Handbuch

VCI - Virtual CAN Interface

VCI-V2 Programmier-Handbuch

IXXAT
The expert for industrial and automotive communication

IXXAT

Hauptsitz
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

Geschäftsbereich USA
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

Sollten Sie zu diesem, oder einem unserer anderen Produkte Support benötigen, wenden Sie sich bitte schriftlich an:

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

Copyright

Dokument Nr. VCIV2_Programmier-Handbuch-D-V2.9

1 Einleitung

Das Virtual CAN Interfaces (**VCI**) ist ein leistungsfähiges Softwarepaket für die IXXAT-PC/CAN-Interfaces. Es wurde für Softwareentwickler konzipiert, welche anspruchsvolle, hardwareunabhängige CAN-Applikationen für den PC entwickeln wollen.

Deshalb wurde besonderer Wert auf einfache Anwendbarkeit sowie auf gutes Echtzeitverhalten des VCI gelegt.

1.1 Anwendungsgebiete

Ziel des VCI ist, dem Anwender eine einheitliche Programmierschnittstelle für die verschiedenen PC/CAN-Interfacevarianten der Firma IXXAT zur Verfügung zu stellen. Dabei spielt weder die Ausführung der PC-Ankopplung (PCI(e), USB, TCP-IP ...) noch der verwendete CAN Controller des Interfaces eine Rolle. Außerdem bietet das VCI die Möglichkeit gleichzeitig mehrere (auch verschiedene) Karten zu betreiben.

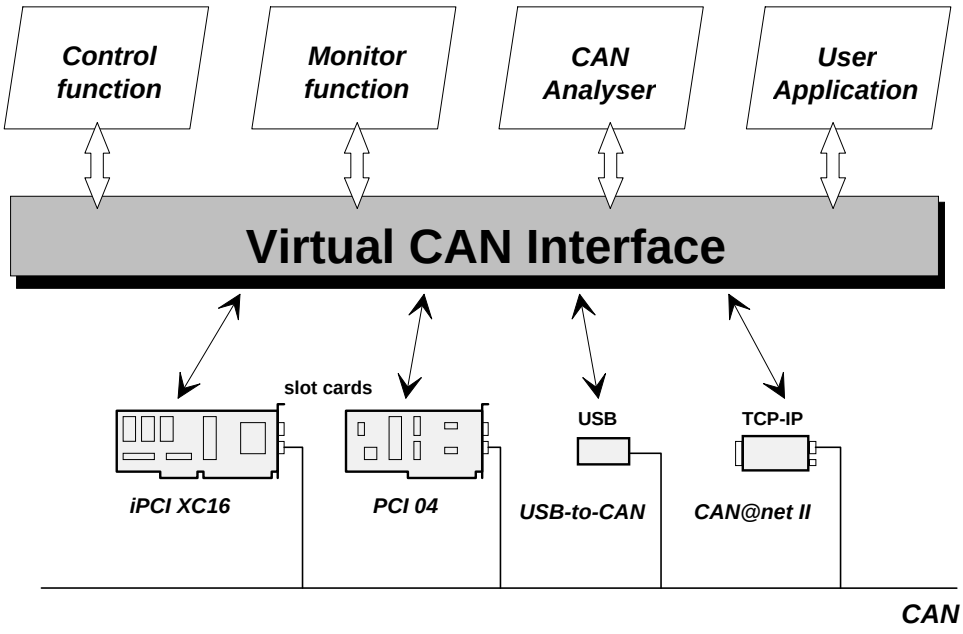

Bild 1 - 1 Virtual CAN Interface

Dieses Konzept ermöglicht eine vom eingesetzten PC/CAN-Interfacetyp unabhängige Realisierung von Anwendungsprogrammen.

Im VCI wurde hierfür ein virtueller CAN Controller definiert, dessen Struktur einem Basic-CAN-Controller entspricht und der den Betrieb mit 11-Bit und 29-Bit Identifier unterstützt. Diesem virtuellen CAN-Controller ist eine Firmware nachgeschal-

tet, welche die Nachrichtenverwaltung organisiert. Der virtuelle CAN-Controller kann bis zu 4-mal auf einem PC/CAN-Interface vorhanden sein, wobei ein gleichzeitiger Betrieb von bis zu 4 Karten möglich ist.

Von der VCI werden sowohl intelligente PC/CAN-Interfaces (mit eigenem Speicher und eigener CPU) sowie auch passive Karten unterstützt.

Aktive PC/CAN-Interfacekarten unterstützen den PC bei der Vorverarbeitung der CAN-Nachrichten sowie bei der Datenhaltung. Dies wirkt sich positiv auf die Prozessorbelastung des PCs aus.

Bei passiven Karten wird der Prozessor des PCs durch die Interruptroutine des CAN-Controllers und der Nachrichtenverwaltung wesentlich mehr belastet. Andererseits ermöglichen passive PC/CAN-Interfacekarten eine preisgünstige Ankopplung eines PCs an ein CAN-Netz. Es werden allerdings hohe Anforderungen an das Echtzeitverhalten des PC gestellt (unter Windows nur bei niedrigen Baudraten sinnvoll).

1.2 Hinweise zu diesem Handbuch

Ziel dieses Handbuches ist es, die Funktionsweise des VCI und deren Funktionen zu erläutern.

Es kann **nicht** Aufgabe dieses Handbuches sein, die komplette Problematik der Programmierung von CAN-Anwendungen zu beschreiben sowie eine Referenz für die Funktionalität einzelner CAN-Controller darzustellen.

Dieses Handbuch setzt Kenntnisse der Programmerstellung unter MS-Windows (Multi-Threading, ereignisgesteuerte Verarbeitung) voraus.

Bevor Sie mit dem VCI arbeiten, sollten Sie **unbedingt** dieses Handbuch mindestens einmal vollständig durchgelesen haben.

Um diese Dokumentation möglichst kurz zu gestalten ist die enthaltene Information weitgehend ohne Redundanz. Es wird deshalb ein mehrmaliges Durcharbeiten des Handbuches empfohlen, da beim ersten Durchlesen wichtige Informationen oft übersehen werden.

In diesem Zusammenhang wird auch das Studium des Headerfiles VCI.h dringend empfohlen.

1.3 Installation des VCI

Hinweise zur Installation des PC/CAN-Interfaces entnehmen Sie bitte dem mitgelieferten „PC/CAN-Interface Hardwarehandbuch“.

