is clearly defined by the parameter board index.

board_index (in)
In the parameter board_index a unique index number is transferred, under which the PC/CAN-interface is registered with the system. Valid index numbers can be defined by the application via the registration functions (see Section 3).

s_addinfo (in)
Pointer to a buffer with maximum 256 bytes, which can accept additional information. This additional information is administered by the VCI_V2 and can be altered by the user in the hardware selection dialogue.

If you no longer require this information, set this parameter to zero.

At the moment this parameter is only used for special hardware. Standard IXXAT PC/CAN-interfaces do not use this additional information.

b_addLength (in)
Maximum length of the buffer for the additional information in s_addinfo.

fp_puts (in)
Call-back function for issue of error and status messages for Prepare
(NULL -> no status issue).

fp_int_hdlr (in)
Function pointer to the Routine for processing the messages received.
(NULL -> no interrupt processing)

fp_exc_hdlr (in)
Function pointer to the Exception handler for processing of the error which has occurred
(NULL -> no Exception handler)

Return value: >= 0 -> Board-Handle
< 0 -> VCI-Return codes.

2.7.3.3 VCI_PrepareBoardMsg

Note: This function is only still included for reasons of compatibility. It should no longer be used. Please use the **VCI2_PrepareBoardMsg** function instead.

2.7.3.4 VCI2_PrepareBoardMsg

Function: int **VCI2_PrepareBoardMsg** (VCI_BOARD_TYPE board_type, UINT16 board_index, char *s_addinfo, UINT8 b_addLength, VCI_t_PutS fp_puts, UINT msg_rx_int_hdlr, VCI_t_UsrExcHdlr fp_exc_hdlr, HWND apl_handle);

Description: see under 2.7.3

By means of this function, a Windows Message-Identifier ('msg_int_hdlr') and a Windows-Handle ('apl_handle') are transferred instead of a Call-back-function for the Interrupt processing of the VCI-DLL.

With the Windows Message, the following parameters are also transferred to the application which is referenced by the Windows Handle.

WPARAM count
(Number of CAN-messages transferred together with the Message)

LPARAM Pointer to transferred data
1. BYTE QueRef
(indicates the queue which has triggered the interrupt)
2..n.BYTE CAN_OBJ
(the number given by ´count´ of the CAN_OBj of type VCI_CAN_OBJ)

Example:

```
void Int_Msg_handler(UINT16 WPARAM,UINT32 LPARAM)
{
  // number of messages is in wparam
  UINT16  count = WPARAM;

  // get queuehandle
  UINT8  QueRef = *((UINT8*)LPARAM)

  // get pointer to first can message
  VCI_CAN OBJ* pObj =
    (VCI_CAN_OBJ*)(((UINT8*)LPARAM)+1);

  // copy messages from queue to destination
  VCI_CAN OBJ  DestObj[20];
  memcpy(DestObj, pObj,
          count*sizeof(VCI_CAN OBJ));
}
```

Parameter: **board_type (in)**
an integer value which marks the type of the PC/CAN-interface used. Is only used for checking consistency, since the board is clearly defined by the board index parameter.

board_index (in)
In the parameter board_index, a unique index number is transferred, under which the PC/CAN-interface is registered with the system. Valid index numbers can be determined by the application via the registration functions (see Section 3).

s_addinfo (in)
Pointer to a buffer with maximum 256 bytes which can accept additional information. This additional information is administered by the VCI_V2 and can be altered by the user in the hardware selection dialogue.

If you do not require this information, set this parameter to zero.

At the moment this parameter is only used for special hardware. Standard IXXAT PC/CAN-interfaces do not use this additional information.

b_addLength (in)
Maximum length of the buffer for the additional information in s_addinfo.

fp_puts (in)
Call-back function for the issue of error and status messages with Prepare
(NULL -> no status issue).

msg_rx_int_hdlr (in)
ID of the Windows-message with which the reception of CAN-messages is to be signaled to the application. Typically one user-defined message (WM_USER + Offset) is determined and transferred at this point.

fp_exc_hdlr (in)
Function pointer to the Exception handler for processing of the error which has occurred
(NULL -> no Exception handler)

apl_handle (in)
Handle of the application to which the agreed Windows-message is to be sent.

Return value: >= 0 -> Board-Handle
< 0 -> VCI-Return codes.

2.7.4 VCI_PrepareBoardVisBas

Unfortunately, the possibility to use Call-backs was again limited from Microsoft Visual Basic 6.0. Therefore you should not use this function under Visual Basic.

2.7.5 VCI_CancelBoard

Function: int **VCI_CancelBoard**(UINT16 board_hdl);

Description: Cancel board with VCI. This involves resetting the interface and the CAN Controller as well as uninstalling the interrupts used. The board handle thus becomes free again.

Parameter: **board_hdl (in)**
Handle of a board logged in previously.

Return value: VCI-Return codes.

2.7.6 VCI_TestBoard

Note: This function is only still included for reasons of compatibility. It should no longer be used.

2.7.7 VCI_ReadBoardInfo

Function: int **VCI_ReadBoardInfo**(UINT16 board_hdl , VCI_BOARD_INFO* p_info);

Description: Reading of the board information according to VCI_BOARD_INFO:

VCI_BOARD_INFO	*Description*
~.hw_version	Hardware version as HEX-value (e.g.: 0x0100 for V1.00)
~.fw_version	Firmware version as HEX-value
~.dd_version	Device-Driver version as HEX-value (only for PC card)
~.sw_version	Version number of PC-software as HEX-value
~.can_num	Number of the CAN-Controllers supported by the board
~.time_stamp_res	Smallest possible Time Stamp resolution in 100nsec
~.timeout_res	Smallest possible Timeout resolution for the Receive queues
~.mem_pool_size	Size of the memory pool for the creation of queues and buffers (In the VCI_V2 this value is set to 0, because it is not necessary any more)
~.irq_num	Interrupt number for the communication with the PC/CAN-interface
~.board_seg	Set Board-Address/Segment/Port number
~.serial_num	16-character-string with the serial number of the board
~str_hw_type	Null-terminated string with the hardware type

The time information of the Time Stamp or Timeout resolution enables correct setting of these times.

The function execution is optional and is only intended for specification of the PC/CAN-interface.

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

p_info (out)
Pointer to the info data.

Return value: VCI-Return codes.

2.7.8 VCI_ReadBoardStatus

Function: int **VCI_ReadBoardStatus**(UINT16 board_hdl, VCI_BRD_STS * p_sts);

Description: Reading of the board information according to VCI_BRD_STS:

VCI_BRD_STS	*Description*
~.sts	Bit-coded information on board status

Dependent on the IXXAT PC/CAN-Interface you are working on the reflectance of the given board status after calls to functions like VCI_StartCan and VCI_ResetCan can be delayed up to 100ms. During this time VCI_ReadBoardStatus may retrieve an obsolete status. Therefore use this function for status visualization in your application and not to verify the accomplishment of called VCI control functions.

