Using VCI V2 with Delphi

Version: 1.0
Editor: Schlenker
Date: August, 2003
Doc. No: WP102-0002

Application Note

Contents

1 Overview

VCI (Virtual CAN Interface) is a universal driver package for all PC/CAN interface boards of IXXAT Automation GmbH. It is supplied together with the CAN boards and is available for the operating systems Windows Me/NT/2000/XP. The VCI provides a uniform application programming interface (API) for the most common programming languages in the form of a DLL (Dynamic Link Library).

This document explains the use of the VCI under Delphi. A typical application is described with one transmit and one receive queue based on Windows messages. The programming examples listed here are suitable for Delphi 4 to Delphi 7.

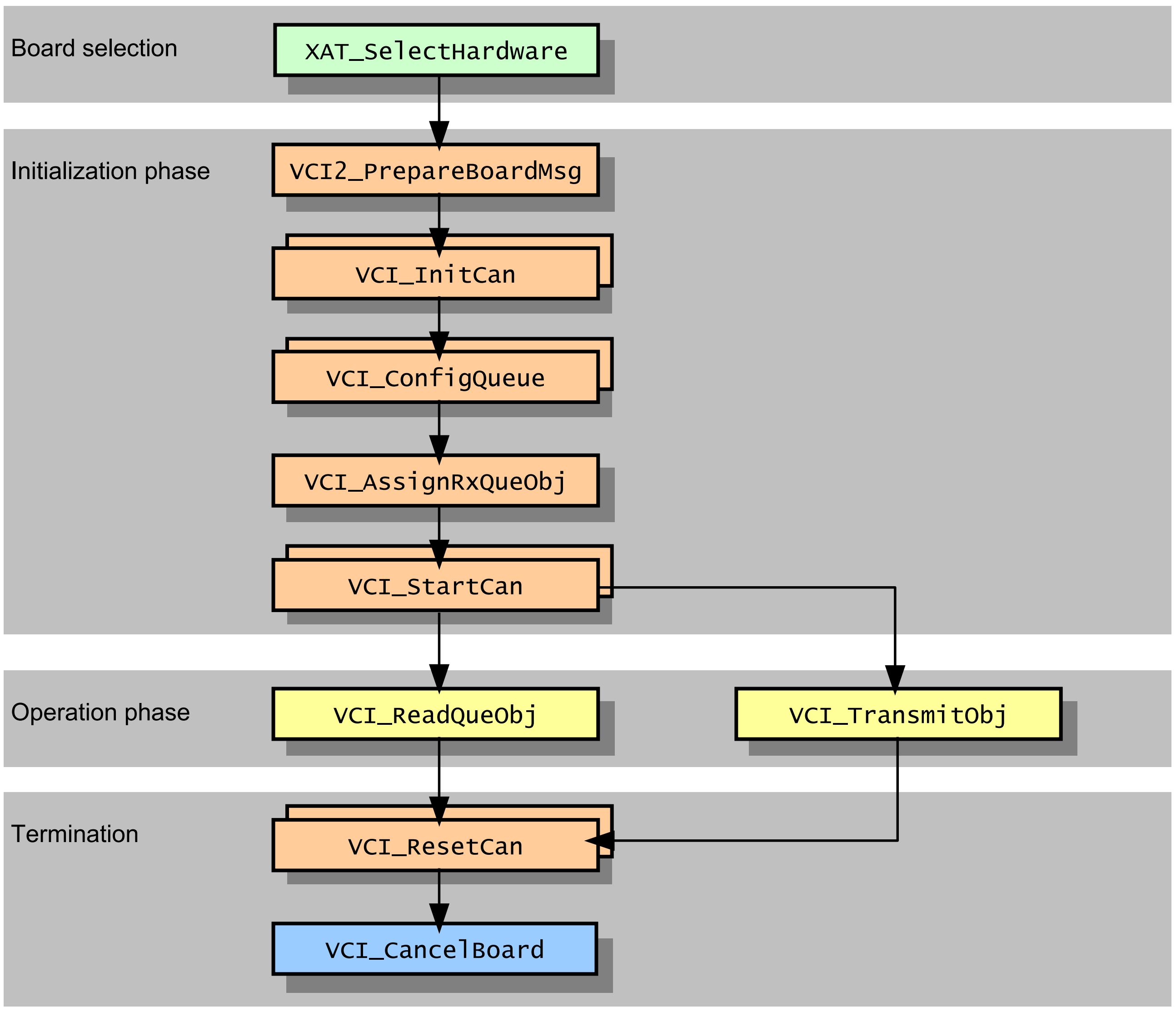
2 Programming of applications with VCI

VCI provides a complete set of functions for programming the CAN controllers and for carrying out CAN communication with other bus subscribers. The broad ability to parameterize VCI functions allows the highly flexible use of the programming library in all CAN application fields.