Der Ablauf der VCI-Softwareinstallation ist im "CAN-Treiber VCI Installationshandbuch" beschreiben.

1.4 Funktionsumfang des VCI

Das VCI unterstützt:

- Standard und Extended Protokoll (11 und 29-Bit-Identifier)
- Mehrere CAN-Controller pro Interface (soweit durch die Hardware unterstützt)
- Gleichzeitiger Betrieb von bis zu vier Interfaces durch eine oder mehrere Applikationen
- Baudraten bis 1000 Kbaud
- Empfang von Nachrichten über konfigurierbare Empfangsqueues (FIFO's) mit Zeitmarke
- Empfang von Nachrichten über konfigurierbare Empfangsbuffer mit Empfangszähler
- Jedem CAN-Controller können mehrere Queue und Buffers zugeordnet werden.
- Senden von Nachrichten (erfolgt über konfigurierbare Sendequeues)
- Queues können gepollt oder per Interrupt gelesen werden (Timeout oder ´Hochwassermarke´)
- Automatische, konfigurierbare Beantwortung von Anforderungsnachrichten (Remoteframes)

Das VCI liefert außerdem statistische Daten zum CAN-Bus, zum CAN-Controller, über die Datenstrukturen sowie die PC/CAN-Interfaces.

1.5 Einschränkungen

- Der Zugriff auf ein PC/CAN-Interface ist exklusiv nur für eine Applikation möglich. Mehrere Applikationen können sich daher nicht ein PC/CAN-Interface teilen.
- Je nach verwendetem CAN-Controller des PC/CAN-Interface bestehen Einschränkungen beim Funktionsumfang des VCI:
  - Philips 82C200: Kein Extended-Protokoll möglich
  - Intel 82527: Kein Remotebetrieb möglich
  - Philips SJA1000: Keine Einschränkungen

  Um die entsprechende Funktionalität zu unterstützen muss Ihre Applikation den entsprechenden CAN-Controller auswählen.
- Remote Buffer nur im 11Bit Standard Mode möglich

1.6 Nachrichtenverwaltung

Das VCI verfügt über eine eigene Nachrichtenverwaltung. In dieser Nachrichtenverwaltung werden sowohl ankommende als auch abgehende Nachrichten in unterschiedlichen Strukturen verwaltet.

Die Zwischenspeicherung der empfangenen Nachrichten erfolgt in sog. Receivequeues bzw. in Receivebuffers. Im Falle einer Queue werden die Nachrichten entsprechend der zeitlichen Abfolge ihres Auftretens inklusive einer Zeitmarke abgespeichert (FIFO-Prinzip), wobei die Nachrichten auch unterschiedliche Identifier haben können. Ein Buffer enthält dagegen jeweils nur die zuletzt unter einem bestimmten Identifier empfangene Nachricht (entsprechend einem aktuellen Prozeßabbild) sowie einem Zähler für die Anzahl der Empfangsvorgänge auf diesem Buffer.

Die zu sendenden Nachrichten werden in Transmitqueues eingetragen. Diese werden dann vom Mikrocontroller (nur bei intelligenten PC/CAN-Interfaces) oder einer Interruptroutine des PC abgearbeitet. Außerdem können auch sog. Remotebuffers eingerichtet werden, in welche Nachrichten eingetragen werden, die nicht direkt, sondern erst bei Anforderung (Remote-Frames) durch einen anderen Netzknoten gesendet werden.

Nachfolgend werden die von der VCI zur Verwaltung der CAN-Nachrichten bereitgestellten Elemente beschrieben.

1.6.1 Receivebuffer

Receivebuffer werden für jeden zu empfangenden Identifier angelegt. Sie enthalten immer die zuletzt, unter dem gewählten Identifier empfangenen Daten. Daraus ergibt sich, dass Daten, welche noch nicht von der Applikation übernommen worden sind, überschrieben werden. Zur Flußkontrolle bei wiederholtem Empfang sind die Receivebuffer mit einem Empfangszähler versehen. Receivebuffer werden von der Applikation gezielt auf das Vorhandensein neuer Daten geprüft und die Daten anschließend übernommen.

Ein ereignisgesteuertes Lesen von Receivebuffern ist nicht möglich, da Receivebuffer i.A. dann zum Einsatz kommen, wenn die Applikation nur sporadisch bestimmte Prozessdaten zu prüfen hat und sich dabei nur für die jeweils aktuellsten Daten interessiert.

Die maximal konfigurierbare Anzahl von Buffer (Receive- und Remotebuffer zusammen) beträgt 2048 pro CAN-Controller.

1.6.2 Receivequeue

Der Einsatz von Receivequeues empfiehlt sich vor allem für solche Anwendungsfälle, bei denen alle Daten entgegengenommen werden sollen, welche unter einem oder mehreren Identifiern übertragenen werden und bei denen das Anwen-

dungsprogramm nicht in der Lage ist, unmittelbar auf den Empfang einer Nachricht zu reagieren.

Das Anwendungsprogramm kann dabei festlegen, wie viele Nachrichten eine Queue aufnehmen kann und welche Identifier einer Queue zugeführt werden sollen. Es können mehrere Queues angelegt werden, so dass durch das VCI bereits eine Vorsortierung vorgenommen werden kann. Alle in einer Receivequeue eingetragenen Nachrichten sind mit einem Zeitstempel versehen.

Falls aufgrund der Struktur des Anwendungsprogramms ein regelmäßiges Pollen der Receivequeue(s) nicht sinnvoll bzw. nicht möglich ist, kann die Applikation über eine Callback-Funktion benachrichtigt werden.

Der Zeitpunkt der Benachrichtigung ist konfigurierbar und alternativ durch folgende Ereignisse angestoßen:

- Überschreiten einer bestimmten Anzahl von Einträgen in einer Queue (Erreichen einer "Hochwassermarke")
- Ablauf einer bestimmten Zeit (Timeoutfunktion).