The individual bits of ~.sts have the following significance:

Bit 0: RxQueue Overrun; an overrun has occurred in a configured Receive queue (queue was already full and a further message could not be entered.) Further information is given by VCI_ReadQueStatus and VCI_ReadQueObj.

Bit 4: CAN0-Running

Bit 5: CAN1-Running

Bit 6: CAN2-Running

Bit 7: CAN3-Running

Status bit for the CAN-Controllers of the boards (up to a maximum of 4 CAN-Controllers per board are supported by the VCI).

CAN-Controllers which have been initialized started and which are working correctly are set to ´1´. If the CAN-Controller is in Bus-Off-status or Init-mode, or if a CAN-Data-Overrun or Remote queue-Overrun occurred, the bit is set to ´0´. The exact cause must then be determined via VCI_ReadCanStatus.

An overview of the status of the CAN-Controller can thus be obtained very quickly without having to read the CAN-State.

VCI_BRD_STS	*Description*
~.cpu_load	Average CPU-load in % (0-100)

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

p_sts (out)
Pointer to the status to be read.

Return value: VCI-Return codes.

2.7.9 VCI_ResetBoard

Function: int **VCI_ResetBoard**(UINT16 board_hdl);

Description: Reset of the interface (software and hardware). The board thus remains logged in, but the communication is interrupted. After this the board and the CAN-Controllers are initialized again.

Parameter: **board_hdl (in)**
Handle of the boards already logged in.

Return value: VCI-Return codes.

2.8 Functions for CAN-Controller handling

2.8.1 VCI_ReadCanInfo

Function: int **VCI_ReadCanInfo**(UINT16 board_hdl, UINT8 can_num , VCI_CAN_INFO * p_info);

Description: Reading of the type of CAN-Controller and of the set parameters according to VCI_CAN_INFO:

VCI_CAN_INFO	*Description*
~.can_type	type of CAN-Controller according to VCI_CAN_TYPE
~.bt0	set value for the Bit Timing Register 0
~.bt1	set value for the Bit Timing Register 1
~.acc_code	set value for the Acceptance-Code-Register
~.acc_mask	set value for the Acceptance-Mask-Register

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

can_num (in)
number of CAN-Controllers (0..3).

p_info (out)
Pointer to the info data.

Return value: VCI-Return codes.

2.8.2 VCI_ReadCanStatus

Function: int **VCI_ReadCanStatus**(UINT16 board_hdl, UINT8 can_num , VCI_CAN_STS * p_sts);

Description: Reading of the status information of the given CAN-Controllers and of the assigned software according to VCI_CAN_STS:

VCI_CAN_STS	*Description*
~.sts	Bit-coded information to the CAN-Status (1 = true):

Dependent on the IXXAT PC/CAN-Interface you are working on the reflectance of the given CAN status after calls to functions like VCI_StartCan and VCI_ResetCan can be delayed up to 100ms. During this time VCI_ReadCanStatus may retrieve an obsolete status. Therefore use this function for status visualization in your application and not to verify the accomplishment of called VCI control functions.

The individual bits of ~.sts have the following significance:

Bit 0: not used,

Bit 1: not used,

Bit 2: RemoteQueueOverrun – An overrun has occurred in the internal queue for the processing of the Remote requests,

Bit 3: CAN-TX-Pending – A send process is running. If this state continues without data being sent again, the CAN-Controller cannot deposit the data (line breakage or similar)

Bit 4: CAN-Init-Mode - CAN is in Initialization status and can be set to running mode via VCI_StartCan,

Bit 5: CAN-Data-Overrun – An overrun of CAN-messages has occurred in the CAN-Controller (or in the software in proximity to the CAN-Controller).

Bit 6: CAN-Error-Warning-Level - The CAN-Controller has reached the Error-Warning level due to faults on the bus.

Bit 7: CAN-Bus-Off-Status, - The CAN-Controller has switched itself off from the bus due to bus faults.

VCI_CAN_STS	*Description*
~.bus_load	Busload in percent. This feature is only supported for active CAN interfaces. Stuff bits are not considered here.

The bits 4 - 7 are delivered directly from the CAN-Controllers. (For further information on these bits, please read the data sheets on the CAN-Controllers Phillips 82C200 or Intel 82527).
If an error has occurred in the CAN-Controller (bits 2,5 and 7), this state can only be exited via the function VCI_ResetCan.

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

can_num (in)
Number of CAN-Controllers (0..n).

p_sts (out)
Pointer to the status data.

Return value: VCI-Return codes.

2.8.3 VCI_InitCan

Function: int **VCI_InitCan**(UINT16 board_hdl, UINT8 can_num, UINT8 bt0, UINT8 bt1, UINT8 mode);

Description: Initialization of the Timing-Register. The values correspond to those of Philips SJA 1000. For other Controllers, the values are converted accordingly. For this purpose, the given CAN-Controller is set to the state of Init-mode and must then be re-started via VCI_StartCan.

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

can_num (in)
number of CAN-Controllers (0..3).

bt0 (in)
value for the Timing-Register 0.

bt1 (in)
value for the Timing-Register 1.

Transmission rate in kBit/s	*bt0*	*bt1*
1000	00h	14h
500	00h	1Ch
250	01h	1Ch
125	03h	1Ch
100	04h	1Ch
50	09h	1Ch
20	18h	1Ch
10	31h	1Ch

mode (in)

VCI_11B:

Standard CAN frame format with 11Bit identifier.

VCI_29B:

Extended CAN frame format with 29Bit identifier.

VCI_LOW_SPEED:

Low speed bus connector (if provided by the hardware).

VCI_TX_ECHO:

Self reception.

VCI_TX_PASSIV:

Passive mode of CAN controller (“Listen only” mode).

VCI_ERRFRM_DET:

Error frames detection.

Except of the VCI_11B and VCI_29B other settings can be combined.

Example for standard mode with self reception and error frame detection:

VCI_InitCan(BoardHdl, CAN_NUM, VCI_125KB, VCI_11B | VCI_TX_ECHO | VCI_ERRFRM_DET);

Return value: VCI-Return codes.

