Using VCI V2 with LabWindows/CVI

IXXAT

Version: 1.0
Editor: Schmid
Date: October, 2003
Doc. No: WP102-0003

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 LabWindows/CVI. A typical application is described with one transmit and one receive queue based on a callback or polling functions. The programming examples listed here are suitable for Measurement Studio Version 6.

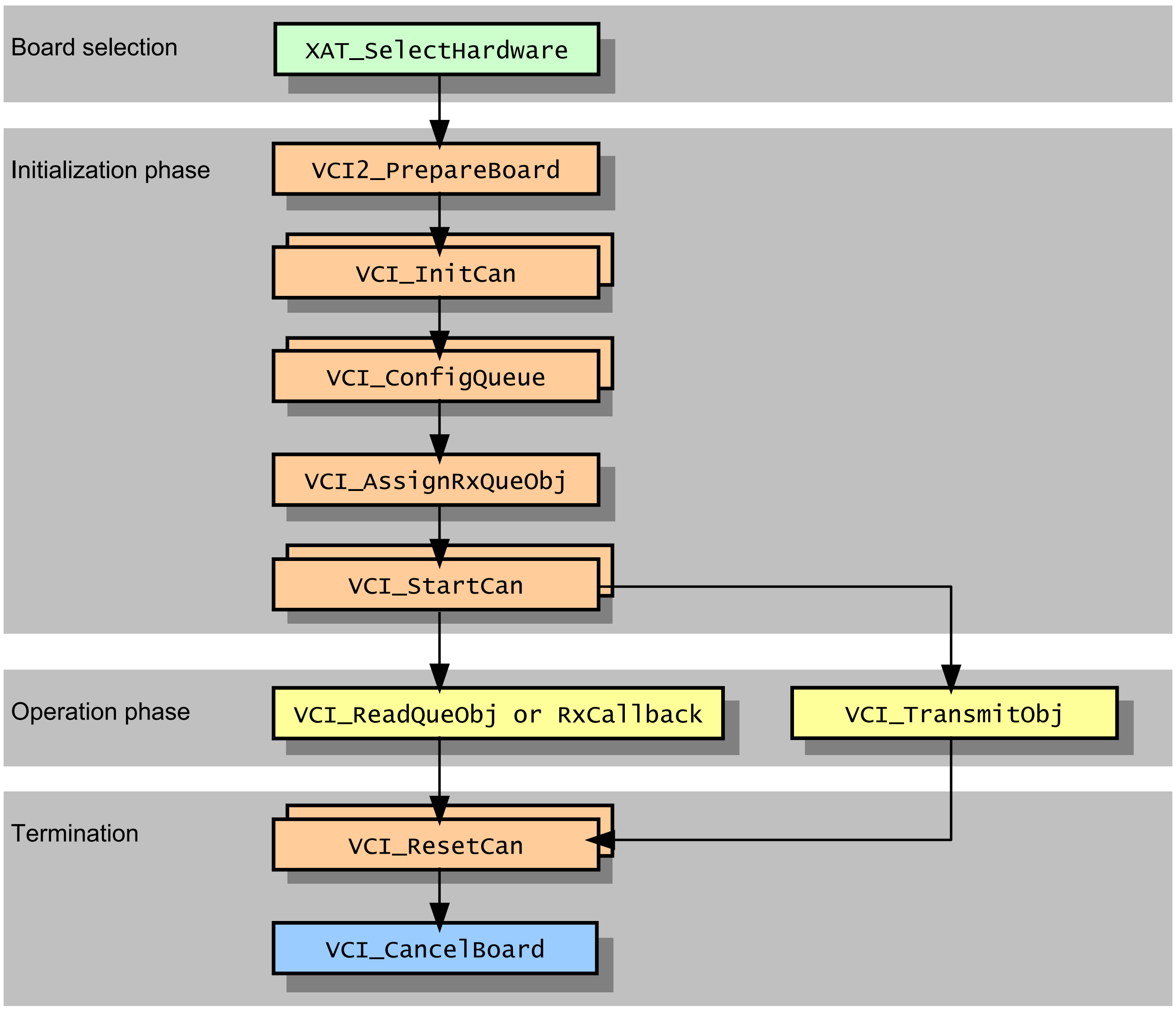
2 Programming of applications with VCI

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. Alternatively the receive queue can be polled. 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 C header files

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

2.3 Callbacks vs. WM Handler

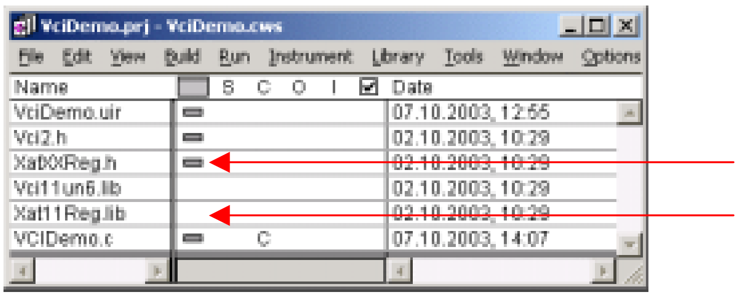
In a typical application case, VCI is operated with one receive and one transmit queue. For LabWindows/CVI applications, it is recommended not to use Windows messages and define own callback functions instead. As LabWindows/CVI is based on ANSI-C and does not provide the window handles of the user interface panels there is no sense in using Windows messages here. There is no possiblilty to handle these.