Die Callback-Funktion wird vom Interrupt-Thread der VCI aus aufgerufen. Dadurch entstehen mehrere Einschränkungen:

- Innerhalb der Callback-Funktion sollten keine zeitkritischen Berechnungen durchgeführt werden, da sonst evtl. CAN-Nachrichten verloren gehen können.
- Sie befinden sich innerhalb der Callback-Funktion im Kontext des Interrupt-Threads. Der Versuch, auf Daten ihrer Applikation zuzugreifen, kann evtl. daran scheitern.
  Eine Möglichkeit, den Callback von ihrer Applikation zu entkoppeln, besteht darin, einen Applikations-Thread für die Bearbeitung einer Queue zu starten. Ankommende CAN-Nachrichten werden in ihrer Callback-Funktion durch Setzen eines Events signalisiert. Der Applikations-Thread wartet auf dieses Event und führt nach dem Setzen des Events die Bearbeitung durch. Nach dem Bearbeitungsschritt fällt er wieder in den Wartezustand zurück.

Die maximal konfigurierbare Anzahl von Receivequeues beträgt 16 pro CAN-Controller.

1.6.3 Transmitqueue

Das Senden von Nachrichten (Daten- und Datenanforderungen) aus der Applikation erfolgt grundsätzlich über Transmitqueues. Auf diese Weise braucht die Applikation beim Stellen eines Sendeauftrags nicht darauf zu warten, dass der CAN-Controller sendebereit wird. Die Abarbeitung der Transmitqueue(s) übernimmt der Mikrocontroller der aktiven PC/CAN-Interfaces bzw. bei passiven PC/CAN-Interfaces die Interruptroutine des PC.

Es können mehrere Queues mit unterschiedlicher Größe (Anzahl der Nachrichten) und unterschiedlicher Priorität eingerichtet werden. Die unterschiedlichen Prioritäten der Queues bestimmen, in welcher Reihenfolge sie durch den Mikrocontroller bearbeitet werden.

Maximal können bis zu 8 Transmitqueues pro CAN-Controller konfiguriert werden.

1.6.4 Remotebuffer

Sollen Daten zur Anforderung durch andere Netzknoten bereitgehalten werden, so können diese in sog. Remotebuffer eingetragen werden. Im Falle des Empfangs einer Anforderungsnachricht (Remoteframe) mit passendem Identifier werden die Daten aus dem Buffer entnommen und gesendet. Die Applikation braucht dabei lediglich die Daten im Buffer zu aktualisieren. Die Bearbeitung einer Anforderungsnachricht erfolgt mit höchster Priorität, also vor der Bearbeitung der Transmitqueues.

Anforderungsnachrichten können alternativ jedoch auch über eine Receivequeue empfangen werden. Die Applikation veranlasst in diesem Fall das Senden der angeforderten Daten selbst, indem sie eine entsprechende Nachricht in eine Transmitqueue einträgt.

Die maximal konfigurierbare Anzahl von Buffer (Receive- und Remotebuffer zusammen) beträgt 2048 pro CAN-Controller.

1.6.5 Öffnen eines PC-CAN Interfaces

Die Funktionen VCI2_PrepareBoard oder VCI2_PrepareBoardMsg werden aufgerufen um ein IXXAT PC-CAN Interfaceboard zu öffnen. Das zu öffnende Board wird hierbei über eine Indexnummer identifiziert, die Sie zuvor über eine von vier Möglichkeiten ermitteln:

- Manuelle Auswahl des Boards im Hardwareauswahl-Dialog (siehe Kapitel 3.2.1)
- Abfragen der Parameter des im IXXAT Interfaces Control Panel Applet als „Default" markierten Boards (siehe Kapitel 3.2.6).
- Enumerieren aller installierten IXXAT PC-CAN Interfaces (siehe Kapitel 3.2.3).
- Suchen nach einem IXXAT PC-CAN Interface anhand spezifischen Eigenschaften (siehe Kapitel 3.2.4)

2 Schnittstellenbeschreibung

Die VCI-Anwenderschnittstelle stellt dem Anwender PC-seitig eine Sammlung von Funktionen zur Verfügung, welche auf das PC/CAN-Interface zugreifen und die Kommunikation über CAN abwickeln. Die Schnittstelle unterscheidet vier Klassen von Funktionen:

- Funktionen zur Steuerung und Konfiguration des PC/CAN Interfaces
- Funktionen zur Kontrolle und Konfiguration der CAN-Controller
- Funktionen zum Empfangen von Nachrichten
- Funktionen zum Senden von Nachrichten

Die Funktionen werden nachfolgend beschrieben. Mitgelieferte Beispielprogramme zeigen die Anwendung der Funktionen.

2.1 Vordefinierte Returncodes des VCI

Um in Zukunft auch andere PC/CAN-Interfacetypen unterstützen zu können und da es nicht möglich ist, heute schon alle Fehler und Returncodes zu spezifizieren, welche bei zukünftigen Implementierungen auftreten können, werden alle möglichen Returncodes über folgende Defines beschrieben. Zusätzliche Informationen (Fehlerstring und weitere Parameter) werden über den Exceptionhandler der VCI (Callback-Funktion) geliefert.

Define	*Wert*	*Fehlerbeschreibung*
VCI_OK	1	Erfolgreiche, nicht weiter spezifizierte Rückmeldung bei korrekt ausgeführten Funktionen
VCI_ERR	0	Standard-Fehlermeldung, weitere Spezifizierung erfolgt über den Exceptionhandler
VCI_QUE_EMPTY	0	Receivequeue leer, keine Nachrichten können gelesen werden
VCI_QUE_FULL	0	die Transmitqueue ist bereits voll, es können im Moment keine weiteren Einträge vorgenommen werden
VCI_OLD	0	es sind keine neuen Daten im Receivebuffer vorhanden, es werden ggf. alte Daten gelesen
VCI_HWSW_ERR	-1	Funktion konnte aufgrund von Hardware- oder Softwarefehlern nicht ausgeführt werden, überprüfen Sie die Funktion des PC/CAN-Interfaces
VCI_SUPP_ERR	-2	Funktion wird in dieser Form nicht unterstützt (support error), überprüfen Sie anhand der Implementierungsübersicht zu Ihrer Plattform Ihren Fehler
VCI_PARA_ERR	-3	übergebene(r) Parameter ist/sind fehlerhaft oder außerhalb des gültigen Bereichs, prüfen Sie die Parameter
VCI_RES_ERR	-4	Ressourcen-Error, beim Anlegen einer Queue etc. sind die Ressourcengrenzen (Speicher, max. Anzahl der Queues, etc.) überschritten worden, überprüfen Sie anhand der Implementierungsübersicht zu Ihrer Plattform Ihren Fehler
VCI_QUE_ERR	-5	Receivequeue Overrun: Mindestens eine Nachricht konnte nicht mehr in die Receivequeue eingetragen werden und ging verloren. Die letzte erfolgreich eingetragene Nachricht wurde mit gesetztem Queue-Overrun Bit markiert.
VCI_TX_ERR	-6	es konnte über einen längeren Zeitraum (einige Sekunden) keine Nachricht über CAN gesendet werden, was auf fehlende Teilnehmer, fehlenden Busabschluß oder falsche Baudrate schließen lässt, überprüfen Sie Ihren CAN-Anschluss und Verkabelung.