2.8.4 VCI_SetAccMask

Function: int **VCI_SetAccMask**(UINT16 board_hdl, UINT8 can_num, UINT32 acc_code, UINT32 acc_mask);

Description: Setting of the Acceptance-Mask-Register of the CAN-Controllers for a global message-filtering in 11-bit- or 29-bit operation. (this Controller function may be replaced by software). The filter works via all identifier-bits. It is fully opened (0x0UL, 0x0UL) as long as this function is not executed. For this purpose, the given CAN-Controller is set to the Init-mode state and must then be re-started via VCI_StartCan.

With the variables acc_code and acc_mask, individual CAN-IDs or whole ID-groups can be defined.

Examples:

- Only the CAN-ID 100 is to be received:
  acc_code = 100 and acc_mask = 0xffffffff
  0xffffffff -> all Bits of acc_code are relevant
- The CAN-IDs 100-103 are to be received:
  acc_code = 100 and acc_mask = 0xfffffffc
  0xfffffffc -> all bits of acc_code are relevant except the lower two (00,01,10,11).

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

can_num (in)
number of CAN-Controllers (0..3).

acc_code (in)
value for the Acceptance-Code-Register

acc_mask (in)
value for the Acceptance-Mask-Register
(0 - don't care; 1 - relevant)

Return value: VCI-Return codes.

2.8.5 VCI_ResetCan

Function: int **VCI_ResetCan**(UINT16 board_hdl, UINT8 can_num);

Description: Reset of the CAN-Controllers and thus stop of communication via the given CAN-Controller. In addition, the status register of the CAN-Controllers is deleted and the queues and buffers allocated to this CAN-Controller are re-initialized via this function.
The CAN-Controller does not lose its configuration, but can be put back into operation via VCI_StartCan.

Parameter: **board_hdl (in)**
Handle of the board logged in previously.
can_num (in)
number of CAN-Controllers (0..3).

Return value: VCI-Return codes.

2.8.6 VCI_StartCan

Function: int **VCI_StartCan**(UINT16 board_hdl, UINT8 can_num);

Description: Start of the given CAN-Controllers

Parameter: **board_hdl (in)**
Handle of the board logged in previously.
can_num (in)
number of CAN-Controllers (0..3).

Return value: VCI-Return codes.

2.9 Functions for the Queue and Buffer Configuration

2.9.1 VCI_ConfigQueue

Function: UINT16 **VCI_ConfigQueue**(UINT16 board_hdl, UINT8 can_num, UINT8 que_type, UINT16 que_size, UINT16 int_limit, UINT16 int_time, UINT16 ts_res, UINT16 * p_que_hdl);

Description: Creation of a Transmit or Receive queue. As a result, a handle is returned to the queue under which the queue can be addressed.

Then, in the case of a Receive queue, all required CAN-messages are signaled with VCI_AssignRxQueObj.

For Receive queues there are 3 different ways of processing queue messages:

- Creation of a queue for polling operation via VCI_ReadQueObj. For this, the parameters int_limit and int_time are set to zero.
- Creation of a queue for the interrupt processing of alarm messages. For this, int_limit is set to 1 (max. 13 messages). The message or messages are transmitted directly to the Interrupt-Callback handler (transmitted in 'VCI2_PrepareBoard') which is called when the number of messages in the receive queue reaches or exceeds int_limit. It's also called if one or more messages are received but int_time is expired and int_limit is not reached.

'VCI_ReadQueObj()' cannot be used here.
This operating mode guarantees very short reaction times, but is not suitable for larger data rates due to relatively low effectiveness. The messages must be copied via the pointer cosigned to the Interrupt-Callback handler.

- Creation of a queue with Event operation. The Interrupt signal (Interrupt-Callback handler is called) can be used here to trigger the corresponding task under a Multitasking environment to poll the received message via VCI_ReadQueObj (The Interrupt-Callback handler itself does not provide the received message here).
  An interrupt is signaled when the number of message in the receive queue reaches or exceeds int_limit. It's also called if one or more messages are received but int_time is expired and int_limit is not reached.
  For configuring the receive queue with event operation the int_limit has to be set to a value bigger than 13.

This operating mode is the most effective and is therefore recommended for the reception of larger amounts of data with higher data rates.

The Call-back handler is described in more detail in chapter 2.2.1.

The user decides whether to use the possibilities of Call-back-handling or to do without and just transfer 'VCI2_PrepareBoard' to a NULL-Pointer instead of to a function pointer.

With the parameters for the time information, the resolution supported by the interface must be observed.

The CAN-Controller, which is assigned to the queue, must be in Init-mode for the configuration of the queues!

Parameter:

board_hdl (in)
Handle of the board logged in previously.

can_num (in)
CAN-number (0..3).

que_type (in)
Queue type (VCI_TX_QUE, VCI_RX_QUE).

que_size (in)
Size of the queue in CAN- messages (must be >= 20!)

int_limit (in)
Number of CAN-messages after which an Interrupt is triggered.

0 =	Do not trigger an Interrupt.
<=13	The messages received are passed up immediately with the Interrupt
>13	The messages received must be read with the aid of the function 'VCI_ReadQueObj()' .

For a transmit queue this parameter can be set to zero.

int_time (in)
Time in ms after which a receive queue interrupt is triggered, if 'int_limit' is not reached. According to the size of 'int_limit', the CAN- messages are transmitted directly with the Interrupt or have to be polled. If for a receive queue int_time is set to zero it's internally set to 500ms to prevent consuming much of CPU time.

For configuration of a transmit queue this parameter is not considered and therefore it can be set to zero.

ts_res (in)
Required resolution in μs of the message-Time-Stamps for one Receive queue. For configuration of a transmit queue this parameter is not considered and therefore it can be set to zero.

p_que_hdl (out)
Handle of the queue.

Return value: VCI-Return codes.

Example: Polling of a receive queue

```
VCI_CAN_OBJ sObj;
INT32       lRes;
UINT16      hRxQue ;

int main(int argc, char* argv[])
{
  ...
  lRes = VCI_ConfigQueue( hBrd
                        , 0           // CAN 1
                        , VCI_RX_QUE  // receive queue
                        , 100         // queue size = 100 can objects
                        , 0           // no limit = polling mode
                        , 0           // timeout not relevant
                        , 100         // timestamp res. 100µsec
                        , &hRxQue);
  if ( VCI_OK == lRes )
  {
    ...
    while ( !_kbhit() )
    {
      lRes = VCI_ReadQueObj(hBrd, hRxQue, 1, &sObj);
      if ( 0 < lRes )
      {
        printf("Id 0x%X received\n", sObj.id);
      }
    }
    ...
  }
}
```