All functions are implemented according to the C __stdcall call convention. For feedback messages of the API to the application, either callbacks or Windows messages can be selected by the programmer. A comprehensive overview of all aspects of VCI programming is given in the VCI programming manual [2].

2.1 General execution of a VCI application

By way of a simple introduction, the following figure shows the order of VCI function calls for a typical application. This sequence is first sub-divided into board selection, initialization phase, operating phase and termination.



2.2 Delphi header files

The Delphi header `VCI2.pas` contains all VCI functions. For selection of the CAN board and requesting board features and capabilities, the header `XatDynl.pas` is to be integrated.

2.3 Callbacks vs. WM Handler

In a typical application case, VCI is operated with one receive and one transmit queue. For Delphi applications, it is recommended not to use callbacks and define own Windows messages instead. As Delphi requires surface-oriented, event-controlled programming, either the asynchronous callbacks would have to be

synchronized – as with thread programming - or the data would have to be buffered in a common memory range with locked access. Both are difficult to implement and resource-intensive. In addition, the callbacks work in VCI-internal thread contexts, where thread processing is delayed until the callback call returns from the client application.
By using own Windows messages, the CAN data already arrive in the form where they are processed and moreover fully synchronously to the other events. The form elements can be accessed directly in the corresponding message handler.

3 Board selection

For initialization of the CAN interface board, the board type and the computer-specific board identification are required. Both pieces of information can be defined using the so-called IXXAT registry functions. For this, the `XatDynl.pas` header must be integrated and the standard IXXAT hardware selection dialog opened with the command sequence `LoadXatLib; MapXatDialogs; XAT_SelectHardware()`. In the hardware selection dialog the user can then select the board to be used. Its main features are given in the structure `TXAT_BoardCFG`. The elements of this structure, in particular `board_no` and `board_type`, are required for the initialization of the CAN interface board; with some boards (e.g. can@net) the structure element `sz_CardAddString` is also necessary. If an application always uses the same CAN board, it is recommended to store the contents of the `TXAT_BoardCFG` structure in the persistent application data, and not to use the board selection for the subsequent program runs.

Here is a summary of the command sequence for board selection:

```
var
  sBoard : TXAT_BoardCFG;            //  Local allocation of Board configuration record

begin
  //  Dynamically load IXXAT Registry functions
  if not(Assigned(XAT_SelectHardware)) then
    if (LoadXatLib) then
      if (1 <> MapXatDialogs) then
        exit;

  //  Open up standard hardware dialog
  sBoard.board_no := 0;
  if (1 = XAT_SelectHardware(self.Handle, @sBoard)) then
  begin
    {  Save Board configuration data sBoard persistent or programglobal  }
  end;

  //  Unload IXXAT Registry functions
  FreeXatLib;
end;
```

4 VCI functions in the initialization phase of the application

4.1 VCI2_PrepareBoard

As soon as the board is clearly identified, the initialization part can be processed. First the relevant CAN board for the application is opened and allocated. For this the function `VCI2_PrepareBoard`, which works with callbacks, is used. However, as Delphi is to work with Windows messages, the sister function `VCI2_PrepareBoardMsg` is to be used. The function has the following syntax:

```
function VCI2_PrepareBoardMsg ( board_type     : VCI_BOARD_TYPE;
                                board_index    : Word;
                                s_addinfo      : PChar;
                                b_addLength    : Byte;
                                fp_puts        : VCI_t_PutS;
                                msg_rx_int_hdlr: Cardinal;
                                fp_exc_hdlr    : VCI_t_UsrExcHdlr;
                                apl_handle     : HWnd ):             Integer;
```

`board_type`, `board_index` and `s_addinfo` are parameters for the board identification, which can be used directly from the structure `TXAT_BoardCFG` of the board selection. `b_addLength` is the byte length of

the zero-terminated string s_addinfo and can thus also be defined easily. fp_puts is a function pointer to an optional callback function for recording the VCI initialization. It is not examined in more detail here. The next parameter msg_rx_int_hdlr is the actual Windows message identifier, which is to be sent to the application window for receive signaling as soon as a CAN telegram is received. Here you can set an offset value of your choice for WM_USER. In the following parameter fp_exc_hdlr a callback function can again be specified which is called in the event of a fatal error during the VCI initialization phase. This is particularly useful at the beginning of the implementation, as an error description text is transmitted. An implementation example for a suitable Delphi function is given below. The last parameter to be defined is the handle of the window to which the VCI receive message is to be sent.
Apart from the values for board identification, all function parameters are optional. 0 can be defined even as the Windows message identifier and as the window handle. In this case the VCI receive queue would have to be polled by the application.
The return value of the function is the handle of the relevant CAN board, also referred to as board_hdl. In the event of an error, a VCI error code with a value < 0 is returned.