Wird bei einem VCI_ERR als Fehlerstring des Exceptionhandlers ein 'CciReqDataError' gemeldet, so handelt es sich um einen Fehler bei der Kommunikation zwischen PC zum PC/CAN-Interface. Nachfolgend die Liste der möglichen Fehler:

Wert	*Bedeutung*
0	Kommando konnte nicht an das PC/CAN-Interface übergeben werden
1	als Antwort vom PC/CAN-Interface kam kein OK sondern ein Error zurück
2	es kam die falsche Antwort auf das Kommando zurück
3	beim Warten auf die Antwort ist ein Timeout aufgetreten
4	Antwort ist zu kurz (falsche Länge)
5	beim Übergeben eines Kommandos an das PC/CAN-Interface ist ein Timeout aufgetreten

Die hier aufgelisteten Fehlern lassen sich meistens auf Installationsprobleme zurückführen, wie:

- Speicherbereich des PC/CAN-Interfaces wird nicht korrekt im Adressraum des PCs eingeblendet (Fehler Nummer 0,1 oder 5).
- Interrupt des PC/CAN-Interfaces wird nicht korrekt an den PC weitergeleitet oder ist durch andere Einsteckkarten belegt (Fehler Nummer 3).
- Die Kommunikation zum PC/CAN-Interface (z. B. Interfaces mit LPT-Schnittstelle) ist gestört.

2.2 Typdefinitionen der Callbackhandler

Callbackhandler sind Funktionen, welche vom Anwender codiert werden und (in diesem Falle von der VCI) aufgerufen werden, wenn bestimmte Ereignisse eintreten.

Sie dienen in diesem Falle zur Fehleranzeige und Fehlerbehandlung, Verarbeitung von Interruptnachrichten oder zur Ausgabe von Test- oder Initialisierungsprotokollen.

Damit die VCI diese Callbackhandler kennt und ausführen kann, müssen diese Funktionen den festgelegten Typdefinitionen entsprechen und der VCI über 'VCI2_PrepareBoard' bekannt gemacht werden.

Soll beispielsweise durch eine Receivequeue ein Interrupt ausgelöst werden, so muss vom Anwender eine entsprechende Funktion codiert werden (Callbackhandler). Diese Funktion muss für jedes installierte PC/CAN-Interface, das Interrupts auslösen soll, codiert werden.

Dabei liegt es dem Anwender offen, die Möglichkeiten des Callback-Handlings zu nutzen oder darauf zu verzichten und 'VCI2_PrepareBoard' anstatt einem gültigen Funktionspointer nur einen NULL-Pointer zu übergeben.

2.2.1 Receive-Interrupt-Handler

An diese Funktion werden die über den Interrupt empfangenen Queuenachrichten (Timeout oder "Hochwassermarke") übergeben, sofern dies über VCI_ConfigQueue angegeben wurde.

Dieser Callbackhandler wird für 2 unterschiedliche Interruptmechanismen genutzt:

- Übertragung von Nachrichten über den Interrupt Callback (max. 13 Nachrichten gleichzeitig)
- Signal einer Receivequeue an die Anwendung

Im ersten Fall werden die Alarmnachrichten beim Interrupt mit übergeben, im zweiten Fall wird nur ein Signal an den Anwender übergeben (count = 0).

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

Parameter:

que_hdl (in)
Handle der Queue, welche den Interrupt ausgelöst hat.

count (in)
Anzahl der empfangenen Nachrichten.

p_obj (in)
FAR-Pointer auf die empfangene(n) Nachricht(en) vom Typ VCI_CAN_OBJ.

Returnwerte: keine

2.2.2 Exception-Handler

Diese Funktion wird immer dann aufgerufen, wenn in einer Systemfunktion ein Fehler aufgetreten ist. Dieser Fehler wird in diesem Fall nicht nur über den Returnwert angezeigt, sondern wird auch an den Exceptionhandler übergeben. Damit stehen dem Anwender zwei Möglichkeiten der Fehlerbehandlung zur Verfügung, wobei die Variante über den Exceptionhandler einen übersichtlicheren Programmcode erlaubt.

An den Exceptionhandler werden Strings mit einer genaueren Fehlerspezifikation übergeben, welche in einem Fehlerfenster ausgegeben oder in ein File geschrieben werden können.

Diese Null-terminierten Strings (ohne Steuerzeichen) mit einer max. Länge von 60 Zeichen geben zum einen den Funktionsnamen der Funktion an, in welchem der Fehler aufgetreten ist, und zum anderen wird der Fehler genauer spezifiziert.

Pro PC/CAN-Interface muss ein eigener Exceptionhandler codiert werden.

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

Parameter: **func_num (in)**
Typname aus dem Aufzählungstyp VCI_FUNC_NUM, über den die Funktion spezifiziert wird, in welcher der Fehler aufgetreten ist.

err_code (in)
Über Defines spezifizierte Standard-Error-Codes (VCI_SUPP_ERR, VCI_PARA_ERR, ...).

ext_err (in)
weitere Fehlerspezifikationen beim Standard-Error-Code VCI_ERR (siehe unten).

s (in)
Fehlerstring (max. 60 Zeichen) mit Angabe des Funktionsnamen sowie weiterer Fehlerspezifikation.

Returnwerte: keine

2.2.3 Handler zur Stringausgabe

Für die Funktionen VCI_TestBoard oder VCI2_PrepareBoard besteht die Möglichkeit, eine Ausgabefunktion zu spezifizieren, über die ein Test- oder Initialisierungsprotokoll ausgegeben werden kann.

An diese Funktion werden Null-terminierte Strings (ohne Steuerzeichen) mit einer max. Länge von 60 Zeichen übergeben.