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 `XatxxReg.h` header and the associated Xat11Reg.lib library must be integrated. After that the standard IXXAT hardware selection dialog can be opened with the command `XAT_SelectHardware()`. In the hardware selection dialog the user can select the board to be used. Its main features are given in the structure `XAT_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 `XAT_BoardCFG` structure in the persistent application data, and not to use the board selection for the subsequent program runs. Further possiblities are the declaration of a default board (refer to XAT_GetDefaultHwEntry description in the VCI programmer manual) or to search for a specific board (refer to XAT_FindHwEntry description in the VCI programmer manual).

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

(1) Add XatxxReg.h and Xat11Reg.lib to the project.


VciDemo.prj - VciDemo.cws
File Edit View Build Run Instrument Library Tools Window Options

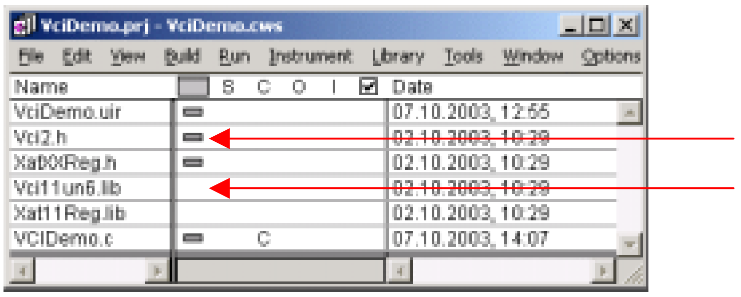
Name		B	C	O	I	✓	Date
VciDemo.uir	▬						07.10.2003, 12:55
Vci2.h	▬						02.10.2003, 10:29
XatXXReg.h	▬						02.10.2003, 10:29
Vci11un6.lib							02.10.2003, 10:29
Xat11Reg.lib							02.10.2003, 10:29
VCIDemo.c	▬		C				07.10.2003, 14:07

(2) Add command sequence to the "open hardware selection" dialog.

```
XAT_BoardCFG sConfig;

// Open hardware selection dialog
if ( XAT_SelectHardware(NULL, &sConfig) )
{
  // Save Board configuration data sBoard persistent or programglobal
}
```

4 VCI functions in the initialization phase of the application

For using the VCI function for initialization, operation and termination phase the `VCI2.h` header and the associated Vci11un6.lib library have to be added to the project:



Name		S	C	O	I	☑	Date
VciDemo.uir	▬						07.10.2003, 12:55
Vci2.h	▬						02.10.2003, 10:29
XatXXReg.h	▬						02.10.2003, 10:29
Vci11un6.lib							02.10.2003, 10:29
Xat11Reg.lib							02.10.2003, 10:29
VCIDemo.c	▬		C				07.10.2003, 14:07

4.1 VCI2_PrepareBoard

As soon as the board is clearly identified, the initialization part can 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. The function has the following syntax:

```
INT32 VCI2_PrepareBoard( VCI_BOARD_TYPE     board_type,
                         UINT16             board_index,
                         char*              s_addinfo,
                         UINT8              b_addLength,
                         VCI_t_PutS         fp_puts,
                         VCI_t_UsrRxIntHdlr fp_int_hdlr,
                         VCI_t_UsrExcHdlr   fp_exc_hdlr);
```

`board_type`, `board_index` and `s_addinfo` are parameters for the board identification, which can be used directly from the structure `XAT_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 `fp_int_hdlr` is the actual callback handler function, which is to be called for receive signaling as soon as a CAN telegram is received. 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 C function is given below. Apart from the values for board identification, all function parameters are optional. 0 can be defined as parameter value. 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`:

```
void VCI_CALLBACKATTR VciExcCallback( VCI_FUNC_NUM func_num, INT32 err_code,
                                      UINT16       ext_err,  char* s )
{
  char szError[256];

  sprintf(szError,
          "Func(%u) Err(%d) ExtErr(%#X) : %s",
          func_num,
          err_code,
          ext_err,
          s);
  SetCtrlVal(hPanel, PANEL_EXCEPTIONBOX, szError);
}
```

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. It has the following syntax:

```
INT32 VCI_InitCan( UINT16 board_hdl,
                   UINT8  can_num,
                   UINT8  bt0,
                   UINT8  bt1,
                   UINT8  mode);
```

`board_hdl` identifies the CAN interface board allocated by means of `VCI2_PrepareBoard` and is returned by that function. The required controller is defined on the board by the `can_num` variable. 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.h` already contains defines for programming the usual baud rates, e.g. `VCI_1000KB`. The last parameter of the function defies the operating mode of the CAN controller. Two 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:

```
INT32 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);
```