Example: Interrupt processing of a receive queue

```
INT32  lRes;
UINT16 hRxQue;

void VCI_CALLBACKATTR ReceiveCallback( UINT16            que_hdl
                                     , UINT16            count
                                     , VCI_CAN_OBJ FAR * p_obj)
{
  for (UINT16 i = 0; i < count; i++)
  {
    printf("Id 0x%X received\n", p_obj[i].id);
  }
}


int main(int argc, char* argv[])
{
  XAT_BoardCFG sConfig;
  ...
  INT32 hBrd = VCI2_PrepareBoard( sConfig.board_type
                                , sConfig.board_no
                                , sConfig.sz_CardAddString
                                , strlen(sConfig.sz_CardAddString)
                                , NULL
                                , ReceiveCallback
                                , NULL );
  if ( 0 <= hBrd )
  {
    ...
    lRes = VCI_ConfigQueue( hBrd
                          , 0           // CAN 1
                          , VCI_RX_QUE  // receive queue
                          , 100         // queue size = 100 can objects
                          , 1           // interrupt mode
                          , 100         // timeout 100msec
                          , 100         // timestamp res. 100µsec
                          , &hRxQue);
    if ( VCI_OK == lRes )
    {
      ...
    }
  }
}
```

Example: Receive queue in event mode.

```
#define NUMOFOBJ 50
INT32  lRes;
UINT16 hRxQue;
HANDLE hRxEvent;

void VCI_CALLBACKATTR ReceiveCallback( UINT16 que_hdl
        , UINT16 count , VCI_CAN_OBJ FAR * p_obj)
{
  SetEvent(hRxEvent);
}

int main(int argc, char* argv[])
{
  XAT_BoardCFG sConfig;
  ...
  INT32 hBrd = VCI2_PrepareBoard( sConfig.board_type
                                , sConfig.board_no
                                , sConfig.sz_CardAddString
                                , strlen(sConfig.sz_CardAddString)
                                , NULL
                                , ReceiveCallback
                                , NULL );
  if ( 0 <= hBrd )
  {
    ...
    lRes = VCI_ConfigQueue( hBrd
                          , 0           // CAN 1
                          , VCI_RX_QUE  // receive queue
                          , 100         // queue size = 100 can objects
                          , 14          // event mode
                          , 100         // timeout 100msec
                          , 100         // timestamp res. 100µsec
                          , &hRxQue);
    if ( VCI_OK == lRes )
    {
      DWORD       dwWaitRes;
      VCI_CAN_OBJ asObj[NUMOFOBJ];
      INT32       lReadCount;
      ...
      while (!_kbhit())
      {
        dwWaitRes = WaitForSingleObject(hRxEvent, 1000);
        if (WAIT_OBJECT_0 == dwWaitRes)
        {
          do
          {
            lReadCount = VCI_ReadQueObj(hBrd, hRxQue, NUMOFOBJ, &asObj);
            for (INT32 i = 0; i<lReadCount; i++)
            {
              printf("Id 0x%X received\n", sObj[i].id);
            }
          } while( 0 < lReadCount )
        }
} } } }
```

2.9.2 VCI_AssignRxQueObj

Function: int **VCI_AssignRxQueObj**(UINT16 board_hdl, UINT16 que_hdl, UINT8 mode, UINT32 id, UINT32 mask);

Description: Assignment / blocking of messages to the given Receive queue. Identifier groups are directly definable via the mask.

Attention: In 29-bit-operation it is not possible to define an unlimited number of identifiers. Depending on hardware, different filter mechanisms are used. Therefore the number of the definable filters is limited.

The use of 'id' and 'mask' is similarly explained in 'VCI_SetAccMask'.

The CAN-Controller to be assigned to the queue must be in Init-mode for the configuration of the queue!!

Parameter:

board_hdl (in)
Handle of the boards logged in previously.

que_hdl (in)
Queue-handle.

mode (in)
Release/blocking of the message(s)
(VCI_ACCEPT, VCI_REJECT).

id (in)
Identifier of the message(s).

mask (in)
Mask for defining the relevant Identifier bits. (0 - don't care; 1 - relevant)

Return value: VCI-Return codes.

2.9.3 VCI_ResetTimeStamp

Function: int **VCI_ResetTimeStamp**(UINT16 board_hdl);

Description: Reset of the timers for the Time-Stamps of the Receive queues.

Parameter:

board_hdl (in)
Handle of the boards logged in previously.

Return value: VCI-Return codes.

2.9.4 VCI_ConfigBuffer

Function: int **VCI_ConfigBuffer**(UINT16 board_hdl, UINT8 can_num, UINT8 type, UINT32 id, UINT16 * p_buf_hdl);

Description: Creation of a Receive or Remote buffer. Access to this buffer is via the returned handle. Handles are assigned as ascending numbers from zero onwards (0, 1, 2, ...n).

Parameter:

board_hdl (in)
Handle of the boards logged in previously.

can_num (in)
CAN-number (0..n).

type (in)
Receive or Remote buffer
(VCI_RX_BUF, VCI_RMT_BUF).

id (in)
Identifier.

p_buf_hdl (out)
Handle to the buffer.

Return value: VCI-Return codes.

2.9.5 VCI_ReConfigBuffer

Function: int **VCI_ReConfigBuffer**(UINT16 board_hdl, UINT16 buf_hdl, UINT8 type, UINT32 id);

Description: Alteration of the identifiers of a Receive or Remote buffer. Access to this buffer is via the Handle.

Parameter:

board_hdl (in)
Handle of the boards logged in previously.

buf_hdl (in)
Buffer-Handle.

type (in)
Receive or Remote buffer
(VCI_RX_BUF, VCI_RMT_BUF).

id (in)
Identifier.

Return value: VCI-Return codes.

2.10 Receiving Messages

2.10.1 VCI_ReadQueStatus

Function: int **VCI_ReadQueStatus**(UINT16 board_hdl, UINT16 que_hdl);

Description: Reading of the status of the given queue.

Parameter: **board_hdl (in)**
Handle of the boards logged in previously.

que_hdl (in)
Handle of the queue.

Return value:

>0	Number of queue entries.
=0	Queue empty (VCI_QUE_EMPTY).
<0	VCI-Return codes.

2.10.2 VCI_ReadQueObj

Function: int **VCI_ReadQueObj**(UINT16 board_hdl, UINT16 que_hdl, UINT16 count, VCI_CAN_OBJ * p_obj);

Description: Reads the first entry/entries of a Receive queue. The number of the entries to be read is given via 'count'. However, only as many entries are read as are in the queue or are supported by the interface per read process. This means that the queue must be read until VCI_QUE_EMPTY is returned as Return value.
If Bit7 (0x80 = Queue-Overrun) of the status byte in the message is set, no further message could be entered in the Receive queue after this message as it is already full.

This means that messages has been lost.