Typdefinition: typedef void (***VCI_t_PutS**)(char * s);

Parameter: **s (in)**
Fehlerstring (max. 60 Zeichen) mit Angabe des Funktionsnamen sowie weiterer Fehlerspezifikation.

Returnwerte: keine

2.3 Zustandsdiagramm zur Boardinitialisierung

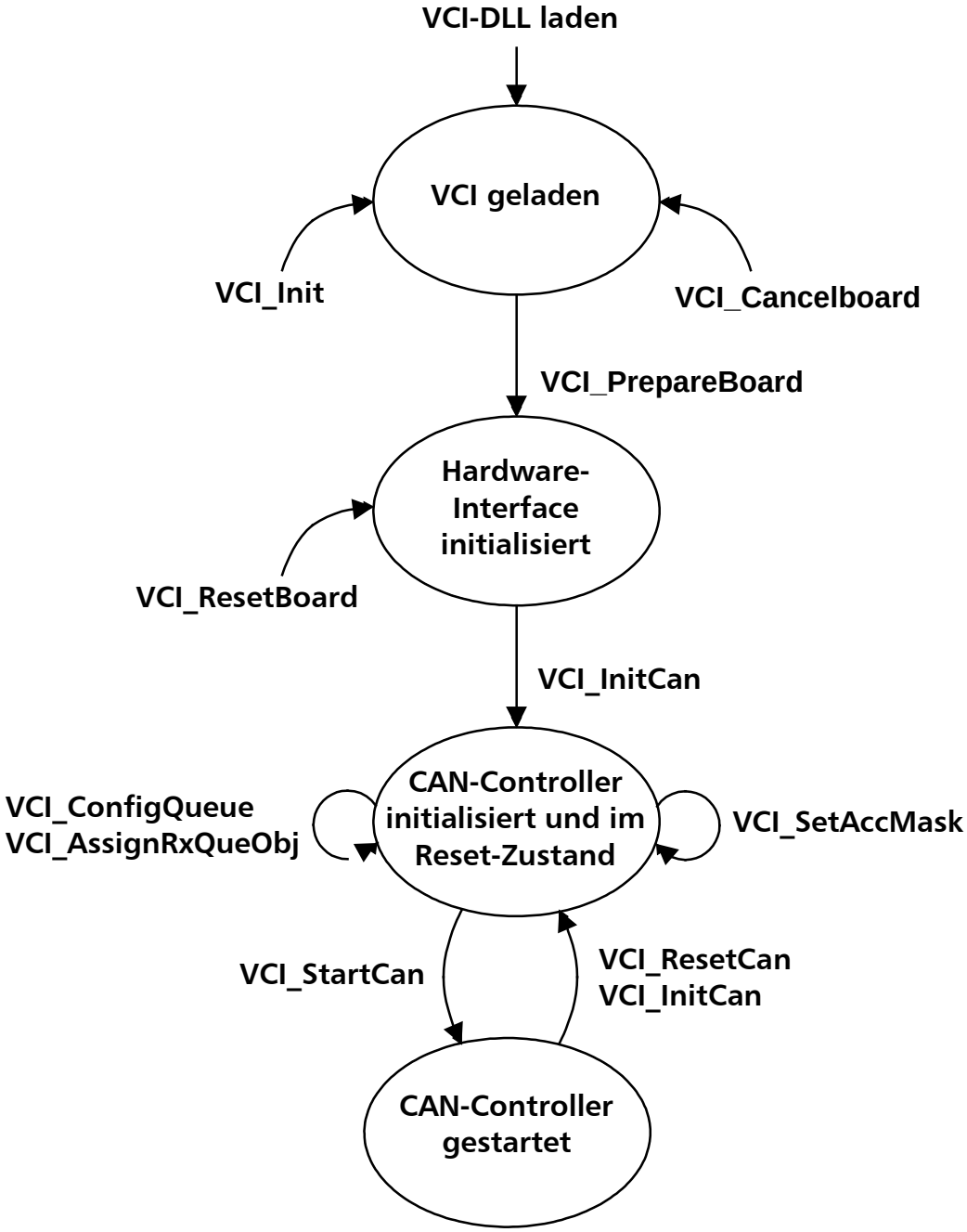

Bild 2 - 1 Zustandsdiagramm zur Boardinitialisierung

2.4 Tabelle der VCI Funktionen

In dieser Tabelle werden die VCI-Funktionen nach Verwendungszweck aufgelistet. Zu jeder VCI Funktion wird angegeben in welchem Zustand (siehe Zustandsdiagramm der VCI Kap. 2.3) ein Aufruf möglich ist.

VCI – Zustand / VCI-Funktionen	Geladen	Board angelegt	Controller initialisiert und im Reset	Queue/Buffer angelegt	Controller gestartet	Bemerkung
Allgemeine Funktionen						
VCI_Init	x	x	x	x	x	Diese Funktion darf nur während der Entwicklungsphase benutzt werden! Siehe Beschreibung.
VCI_Get_LibType	x	x	x	x	x	Nur aus Kompatibilitätsgründen
VCI_GetBrdTypeInfo	x	x	x	x	x	Nur aus Kompatibilitätsgründen
VCI_GetBrdNameByType	x	x	x	x	x	Nur aus Kompatibilitätsgründen
VCI_GetBrdTypeByName	x	x	x	x	x	Nur aus Kompatibilitätsgründen
Board-Verwaltung						
VCI_Prepareboard VCI_PrepareboardMsg	x					Nur aus Kompatibilitätsgründen
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 – Zustand / VCI-Funktionen	Geladen	Board angelegt	Controller initialisiert und im Reset	Queue/Buffer angelegt	Controller gestartet	Bemerkung
CAN-Verwaltung						
VCI_ReadCanInfo		x	x	x	x	
VCI_ReadCanStatus		x	x	x	x	
VCI_InitCan		x	x	x	x	
VCI_SetAccMask			x			Zum Setzen der Akzeptanzmaske muss sich der CAN-Controller im Reset befinden!
VCI_ResetCan			x	x	x	
VCI_StartCan			x	x	x	Vor dem Starten des CANs sollten alle notwendigen Initialisierungen wie das Anlegen der Queues/Buffers abgeschlossen sein.
Queue-Verwaltung						
VCI_ConfigQueue			x			
VCI_AssignRxQueObj				x		
VCI_ReadQueObj				x	x	
VCI_ReadQueStatus				x	x	
VCI_TransmitObj					x	
Buffer-Verwaltung						
VCI_ConfigBuffer			x			
VCI_ReConfigBuffer				x		
VCI_ReadBufStatus				x	x	
VCI_ReadBufData				x	x	

VCI – Zustand / VCI-Funktionen	Geladen	Board angelegt	Controller initialisiert und im Reset	Queue/Buffer angelegt	Controller gestartet	Bemerkung
Remote Buffer – Verwaltung						
VCI_RequestObj					x	
VCI_UpdateBufObj				x	x	

2.5 Initialisierung des VCI

2.5.1 VCI_Init

Funktion: void VCI_Init(void);

Beschreibung: Initialisierung der VCI-Strukturen (ohne Board-Initialisierung). Bereits initialisierte Boards werden zurückgesetzt und abgemeldet (alle Handles werden verworfen!!!).