There now follows a simple implementation example for the VCI exception handler fp_exc_hdlr:

```
procedure VCI_UsrExcHdlr (func_num: VCI_FUNC_NUM; err_code: Integer; ext_err: Word;
                          s: PChar);  cdecl;
begin
  Application.MessageBox (s, 'VCI_UsrExcHdlr', MB_TOPMOST);
end;
```

It is to be noted here that the function must not be defined as a member of a Delphi object but as a local function of the unit.

4.2 VCI_InitCan

In the second step of the VCI initialization, the required CAN controller is to be parameterized. The function VCI_InitCan is used for this. This has the following syntax:

```
function VCI_InitCan ( board_hdl : Word;
                       can_num   : Byte;
                       bt0       : Byte;
                       bt1       : Byte;
                       const mode: Byte ): Integer;
```

board_hdl identifies the CAN interface board allocated by means of VCI_PrepareBoardMsg and is returned by that function. The required controller is defined on the board the can_num value. The CAN controllers available on the board are count-up in order, beginning with 0. In the two parameters bt0 and bt1, the values of the bit-timing registers of the CAN controller are defined - VCI2.pas already contains constants for programming the usual baud rates, e.g. VCI_1000KB_BT0 and VCI_1000KB_BT1. The last parameter of the function defies the operating mode of the CAN controller. Three different values are possible here, which are also pre-defined as constants: VCI_11B for standard identifiers and VCI_29B for extended identifiers. A mixed mode is not supported by VCI2!
The return value of the function is a VCI error code. If successful, VCI_OK is returned, in the event of an error a value of < 0.

4.3 VCI_ConfigQueue

In the next step the VCI receive queue and where applicable the VCI transmit queue is to be created. This is done with the function VCI_ConfigQueue. It has the following syntax:

```
function VCI_ConfigQueue ( board_hdl    : Word;
                           can_num      : Byte;
                           que_type     : Byte;
                           que_size     : Word;
                           int_limit    : Word;
                           int_time     : Word;
                           ts_res       : Word;
                           var p_que_hdl: Word ): Integer;
```

Parameterization of the VCI queues is a difficult undertaking. For standard applications, the following configuration has been proved successful:

```
VCI_ConfigQueue (m_hXatBoard, 0, VCI_RX_QUE, 100, 1, 100, 100, m_hRxQueue);
```

or

```
VCI_ConfigQueue (m_hXatBoard, 0, VCI_TX_QUE, 100, 0, 0, 0, m_hTxQueue);
```

The last parameter `p_que_hdl` is important, as here VCI enters the queue handle which clearly identifies the relevant queue.
The return value of the function is a VCI error code. If successful, `VCI_OK` is returned, in the event of an error a value of < 0.

4.4 VCI_AssignRxQueObj

Initialization of the VCI is now complete. However, you will not yet receive anything, as filtering of the receive queue is set as standard so that all identifiers are blocked. Therefore, in a further step, the filter has to be set that all or defined massages will be received. This is done with the function `VCI_AssignRxQueObj`. To receive all CAN telegrams, use the following instruction:

```
VCI_AssignRxQueObj (m_hXatBoard, m_hRxQueue, VCI_ACCEPT, 0, 0);
```

4.5 VCI_StartCan

In the last step, which already marks the transition to the operating phase, start the CAN controller that has just been parameterized. This is done with the function `VCI_StartCan`:

```
function VCI_StartCan ( board_hdl: Word;
                        can_num  : Byte ): Integer;
```