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

```
VCI_ConfigQueue (hBoard, 0, VCI_RX_QUE, 100, 1, 100, 100, &hRxQue);
```

or

```
VCI_ConfigQueue (hBoard, 0, VCI_TX_QUE, 100, 0, 0, 0, &hTxQue);
```

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 messages will be received. This is done with the function `VCI_AssignRxQueObj`. To receive all CAN telegrams, use the following instruction:

```
VCI_AssignRxQueObj (hBoard, hRxQue, 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`:

```
INT32 VCI_StartCan( UINT16 board_hdl,
                    UINT8  can_num );
```

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:

```
INT32  hVciRes = VCI_OK;
UINT16 hBoard  = 0xFFFF;


//  Open IXXAT CAN Board described in sConfig.
//  On CAN message reception call VciRxCallback.
//  For VCI errors use callbackfunction named VciExcCallback);
hVciRes = VCI2_PrepareBoard( sConfig.board_type, sConfig.board_no,
                             sConfig.sz_CardAddString, strlen(sConfig.sz_CardAddString),
                             NULL, VciRxCallback, VciExcCallback);
if ( 0 <= hVciRes )
{
  hBoard = hVciRes;

  // Initialize CAN controller
  hVciRes = VCI_InitCan(hBoard, 0, VCI_125KB, VCI_11B);
  if ( VCI_OK == hVciRes )
  {
    // Configuration of transmit queue
    hVciRes = VCI_ConfigQueue(hBoard, 0, VCI_TX_QUE, 20, 0, 0, 0, &hTxQue);
    if ( VCI_OK == hVciRes )
    {
      // Configuration of receive queue
      hVciRes = VCI_ConfigQueue(hBoard, 0, VCI_RX_QUE, 100, 1, 100, 100, &hRxQue);
      if ( VCI_OK == hVciRes )
      {
        // Open rx queue filter
        hVciRes = VCI_AssignRxQueObj(hBoard, hRxQue, VCI_ACCEPT, 0, 0);
        if ( VCI_OK == hVciRes )
        {
          // Start CAN contoller
          hVciRes = VCI_StartCan(hBoard, 0);
        }
      }
    }
  }
}
```

5 VCI functions in the operating phase of the application

The condition for the correct access of all received CAN objects is that the alignment is set to 1 via the following #pragma statement before the #include of `VCI2.h`:

```
#pragma pack(1)

#include “VCI2.h”
```

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`:

```
INT32 VCI_ReadQueObj( UINT16       board_hdl,
                      UINT16       que_hdl,
                      UINT16       count,
                      VCI_CAN_OBJ* p_obj);
```

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:

```
VCI_CAN_OBJ asCanObj[13];     //  Local allocation of CAN receive buffer
INT32       i;
INT32       lVciRes;          // VCI result;

while(1)
{
  lVciRes = VCI_ReadQueObj(hBoard, hRxQue, sizeof(asCanObj)/sizeof(VCI_CAN_OBJ),
                           asCanObj);
  for (i=0; i<lVciRes; i++)
  {
    // Process received CAN object asCanObj[i]
  }
  Sleep(10);
};
```

5.2 Receive callback

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 callback handler, the received CAN objects can therefore be accessed immediately:

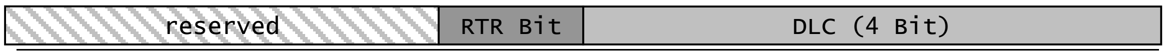
```
void VCI_CALLBACKATTR VciRxCallback( UINT16        que_hdl,
                                     UINT16        count,
                                     VCI_CAN_OBJ  *p_obj )
{
  UINT16 wObj;

  for (wObj=0; wObj<count; wObj++)
  {
    // Process received CAN object p_obj[wObj]
  }
}
```

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 in 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:

```
typedef struct{
  UINT32 time_stamp;
  UINT32 id;
  UINT8  len:4;
  UINT8  rtr:1;
  UINT8  res:3;
  UINT8  a_data[8];
  UINT8  sts;
}VCI_CAN_OBJ;
```

`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. `len, rtr and res` define 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 len. 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:

```
INT32 VCI_TransmitObj( UINT16  board_hdl,
                       UINT16  que_hdl,
                       UINT32  id,
                       UINT8   len,
                       UINT8  *p_data);
```

`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 parameters.
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.

6 VCI functions for termination of the application

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

```
INT32 VCI_CancelBoard(UINT16 board_hdl);
```

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 main function right before the return call:

```
int main (int argc, char *argv[])
{
  // Board selection

  // Initilization phase

  DisplayPanel (hPanel);
  RunUserInterface ();
  DiscardPanel (hPanel);

  VCI_ResetCan(hBoard, 0);
  VCI_CancelBoard(hBoard);
  return 0;
}
```

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:

```
INT32 VCI_ResetCan( UINT16 board_hdl,
                    UINT8 can_num );
```

7 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

8 Contact Information

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