Diese Funktion sollte nur während der Programmentwicklung von VCI-Applikationen genutzt werden. Falls während eines Tests eine VCI-Applikation abgestürzt ist, und die VCI in einem undefinierten Zustand hinterlassen hat, können die internen Datenstrukturen mit dieser Funktion zurückgesetzt werden.

Dadurch kann auf die VCI wieder aufgesetzt werden ohne den Rechner neu booten zu müssen.

Beachten Sie: Ein Aufruf der Funktion VCI_Init() bewirkt, dass alle Handles der momentan laufenden VCI-Applikationen ungültig werden. Dies kann zu Abstürzen führen, wenn zum Zeitpunkt des Aufrufs bereits eine VCI-Applikation ihr Board initialisiert hat und danach versucht darauf zuzugreifen:
Daher: In Release-Versionen einer VCI-Applikation nicht VCI-Init ausführen!

Returnwert: keine

2.6 Funktionen für VCI Support Informationen

Diese Funktionen wurden früher benutzt, um Informationen zu den verschiedenen Interface-Typen zu bekommen. Sie werden durch die Funktionen der XATxxReg.DLL (siehe Kap. 3) abgelöst.

2.6.1 VCI_Get_LibType

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. Sie sollte nicht mehr benutzt werden.

2.6.2 VCI_GetBrdNameByType

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. Sie sollte nicht mehr benutzt werden.

2.6.3 VCI_GetBrdTypeByName

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. Sie sollte nicht mehr benutzt werden.

2.7 Funktionen zur Boardinitialisierung

2.7.1 VCI_SearchBoard

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. Sie sollte nicht mehr benutzt werden.

2.7.2 VCI_SetDownloadState

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. In der aktuellen Version der VCI ist diese Funktion ohne jegliche Aktion. Sie sollte daher nicht mehr benutzt werden.

2.7.3 VCI2_PrepareBoard und VCI2_PrepareBoardMsg

Von der Funktion VCI2_PrepareBoard existieren mehrere verschiedene Varianten, welche sich durch folgende Eigenschaften unterscheiden:

- Parameterübergabe entsprechend der **VCI V1** oder konform der **VCI V2**.
  Die Parameter zur Angabe des PC/CAN-Interfaces müssen bei der Variante **VCI V1** für den Anwender bekannt sein und explizit angegeben werden.
  Bei der Variante **VCI V2** werden die Parameter zur Angabe des PC/CAN-Interfaces über die XAT-Registierungsfunktionen gewonnen (siehe Kap. 3).
- Nutzung einer **Callbackfunktion** oder eines **Messagehandlers** für die Interrupt gesteuerte Verarbeitung von empfangenen Nachrichten. (siehe Kap. 2.2 sowie Kap. 2.9.1).

Übersicht der Varianten:

Funktion	*Einsatz*
VCI_PrepareBoard	VCI V1 mit Callbackfunktion
VCI2_PrepareBoard	VCI V2 mit Callbackfunktion
VCI_PrepareBoardMsg	VCI V1 mit Message-Handler
VCI2_PrepareBoardMsg	VCI V2 mit Message-Handler

Beschreibung: Funktion beantragt die Verwendung eines PC/CAN-Interfaces bei der VCI. Dies beinhaltet das Rücksetzen des Interface, Firmware-Download und Starten der Firmware auf intelligenten Interfaces sowie die Registrierung der Callback-Funktionen. Die CAN-Controller des Interfaces werden in den Init-Mode versetzt.

Als Returnwert wird ein Handle auf das PC/CAN-Interface zurückgeliefert, unter dem das Interface angesprochen werden kann. Handles werden als aufsteigende Nummern von Null an vergeben (0, 1, 2, ...n).

Die Funktion muss ausgeführt werden, bevor auf das Interface zugegriffen wird. Bereits angemeldete und damit durch ein Programm belegte Interfaces können ***nicht*** nochmals angemeldet werden. (Soll das PC/CAN-Interface von einer anderen Applikation genutzt werden, muss das Interface zuvor über VCI_CancelBoard freigegeben werden.)

Durch VCI2_PrepareBoard werden auch die Callbackhandler gesetzt.

- PutString zur Bildschirmausgabe bei PrepareBoard.
- Exceptionhandler zur Fehlerbehandlung.
- Receiveinterrupthandler **oder** Message-Handler für den Interruptbetrieb, je nach Variante.
  Siehe dazu Typdefinitionen Callbackhandler.

2.7.3.1 VCI_PrepareBoard

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. Sie sollte nicht mehr benutzt werden. Bitte nutzen Sie die **VCI2_PrepareBoard** Funktion.

2.7.3.2 VCI2_PrepareBoard

Funktion: 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);

Beschreibung: siehe unter 2.7.3

Parameter:

board_type (in)
ein Integer-Wert, der den benutzten Typ des PC/CAN-Interfaces kennzeichnet. Wird nur zur Konsistenzprüfung verwendet, da das Board über den Parameter Boardindex eindeutig bestimmt wird.

board_index (in)
Im Parameter board_index wird eine eindeutige Indexnummer übergeben, unter der das PC/CAN-Interface beim System registriert ist. Gültige Indexnummern können über die Registrierungsfunktionen (siehe Kapitel 3) durch die Applikation bestimmt werden.

s_addinfo (in)
Zeiger auf einen Puffer mit maximal 256 Byte, der Zusatzinformationen aufnehmen kann. Diese Zusatzinformation wird von der VCI_V2 verwaltet und kann vom Benutzer im Hardware-Auswahldialog geändert werden.