Parameter: **board_hdl (in)**
Handle of the boards logged in previously.

que_hdl (in)
Handle of the queue.

count (in)
Maximum number of messages to be read (max = 13)

p_obj (out)
Pointer to the message(s) to be read.

Return value:

>0	Number of the queue entries read.
=0	Queue empty (VCI_QUE_EMPTY).
<0	VCI-Return codes.

2.10.3 VCI_ReadBufStatus

Function: int **VCI_ReadBufStatus**(UINT16 board_hdl, UINT16 buf_hdl);

Description: Reading of the buffer status without altering it. The buffer status shows the number of Receive processes on this buffer since the last Read process.

Parameter: **board_hdl (in)**
Handle of the boards logged in previously.

buf_hdl (in)
Buffer-handle.

Return value:

=0	VCI_OLD	no new data received.
>0		Number, how often the data were received after the last Read process.
<0		VCI-Return codes.

2.10.4 VCI_ReadBufData

Function: int **VCI_ReadBufData**(UINT16 board_hdl,
UINT16 buf_hdl, UINT8 * p_data, UINT8 * p_len);

Description: Reading of the buffer data and return of the buffer status. The status (number of the Receive processes since the last reading) is delivered as Return. If this value is added up, the absolute number of Receive processes since program start is obtained.

Parameter: **board_hdl (in)**
Handle of the boards logged in previously.

buf_hdl (in)
Buffer-Handle.

p_data (out)
Pointer to the data to be read.

p_len (out)
Pointer to the number of data bytes.

Return value:

= 0	VCI_OLD	no new data received.
>0		Number, how often the data were received after the last Read process.
<0		VCI-Return codes.

2.11 Sending Messages

2.11.1 VCI_TransmitObj

Function: int **VCI_TransmitObj**(UINT16 board_hdl,
UINT16 que_hdl, UINT32 id, UINT8 len,
UINT8 * p_data);

Description: Sending of a CAN-message via the given Send queue. If VCI_QUE_FULL is returned, the given Send queue is full at the moment and the Send request must be repeated (later). If VCI_TX_ERR is returned, the CAN-Controller is not able to send messages.

Possible Reasons are missing devices, missing bus terminators or wrong baud rate. Please check your CAN connector and cabling.

For invalid parameter values as an invalid identifier (>7FFh for 11-bit mode; >1FFFFFFFh for 29-bit mode) the return value is VCI_PARA_ERR. The VCI exception callback gives a detailed error description.

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

que_hdl (in)
Queue handle.

id (in)
Identifier of the Send message.

len (in)
Number of the data bytes.

p_data (in)
Pointer to the Send data.

Return value: VCI-Return codes.

2.11.2 VCI_RequestObj

Function: int **VCI_RequestObj**(UINT16 board_hdl,
UINT16 que_hdl, UINT32 id, UINT8 len);

Description: Sending of a request message (Remote frame) via the given Send queue. If VCI_QUE_FULL is returned, the given Send queue is full at the moment and the Send request must be repeated (later). If VCI_TX_ERR is returned, the CAN-Controller is not able to send messages.

Possible Reasons are missing devices, missing bus terminators or wrong baud rate. Please check your CAN connector and cabling.

For invalid parameter values as an invalid identifier (>7FFh for 11-bit mode; >1FFFFFFFh for 29-bit mode) the return value is VCI_PARA_ERR. The VCI exception callback gives a detailed error description.

If the CAN controller does not support remote frames VCI_RequestObj returns VCI_SUPP_ERR.

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

que_hdl (in)
Queue handle.

id (in)
Identifier of the Send message.

len (in)
Number of data bytes.

Return value: VCI-Return codes.

2.11.3 VCI_UpdateBufObj

Function: int **VCI_UpdateBufObj**(UINT16 board_hdl, UINT16 buf_hdl, UINT8 len, UINT8 * p_data);

Description: Update of data in a Remote buffer, which can be requested via the CAN-network by another CAN-Controller.

Parameter: **board_hdl (in)**
Handle of the board logged in previously.

buf_hdl (in)
Buffer handle.

len (in)
Number of the data bytes.

p_data (in)
Pointer to the data.

Return value: VCI_OK, VCI_QUE_ERR, VCI_HWSW_ERR, VCI_SUPP_ERR, VCI_PARA_ERR.

2.12 Data Types Used

For the exact specification of the data types used, please see File VCI.H. In the following section, the most important structures are explained.

2.12.1 VCI-CAN-Object

Sending of CAN-messages via Transmit queues and reading of CAN-messages from Receive queues is carried out via the data type VCI_CAN_OBJ:

VCI_CAN_OBJ	*Description*
~.time_stamp	Reception time stamp for Receive queue-messages. The resolution is prescribed the function VCI_ConfigQueue. Please note that independently of formatting of the time stamp after (2 ^ 32 * Timestamp resolution) an overrun occurs (> 12 hours). The time stamp can be reset via the function VCI_ResetTimeStamp.
~.id	11/29-bit-identifier of the CAN-message (always right-justified)
~.len	Number of data bytes of the CAN-message(0-8 bytes)
~.rtr	1 = Remote-Request (data request message; the following data bytes therefore have no significance 0 = Data frame (Data)
~.res	Not used
~.a_data[8]	8-byte-array for the data bytes of the message
~.sts	Status of the message: 0 = OK; 0x80 = Queue-Overrun (After this message no further could be entered in the Receive queue -> data loss!)

2.12.2 VCI-Board Information

Reading of the board information occurs via the structure VCI_BOARD_INFO:

VCI_BOARD_INFO	*Description*
~.hw_version	Hardware version as HEX-value (z. B: 0x0100 for V1.00)
~.fw_version	Firmware version as HEX-value
~.dd_version	Device-Driver version as HEX-value (only for PCMCIA-cards)
~.sw_version	Version number of the PC-software as HEX-value
~.can_num	Number of CAN-Controllers supported by the board
~.time_stamp_res	Smallest possible Time Stamp resolution in 100 nsec
~.timeout_res	Smallest possible Timeout resolution for the Receive queues
~.mem_pool_size	Size of the memory pools creating queues and buffers
~.irq_num	Interrupt number for communication with the PC/CAN-interface

~.board_seg	Set Board-Address/Segment/Port number
~.serial_num	16-character-string with the serial number of the board
~.str_hw_type	Null-terminated string with the hardware type

2.12.3 VCI-Board-Status

Reading of the board status occurs via the structure VCI_BRD_STS:

VCI_BRD_STS	*Description*
~.sts	Bit-coded information on board status: Bit 0: RxQueue Overrun; an overrun has occurred in a Receive queue (queue was already full and a further message could not be entered.) Further information via VCI_ReadQueStatus and VCI_ReadQueObj. Bit 4: CAN0-Running Bit 5: CAN1-Running Bit 6: CAN2-Running Bit 7: CAN3-Running Status-bit for the CAN-Controller of the board (a maximum of 4 CAN-Controllers per board are supported). CAN-Controllers which have been initialized started and which are working correctly are set to ´1´. If the CAN-Controller is in Bus-Off-status or Init-mode, or if a CAN-Data-Overrun or Remote-Queue-Overrun occurred, the bit is set to ´0´. The exact cause must then be determined via VCI_ReadCanStatus. In this way an overview of the state of the CAN-Controller can be obtained very quickly without having to read CAN-State.
~.cpu_load	average CPU-load in % (0-100)

2.12.4 VCI-CAN-Information

Reading of the CAN-information occurs via the structure VCI_CAN_INFO:

VCI_CAN_INFO	*Description*
~.can_type	Type of the CAN-Controllers according to VCI_CAN_TYPE
~.bt0	Set value for the Bit Timing Register 0
~.bt1	Set value for the Bit Timing Register 1
~.acc_code	Set value for the Acceptance-Code-Register
~.acc_mask	Set value for the Acceptance-Mask-Register

2.12.5 VCI-CAN-Status

Reading of the CAN-status occurs via the structure VCI_CAN_STS:

VCI_CAN_STS	*Description*
~.sts	Bit-coded information on the CAN-status (1 = true): Bit 0: not used, Bit 1: not used, Bit 2: Remote queue overrun – An overrun has occurred in the internal queue for the processing of the Remote requests, Bit 3: CAN-TX-Pending – A send process is running. If this state continues without data being sent again, the CAN-Controller cannot deposit the data (missing device), Bit 4: CAN-Init-Mode - CAN is in initialization state and can be set to Running mode via VCI_StartCan, Bit 5: CAN-Data-Overrun – An overrun of CAN-messages has occurred in the CAN-Controller (or in the software in proximity to the CAN- Controller), Bit 6: CAN-Error-Warning-Level - The CAN-Controller has reached the Error-Warning-level due to many faults on the bus Bit 7: CAN-Bus-Off-Status, - The CAN-Controller has switched off completely from the bus due to bus faults. The bits 4 - 7 are delivered directly from the CAN-Controllers. (For further information on these bits, please read the data sheets of the CAN-Controllers Phillips 82C200 or Intel 82527). If an error has occurred in the CAN-Controller (bits 2,5 and 7), this state can only be exited via the function VCI_ResetCan.
~.bus_load	Bus load in percent. This feature is only supported by active CAN interfaces.

3 Registration Functions (XAT*xx*Reg.DLL)

The VCI_V2 introduced the possibility to register PC/CAN-interfaces which can be addressed via the VCI, under a **unique index number** in the system.

The VCI_V2 now provides an interface with the XAT*xx*Reg.DLL (*xx* stands for version number, e.g. 10) in order to access this registration information.

The interface to this DLL is contained in the Header-file XAT*xx*Reg.h with the same name. For integration of the XAT*xx*Reg.DLL, the same notes apply as for the VCI-DLL in Section 4.

In the following, only those functions of the XAT*xx*Reg.DLL are described which are necessary for a VCI-application in order to access information on the PC/CAN-interfaces registered in the system.

The functions provided cover the following areas:

- Listing (enumeration) of all registered PC/CAN-interfaces and the assigned parameters
- Search for a certain PC/CAN-interface
- Call-up of a hardware-selection dialogue
- Reading of the configuration of a PC/CAN-interface
- Selection/querying of a system-wide default-PC/CAN-interface

3.1 Type Definitions of the Call-back Handler

The XATxxReg.DLL uses Call-back-mechanism in order to read out all available information on registered PC/CAN-interfaces.

3.1.1 Call-back to list the registered PC/CAN-interfaces

The registered PC/CAN-interfaces are transferred to this function after call-up of the function XAT_EnumHwEntry.

Type definition: typedef short (XATREG_CALLBACKATTR * **ENUM_CALLBACK**)
(int i_index, int hw_key, char * name, char * value, char * valuehex, void* vp_context);

Parameter: **i_index (in)**
Type of entry.

- 0 -> Hardware entry
- 1 -> Hardware parameter

hw_key (in)
Unique index number, under which the PC/CAN-interface is registered with the system.

name (in)
For Hardware parameter:
Name of the entry

value (in)
For Hardware-parameter:
Value of the entry

valuehex (in)
For Hardware-parameter:
Hex-value of the entry

vp_context (in)
Void* to the context , which was transferred in the function XAT_EnumHwEntry.

Return value: none

3.2 Function Definitions

3.2.1 XAT_SelectHardware

Function: int XATREG_CALLATTR **XAT_SelectHardware**
(HWND hwndOwner, XAT_BoardCFG* pConfig);

Description: Shows a dialogue for selection of the PC/CAN-interfaces. The configuration selected by the user is deposited in a structure indicated by the parameter pConfig.

Parameter: **hwndOwner (in)**
Window-Handle of the parent window of the dialogue.
Normally, the Handle of the main window of the application is transmitted here.

pConfig (in/out)
Pointer to a data structure in which the board configuration selected by the user is written.

Return value:
0 -> User pressed CANCEL button
1 -> User pressed OK-Button
-1 -> Error (use GetLastError()-function to retrieve extended error info)

Example:

```
XAT_BoardCFG sConfig;
HRESULT      hr;

hr = XAT_SelectHardware( hwndParent
                       , &sConfig );
if ( 1 == hr )
{
  INT32 hBrd = VCI2_PrepareBoard( sConfig.board_type
                                , sConfig.board_no
                                 , sConfig.sz_CardAddString
                                 , strlen(sConfig.sz_CardAddString)
                                 , ...);
  ...
}
```

3.2.2 XAT_GetConfig

Function: HRESULT XATREG_CALLATTR **XAT_GetConfig**
(DWORD dw_key, XAT_BoardCFG* pConfig);

Description: Reads the configuration of the PC/CAN-interface which is registered with the system under the unique index number dw_key. The configuration is deposited in the structure indicated by the pointer pConfig.

Parameter: **dw_key (in)**
Unique index number under which the PC/CAN-interface is registered with the system.

pConfig (out)
Pointer to a data structure in which retrieves the board configuration.