4.6 Program listing of a standard initialization

Here is a summary of the command sequence of the VCI initialization phase in a typical application:

```
var
  res : integer;

begin
  //  Open IXXAT CAN Board described in m_sBoard
  //  On CAN message reception, send WM_VCIRX to ourselves
  //  For VCI errors, use callbackfunction named VCI_UsrExcHdlr
  m_hXatBoard := VCI2_PrepareBoardMsg (m_sBoard.board_type, m_sBoard.board_no,
                                       m_sBoard.sz_CardAddString,
                                       StrLen(m_sBoard.sz_CardAddString), nil,
                                       WM_VCIRX, VCI_UsrExcHdlr, self.Handle);

  //  Parameterisation of first CAN controller
  res := 0;
  if (0 <= m_hXatBoard) then
    res := VCI_InitCan (m_hXatBoard, 0, VCI_125KB_BT0, VCI_125KB_BT1, VCI_11B);

  //  Configuration of receive queue
  if (VCI_OK = res) then
    res := VCI_ConfigQueue (m_hXatBoard, 0, VCI_RX_QUE, 100, 1, 100, 100, m_hRxQueue);
  if (VCI_OK = res) then
    res := VCI_AssignRxQueObj (m_hXatBoard, m_hRxQueue, VCI_ACCEPT, 0, 0);

  //  Configuration of transmit queue
  if (VCI_OK = res) then
    res := VCI_ConfigQueue (m_hXatBoard, 0, VCI_TX_QUE, 100, 1, 100, 100, m_hTxQueue);

  //  Let’s go
  if (VCI_OK = res) then
    res := VCI_StartCan (m_hXatBoard, 0);
end;
```

5 VCI functions in the operating phase of the application

5.1 VCI_ReadQueObj

In a VCI application that not uses callbacks and Windows messages, the receive queue must be polled. For this, VCI provides the function `VCI_ReadQueObj`:

```
function VCI_ReadQueObj ( board_hdl: Word;
                          que_hdl  : Word;
                          count    : Word;
                          p_obj    : PVCI_CAN_OBJ_ARRAY ): Integer;
```

The function copies a selectable number of CAN receive telegrams into a memory range managed by the programmer. The memory range must be allocated as a vector (array) or individual element of the structure `VCI_CAN_OBJ`. The size and commencement location address of the memory range are transferred to the function as `count` and `p_obj`; `board_hdl` and `que_hdl` identify the CAN board and the queue. The maximum number of supported vector dimensions is 13, i.e. up to 13 CAN objects can be transferred by VCI with one function call.
The return value of the function is the number of CAN objects actually copied (where applicable 0), in the event of an error a VCI error code < 0 is returned.

Here is an implementation example for the continuous query of the VCI receive queue:

```
var
  asCanObj : VCI_CAN_OBJ_ARRAY;            //  Local allocation of CAN receive buffer
  pCanObj  : PVCI_CAN_OBJ;
  i        : integer;
  iVCIres  : integer;                      //  VCI result

begin
  while not(Application.Terminated) do
  begin
    pCanObj := @asCanObj;
    iVCIres := VCI_ReadQueObj ( m_hXatBoard, m_hRxQueue,
                                sizeof(asCanObj) div sizeof(VCI_CAN_OBJ), @asCanObj );
    if (0 < iVCIres) then
    for i:=1 to iVCIres do
    begin
      {  Process received CAN message pCanObj  }
      inc(pCanObj);
    end;
    Sleep(10);
  end;
end;
```

5.2 Receive WM Handler

If a Window message handler is defined, or if working with a VCI Rx callback function, the allocation of a local VCI receive buffer as with `VCI_ReadQueObj` is not necessary, as both the commencement location address of the VCI receive buffer and the number of the receive objects are supplied directly. In the receive WM Handler, the received CAN objects can therefore be accessed immediately:

```
procedure WMVCIRX (var Msg: TVciRxMsg); message WM_VCIRX;
var
  pCanObj : PVCI_CAN_OBJ;
  i       : integer;

begin
  pCanObj := @Msg.pData.sData;
  for i:=1 to Msg.dwCnt do
  begin
    {  Process received CAN message pCanObj  }
    inc(pCanObj);
  end;
end;
```

The condition for the allocation of the `pCanObj` is that the compiler switch `{$TYPEDADDRESS}` (“Typed @-Operator“) is deactivated – this conveniently already corresponds to the standard project setting. However, the type definition of the VCI receive Windows message `TVciRxMsg`, which is available in the `VCI2.pas` header only from VCI Version 2.16, is much more important. Therefore, if necessary, insert these type definitions in the `interface` section of your Delphi unit:

```
type
  PVciRxMsgData = ^TVciRxMsgData;
  TVciRxMsgData = packed record
    bQue  : byte;            //  Handle of VCI RxQueue
    sData : VCI_CAN_OBJ_ARRAY;
  end;
  TVciRxMsg = packed record   //  Equals Messages.TMessage
    Msg   : Cardinal;         //  WM_VCIRX (or equivalent)
    dwCnt : dword;            //  WPARAM
    pData : PVciRxMsgData;    //  LPARAM
    Result: Longint
  end;
```

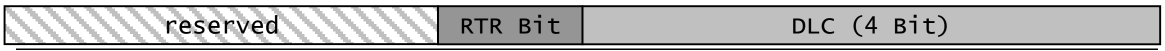
On the one hand, this declares a Delphi-compatible VCI receive Windows message, on the other data structures are introduced for direct access to the VCI CAN data transferred in the Windows message.