Falls sie diese Information nicht benötigen setzen Sie diesen Parameter auf null.

Im Moment wird dieser Parameter nur für spezielle Hardware genutzt. Standard IXXAT PC/CAN-Interfaces nutzen diese Zusatzinformation nicht.

b_addLength (in)
Maximale Länge des Puffers für die Zusatzinformation in s_addinfo.

fp_puts (in)
Callbackfunktion zur Ausgabe von Fehler- und Statusmeldungen beim Prepare
(NULL -> keine Statusausgabe).

fp_int_hdlr (in)
Funktionspointer auf die Routine zur Bearbeitung der empfangenen Nachrichten.
(NULL -> keine Interruptverarbeitung)

fp_exc_hdlr (in)
Funktionspointer auf den Exceptionhandler zur Verarbeitung der aufgetretenen Fehler
(NULL -> kein Exceptionhandler)

Returnwert:
>= 0 -> Board-Handle
< 0 -> VCI-Return Codes.

2.7.3.3 VCI_PrepareBoardMsg

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. Sie sollte nicht mehr benutzt werden. Bitte nutzen Sie die **VCI2_PrepareBoardMsg** Funktion.

2.7.3.4 VCI2_PrepareBoardMsg

Funktion: 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);

Beschreibung: siehe unter 2.7.3

Über diese Funktion wird statt einer Callback-Funktion für die Interruptverarbeitung der VCI-DLL ein Windows Message-Identifier ('msg_int_hdlr') und ein Windows-Handle ('apl_handle') übergeben.

Mit der Windows Message werden der Anwendung, die durch das Windows Handle referenziert wird, die folgenden Parameter mit übergeben.

WPARAM count
(Anzahl der CAN-Nachrichten, die mit der Message übergeben werden)

LPARAM Zeiger auf übertragene Daten
1. BYTE QueRef
(gibt die Queue an, die den Interrupt ausgelöst hat)
2..n. BYTE CAN_OBJ
(die durch ´count´ angegebene Anzahl der CAN_OBj des Typs VCI_CAN_OBJ)