Return value: ERROR_SUCCESS –> success
HRESULT error code otherwise

Example:

```
XAT_BoardCFG sConfig;
hr = XAT_GetConfig( dwBrdKey     // Unique board index that identifies
                                 // the board.
                  , &sConfig );
if ( ERROR_SUCCESS == hr )
{
  INT32 hBrd = VCI2_PrepareBoard( sConfig.board_type
                                , sConfig.board_no
                                , sConfig.sz_CardAddString
                                , strlen(sConfig.sz_CardAddString)
                                , ...);
  ...
}
```

3.2.3 XAT_EnumHWEntry

Function: HRESULT XATREG_CALLATTR **XAT_EnumHwEntry**
(ENUM_CALLBACK fp_callback, void * vp_context);

Description: Enumerates all registered IXXAT PC/CAN-interfaces. For each entry the Call-back-function transferred in the parameter fp_callback is called up.

Parameter: **fp_callback (in)**
Pointer to the Call-back-function which is called up for each entry.

vp_context (in)
Optional context which is passed to the Call-back-function.

Return value: ERROR_SUCCESS –> success
HRESULT error code otherwise

Example:

```
short EnumCallback ( int  i_index
                   , int  i_hw_key
                   , char* name
                   , char* value
                   , char* valuehex
                   , void* vp_context)
{
  // callback for hardware entry?
  if ( 0 == i_index )
  {
    // get hardware configuration
    XAT_BoardCFG sConfig;
    if ( ERROR_SUCCESS == XAT_GetConfig(i_hw_key, &sConfig) )
    {
      // using the attributes of sConfig you may open the board
      // via VCI2_PrepareBoard function
  }

  return 0;
}


int main(int argc, char* argv[])
{
  ...
  XAT_EnumHwEntry( EnumCallback, 0 );
  ...
}
```

3.2.4 XAT_FindHWEntry

Function: HRESULT XATREG_CALLATTR **XAT_FindHwEntry** (BYTE b_typ, DWORD * p_dw_key, int* p_i_boardtyp, char ca_entryname[255], DWORD dw_arg);

Description: Search for certain registered PC/CAN-interface. Several search options are supported which can be selected via parameter b_typ.

Parameter: **b_typ (in)**

The Parameter b_typ decides on the type of search to be carried out:

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION

Search for a board (its unique index number) by the given board type and the board type related index (e.g. search for the second registered USB-to-CAN interface).

- XATREG_FIND_RELATIVE_BTYPE_POSITION

Search for the board type related index of a board by the given unique board index.

- XATREG_FIND_ADDRESS

Search for a board by its given board address. This is reasonable only for ISA cards.

- XATREG_FIND_ENTRY_WITH_VALUE

Search for a board by its given board type and a board parameter / board parameter value combination (e.g. Search for the tinCAN with IRQ 15). The supported board type specific parameters are shown in the IXXAT Interfaces applet in the Control Panel.

p_dw_key (in/out)

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION:

(out) Retrieves the unique index of the found board.

- XATREG_FIND_RELATIVE_BTYPE_POSITION

(in) Unique index of the board which board type related index is wanted.

- XATREG_FIND_ADDRESS

(out) Retrieves the unique index of the found board.

- XATREG_FIND_ENTRY_WITH_VALUE

(out) Retrieves the unique index of the found board.

p_i_boardtyp (in/out)

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION:

(in) Type of the board to look for.

- XATREG_FIND_RELATIVE_BTYPE_POSITION

(in) Type of the board to look for
(out) Retrieves the board type related index of the found board.

- XATREG_FIND_ADDRESS

(in) Type of the board to look for.

- XATREG_FIND_ENTRY_WITH_VALUE

(in) Type of the board to look for.

ca_entryname (in)

This parameter is used for XATREG_FIND_ENTRY_WITH_VALUE and specifies the name of the parameter which value is consigned by dw_arg.

dw_arg (in)

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION:

(in) Board type related index of the wanted board.

- XATREG_FIND_RELATIVE_BTYPE_POSITION

Parameter is not relevant here.

- XATREG_FIND_ADDRESS

(in) Address of the wanted board.

- XATREG_FIND_ENTRY_WITH_VALUE

(in) Value of the board parameter which name is specified by ca_entryname.

Return value: ERROR_SUCCESS –> success
HRESULT error code otherwise

Example for XATREG_FIND_BOARD_AT_RELATIVE_POSITION:

Search for the second registered USB-to-CAN interface.

```
DWORD dwBrdKey;
DWORD dwBrdType             = VCI_USB2CAN;
DWORD dwBrdTypeRelatedIndex = 1;            // second USB-to-CAN wanted

HRESULT hr = XAT_FindHwEntry( XATREG_FIND_BOARD_AT_RELATIVE_POSITION
                            , &dwBrdKey
                            , &dwBrdType
                            , NULL
                            , dwBrdTypeRelatedIndex);
if ( ERROR_SUCCESS == hr )
{
  // dwBrdKey holds the unique board index now which can be used to
  // open the found board.
  XAT_BoardCFG sConfig;
  hr = XAT_GetConfig( dwBrdKey
                    , &sConfig );
  if ( ERROR_SUCCESS == hr )
  {
    INT32 hBrd = VCI2_PrepareBoard( sConfig.board_type
                                  , sConfig.board_no )
                                  , ...);
    ...
  }
}
```

Example for XATREG_FIND_RELATIVE_BTYPE_POSITION:

Look for the index of the currently used USB-to-CAN interface.

```
DWORD dwBrdType = VCI_USB2CAN;
DWORD dwBrdKey  = 4;           // Unique machine specific board index
                               // e.g. out of XAT_SelectHardware

HRESULT hr = XAT_FindHwEntry( XATREG_FIND_RELATIVE_BTYPE_POSITION
                            , &dwBrdKey
                            , &dwBrdType
                            , NULL
                            , 0 );
if ( ERROR_SUCCESS == hr )
{
  // dwBrdType now holds the board type related index
  printf( "It's the %uth registered USB-to-CAN\n"
        , dwBrdType+1 );
}
```

Example for XATREG_FIND_ADDRESS:
Search for the installed iPC-I 320 with address 0xD0000.

```
DWORD dwBrdKey;
DWORD dwBrdType    = VCI_IPCI320;
DWORD dwBrdAddress = 0xD0000;

HRESULT hr = XAT_FindHwEntry( XATREG_FIND_ADDRESS
                            , &dwBrdKey
                            , &dwBrdType
                            , NULL
                            , dwBrdAddress );
if ( ERROR_SUCCESS == hr )
{
  // dwBrdKey holds the unique board index now which can be used to
  // open the found board.
}
```

Example for XATREG_FIND_ENTRY_WITH_VALUE:
Search for the installed tinCAN with IRQ 15.

```
DWORD dwBrdKey;
DWORD dwBrdType        = VCI_PCMCIA;
char  caEntryName[255] = "IRQ";
DWORD dwEntryValue     = 15;

HRESULT hr = XAT_FindHwEntry( XATREG_FIND_ENTRY_WITH_VALUE
                            , &dwBrdKey
                            , &dwBrdType
                            , caEntryName
                            , dwEntryValue );
if ( ERROR_SUCCESS == hr )
{
  // dwBrdKey holds the unique board index now which can be used to
  // open the found board.
}
```

3.2.5 XAT_SetDefaultHwEntry

Function: HRESULT XATREG_CALLATTR **XAT_SetDefaultHwEntry** (DWORD dw_key);

Description: Sets the default-hardware entry to the PC/CAN-interface with the index number dw_key.