5.3 VCI_CAN_OBJ structure

In conclusion, a few words on the format of the VCI CAN data themselves: every CAN message received by VCI is provided ion a structure named `VCI_CAN_OBJ`. In addition to the CAN telegram (ID + 8 data bytes), this also includes the timestamp, RTR bit and various status information:

```
VCI_CAN_OBJ = packed record
  time_stamp    : Longint;
  id            : Longint;
  len4_rtr1_res3: Byte;
  a_data        : VCI_CAN_DATA;
  sts           : Byte;
end;
```

`time_stamp` is the absolute value of the time of reception of the CAN frame, standardized to the time interval `ts_res` defined in the function `VCI_ConfigQueue`. In `id`, the identifier of the CAN telegram, both for 11-bit and for 29-bit frames, is right adjusted. `len4_rtr1_res3` defines a bit field. It is made up of the DLC of the CAN frame and the RTR bit. The upper three bits of the structure element are reserved, but not set to 0:



Composition of the structure element VCI_CAN_OBJ.len4_rtr1_res3

`a_data` contains the data field of the CAN frame. The number of valid bytes, i.e. the length of the data field, is obtained by masking the DLCs: `(VCI_CAN_OBJ.len4_rtr1_res3 and $F)`. The last element `VCI_CAN_OBJ.sts` contains VCI-internal status information.

5.4 VCI_TransmitObj

Compared with the acceptance of received CAN data, transmission of a CAN message is fairly simple. The function `VCI_TransmitObj` available for this has the following syntax:

```
function VCI_TransmitObj ( board_hdl: Word;
                           que_hdl  : Word;
                           id       : Longint;
                           len      : Byte;
                           pData    : PVCI_CAN_DATA ): Integer;
```

`board_hdl` and `que_hdl` identify the CAN interface board and the VCI transmit queue. The CAN identifier `id`, the CAN data field `pData` and the length of the data field used `len`, are transmitted as individual pa-

rameters.
The return value of the function is a VCI error code. If successful, `VCI_OK` is returned, in the event of an error a value of < 0.

VCI functions for termination of the application

When terminating the application, it is necessary to call `VCI_CancelBoard`:

```
function VCI_CancelBoard (board_hdl: Word): Integer;
```

This call must not be made only when unloading the application (or DLL), but must be made already during the regular run time of the program, for example in the OnDestroy handler of the main form:

```
procedure TFrmMain.FormDestroy (Sender: TObject);

begin
  if (0 <= m_hXatBoard) then
    VCI_CancelBoard (m_hXatBoard);

  {  Further cleanup  }
end;
```

It is recommended to deactivate the CAN controller used already at the start of program deinitialization with `VCI_ResetCan`, so that no CAN receive telegrams interfere with program deinitialization. The syntax of the corresponding VCI function is:

```
function VCI_ResetCan ( board_hdl: Word;
                        can_num  : Byte ): Integer;
```

6 Further Literature

[1] Application Note “Building Applications with the VCI CAN driver“

[2] VCI Programming Manual

[3] CAN book:
Prof. Dr.-Ing. K. Etschberger (Publisher)
Controller Area Network (CAN)
Basics, Protocols, Chips, Applications
English edition, 2001
ISBN: 3-00-007376-0, 440 Pages

7 Contact Information

IXXAT Automation GmbH	**IXXAT Inc.**	**Others**
Leibnizstr. 15 88250 Weingarten Germany Tel.: +49-(0)7 51 / 5 61 46-0 Fax: +49-(0)7 51 / 5 61 46-29 E-Mail: info@ixxat.de Internet: www.ixxat.de	120 Bedford Center Road Bedford, NH 03110 USA Phone: +1-603-471-0800 Fax: +1-603-471-0880 E-mail: sales@ixxat.com Internet: www.ixxat.com	For a list of international distributors please consult the IXXAT web page under: www.ixxat.de | contact | distributors