Parameter: **dw_key (in/out)**
index number of the PC/CAN-interface.

Return value: ERROR_SUCCESS –> success
HRESULT error code otherwise

3.2.6 XAT_GetDefaultHwEntry

Function: HRESULT XATREG_CALLATTR **XAT_GetDefaultHwEntry** (DWORD * p_dw_key);

Description: Determines the default-hardware entry.

Parameter: **p_dw_key (in/out)**
Pointer to a DWORD in which the index number of the PC/CAN-interface is deposited.

Return value: ERROR_SUCCESS –> success
HRESULT error code otherwise

Example:

```
DWORD        dwBrdKey;
XAT_BoardCFG sConfig;
HRESULT      hr;

hr = XAT_GetDefaultHwEntry( &dwBrdKey );
if ( ERROR_SUCCESS == hr )
{
  hr = XAT_GetConfig( dwBrdKey
                    , &sConfig );
  if ( ERROR_SUCCESS == hr )
  {
    INT32 hBrd = VCI2_PrepareBoard( sConfig.board_type
                                  , sConfig.board_no
                                  , sConfig.sz_CardAddString
                                  , strlen(sConfig.sz_CardAddString)
                                  ...);
    ...
  }
}
```

3.2.7 XAT_BoardCFG

Reading of the information on registered PC/CAN-interfaces is done via the structure XAT_BoardCFG:

XAT_BoardCFG	*Description*
~.board_no	Unique index number
~.board_type	type of the PC/CAN-interface
~.sz_brd_name[255]	Name
~.sz_manufacturer[50]	Manufacturer
~.sz_brd_info[50];	additional information
~.sz_CardAddString[255];	Card-specific information

3.2.8 HRESULT error codes

The functions within XATxxReg.DLL are mainly based on the registry access functions from Microsoft. Because of this they the error codes are returned directly. You can use the Win32-API-function FormatMessage() to convert the error code to readable text.

4 Notes on Use of the VCI-DLLs

The Virtual CAN Interface for Windows is implemented as a Dynamic Link Library (DLL).

- The DLL is not integrated like a normal C-library but loaded at the run-time of the application and connected with it dynamically; the functions of the DLL are therefore located in their own compiled module and must be integrated in a certain way; integration is explained in Section 3.1.
- The function **VCI_Init()** should not be used under Windows in normal operation; however, for the development in an Interpreter environment, it can be helpful to reset the VCI explicitly with **VCI_Init()**; however, this should not apply to the release version of the application; there the 'VCI_CancelBoard' must be used. See also the description of VCI_Init().

4.1 Common Notes

The installation of the VCI_V2 comes with Header-Files and examples for the following development systems:

- Visual C++ 6.0
- Borland C++ Builder 4
- Delphi 5

It is possible to develop applications on top of VCI_V2 with other development systems. For this have a look at the documentation of your development system.

For statically linking you need an import library suitable for your system. Most systems ship with little command-line tools to generate an import library from the function signatures of a DLL. Is it not possible to generate the import library you could in all cases load the VCI-DLL dynamically.

Users of script languages and Visualization-Software (e.g. Labview) should determine if the software supports invocation of COM-Objects, and use the VCIWrapper (see Chapter 0).

Refer to our web page http://www.ixxat.com for performance data of the different types of IXXAT CAN-Interfaces.

4.2 Integration of the DLL in an Application

Integration of the DLL can occur in different ways.

- Implicit Import via import library
- Dynamic Import

The Header 'VCI2.H' contains the prototypes for the exportable functions.

4.2.1 Implicit Import during Linking

The DLL can be integrated in a project file of the application by inserting the Import-Library. The Import-Library has the same name as the DLL with the file extension ".LIB". This contains the entries which the Linker uses to create a "Relocation Table". During the run-time, the addresses of the functions of the DLL are entered here. With this procedure, the library is loaded during the start of the application. The installation contains libraries for Microsoft Visual C++ 5.0, Microsoft Visual C++ 6.0 and Borland C++ Builder. Import-libraries can also be created for other compilers by means of the Module-Definition-File (ending ".DEF") also contained in the installation. Please see the documentation of your development environment for the procedure.

4.2.2 Dynamic Import during the Run-time

With the dynamic import, the DLL is not loaded at the start of the application but only when it is actually needed. After this, the DLL can similarly be closed again without ending the application at the same time. This import is done by hand in the application itself.

For this, the Windows-API-functions

- LoadLibrary
- GetProcAdress
- FreeLibrary

Are used. For more information, please see the documentation of the Windows-API. The following code fragment explains the procedure for dynamic loading:

```
HINSTANCE hLibrary:
FARPROC lpVCI_PREPAREBOARD:

hLibrary = LoadLibrary("VCI_11un6");
if (NULL != hLibrary)
{
  lpVCI_PREPAREBOARD = GetProcAddress(hLibrary, "VCI_PREPAREBOARD");
  if (lpVCI_PREPAREBOARD != (FARPROC) NULL)
  {
    *(lpVCI_PREPAREBOARD) ( board_type
                          , board_seg
                          , irq_num
                          , fp_puts
                          , msg_int_hdlr
                          , fp_exc_hdlr
                          , apl_handle);
    ...
  }
  FreeLibrary(hLibrary);
}
```

4.3 Notes for VisualBasic developers

In former times the VCI has been available as a C-API, with some extensions for VisualBasic. VisualBasic has the ability to use functions within DLL's but there are some annoying traps:

- The VisualBasic debugger is not able to handle multithreading outside of COM-objects. Because the VCI is using some internal threads the VisualBasic-IDE could crash when you are within debug mode.
- Problems with the alignment of user defined data types could occur when you are calling a DLL-function.
- There is only a limited support of callback functions within VisualBasic. Even worse the callback support has been removed from Version 5.0 to Version 6.0 of VisualBasic.

All these Problems can be avoided by using a COM-component which encapsulates access to the VCI-DLL. IXXAT Automation GmbH has already implemented such a VCIWrapper component.

You could download the installation file of the VCIWrapper via IXXAT Web server under http://www.ixxat.com. The installation includes a user manual along with a simple VisualBasic example which demonstrates the usage of the VCIWrapper.