Bsp.:

```
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)
ein Integer-Wert, der den benutzten Typ des PC/CAN-Interfaces kennzeichnet. Wird nur zur Konsistenzprüfung verwendet, da das Board über den Parameter Boardindex eindeutig bestimmt wird.

board_index (in)
Im Parameter board_index wird eine eindeutige Indexnummer übergeben, unter der das PC/CAN-Interface beim System registriert ist. Gültige Indexnummern können über die Registrierungsfunktionen (siehe Kapitel 3) durch die Applikation bestimmt werden.

s_addinfo (in)
Zeiger auf einen Puffer mit maximal 256 Byte, der Zusatzinformationen aufnehmen kann. Diese Zusatzinformation wird von der VCI_V2 verwaltet und kann vom Benutzer im Hardware-Auswahldialog geändert werden.

Falls sie diese Information nicht benötigen setzen Sie diesen Parameter auf null.

Im Moment wird dieser Parameter nur für spezielle Hardware genutzt. Standard IXXAT PC/CAN-Interfaces nutzen diese Zusatzinformation nicht.

b_addLength (in)
Maximale Länge des Puffers für die Zusatzinformation in s_addinfo.

fp_puts (in)
Callbackfunktion zur Ausgabe von Fehler- und Statusmeldungen beim Prepare
(NULL -> keine Statusausgabe).

msg_rx_int_hdlr (in)
ID der Windows-Nachricht mit der der Empfang von CAN-Nachrichten an die Applikation gemeldet werden soll. Typischerweise wird eine benutzerdefinierte Nachricht (WM_USER + Offset) festgelegt und an dieser Stelle übergeben.

fp_exc_hdlr (in)
Funktionspointer auf den Exceptionhandler zur Verarbeitung der aufgetretenen Fehler
(NULL -> kein Exceptionhandler)

apl_handle (in)
Handle der Applikation an die die vereinbarte Windows-Nachricht geschickt werden soll.

Returnwert: >= 0 -> Board-Handle
< 0 -> VCI-Return Codes.

2.7.4 VCI_PrepareBoardVisBas

Leider wurde die Möglichkeit Callbacks zu nutzen, ab Microsoft Visual Basic 6.0 wieder eingeschränkt. Sie sollten diese Funktion deshalb unter Visual Basic nicht benutzen.

2.7.5 VCI_CancelBoard

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

Beschreibung: Angemeldetes Board beim VCI abmelden. Dies beinhaltet das Rücksetzen des Interfaces und der CAN Controller sowie das De-installieren der verwendeten Interrupts. Der Boardhandle wird dadurch wieder frei.

Parameter: **board_hdl (in)**
Handle eines zuvor angemeldeten Boards.

Returnwert: VCI-Return Codes.

2.7.6 VCI_TestBoard

Hinweis: Diese Funktion ist nur noch aus Kompatibilitätsgründen enthalten. In der aktuellen Version der VCI ist diese Funktion ohne jegliche Aktion. Sie sollte daher nicht mehr benutzt werden.

2.7.7 VCI_ReadBoardInfo

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

Beschreibung: Lesen der Board-Informationen entsprechend VCI_BOARD_INFO:

VCI_BOARD_INFO	*Beschreibung*
~.hw_version	Hardware-Version als HEX-Wert (z. B: 0x0100 für V1.00)
~.fw_version	Firmware-Version als HEX-Wert
~.dd_version	Device-Driver-Version als HEX-Wert (nur für PC card)
~.sw_version	Versionsnummer der PC-Software als HEX-Wert
~.can_num	Anzahl der vom Board unterstützten CAN-Controller
~.time_stamp_res	kleinstmögliche Timestamp Auflösung in 100nsec
~.timeout_res	kleinstmögliche Timeout Auflösung für die Receivequeues
~.mem_pool_size	Größe der Speicherpools für die Einrichtung von Queues und Buffers
~.irq_num	Interruptnummer zur Kommunikation mit dem PC/CAN-Interface
~.board_seg	eingestellte Board-Adresse/Segment/Portnummer
~.serial_num	16-Zeichen-String mit der Seriennummer des Boards
~str_hw_type	Null-terminierter String mit der Hardware-Kennung

Die Zeitangaben zur Timestamp bzw. Timeout Auflösung dienen zur sinnvollen Einstellung dieser Zeiten.

Die Funktionsausführung ist optional und dient nur zur Spezifizierung des PC/CAN-Interfaces.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

p_info (out)
Pointer auf die Infodaten.

Returnwert: VCI-Return Codes.

2.7.8 VCI_ReadBoardStatus

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

Beschreibung: Lesen der Board-Information entsprechend VCI_BRD_STS:

VCI_BRD_STS	*Beschreibung*
~.sts	Bit-kodierte Informationen zum Boardstatus:

Abhängig vom Typ des verwendeten IXXAT PC/CAN-Interfaces kann die Widerspiegelung des real vorhandenen Boardstatus nach Aufruf von Funktionen wie VCI_StartCan oder VCI_ResetCan bis zu 100ms verzögert werden. Während dieser Zeit liefert VCI_ReadBoardStatus möglicherweise einen „veralteten" Status. Verwenden Sie diese Funktion daher nur zur Statusvisualisierung in Ihrer Anwendung und nicht um die Ausführung von VCI Steuerungsfunktionen zu überprüfen.

Die einzelnen Bits von ~.sts haben folgende Bedeutung:

Bit 0: RxQueue Overrun; in einer konfigurierten Receivequeue ist ein Overrun aufgetreten (Queue war bereits voll und eine weitere Nachricht konnte nicht eingetragen werden.) Weitere Informationen liefert VCI_ReadQueStatus und VCI_ReadQueObj.

Bit 4: CAN0-Running

Bit 5: CAN1-Running

Bit 6: CAN2-Running

Bit 7: CAN3-Running

Status-Bit für die CAN-Controller des Boards (es werden bis max. 4 CAN-Controller pro Board von der VCI unterstützt).

Initialisierte, gestartete und korrekt arbeitende CAN-Controller werden auf ´1´ gesetzt. Befindet sich der CAN-Controller im Bus-Off-Status oder Init-Mode, oder trat ein CAN-Data-Overrun oder Remotequeue-Overrun auf, so wird das Bit auf ´0´ gesetzt. Die genaue Ursache muss dann über VCI_ReadCanStatus ermittelt werden. Damit kann sehr schnell ein Überblick über den Zustand der CAN-Controller gewonnen werden, ohne die CAN-Stati lesen zu müssen.

VCI_BRD_STS	*Beschreibung*
~.cpu_load	durchschnittliche CPU-Auslastung in % (0-100).

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

p_sts (out)
Pointer auf den zu lesenden Status.

Returnwert: VCI-Return Codes.

2.7.9 VCI_ResetBoard

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

Beschreibung: Rücksetzen des Interface (Soft- und Hardware). Damit bleibt das Board angemeldet, aber die Kommunikation ist unterbrochen. Danach muss das Board sowie die CAN-Controller wieder neu initialisiert werden.

Parameter: **board_hdl (in)**
Handle des bereits angemeldeten Boards.

Returnwert: VCI-Return Codes.

2.8 Funktionen für CAN-Controller Verwaltung

2.8.1 VCI_ReadCanInfo

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

Beschreibung: Lesen des Typs des CAN-Controllers sowie der eingestellten Parameter entsprechend VCI_CAN_INFO:

VCI_CAN_INFO	*Beschreibung*
~.can_type	Typ des CAN-Controllers entsprechend VCI_CAN_TYPE
~.bt0	eingestellter Wert für das Bit Timing Register 0
~.bt1	eingestellter Wert für das Bit Timing Register 1
~.acc_code	eingestellter Wert für das Acceptance-Code-Register
~.acc_mask	eingestellter Wert für das Acceptance-Mask-Register

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

can_num (in)
Nummer der CAN-Controllers (0..3).

p_info (out)
Pointer auf die Infodaten.

Returnwert: VCI-Return Codes.

2.8.2 VCI_ReadCanStatus

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

Beschreibung: Lesen der Statusinformationen des angegebenen CAN-Controllers.

VCI_CAN_STS	*Beschreibung*
~.sts	Bit-kodierte Informationen zum CAN-Status (1 = true):

Abhängig vom Typ des verwendeten IXXAT PC/CAN-Interfaces kann die Widerspiegelung des real vorhandenen Canstatus nach Aufruf von Funktionen wie VCI_StartCan oder VCI_ResetCan bis zu 100ms verzögert werden. Während dieser Zeit liefert VCI_ReadCanStatus möglicherweise einen „veralteten" Status. Verwenden Sie diese Funktion daher nur zur Statusvisualisierung in Ihrer Anwendung und nicht um die Ausführung von VCI Steuerungsfunktionen zu überprüfen.

Die einzelnen Bits von ~.sts haben folgende Bedeutung:

Bit 0: nicht genutzt,

Bit 1: nicht genutzt,

Bit 2: RemoteQueueOverrun - In der internen Queue zur Bearbeitung der Remoteanforderungen ist ein Überlauf aufgetreten,

Bit 3: CAN-TX-Pending - Es läuft gerade ein Sendevorgang. Dauert dieser Zustand an, ohne dass erneut Daten gesendet werden, so kann der CAN-Controller die Daten nicht absetzen, (Leitungsbruch oder ähnliches)

Bit 4: CAN-Init-Mode - CAN befindet sich im Initialisierungszustand und kann über VCI_StartCan in den Running-Mode gesetzt werden,

Bit 5: CAN-Data-Overrun - Im CAN-Controller (oder in der CAN-Controller nahen Software) ist ein Überlauf von CAN-Nachrichten aufgetreten,

Bit 6: CAN-Error-Warning-Level - Der CAN-Controller hat wegen Störungen auf dem Bus den Error-Warning-Level erreicht,

Bit 7: CAN-Bus-Off-Status, - Der CAN-Controller hat sich wegen Busstörungen ganz vom Bus abgeschaltet.

VCI_CAN_STS	*Beschreibung*
~.bus_load	Reservierte, nicht unterstützte Funktion

Die Bits 4 - 7 werden direkt aus den CAN-Controllern geliefert. (Zu weiteren Informationen zu diesen Bits lesen Sie bitte die Datenblätter zu den CAN-Controllern Phillips 82C200 oder Intel 82527).
Ist im CAN-Controller ein Fehler aufgetreten (Bit 2,5 und 7), so kann dieser Zustand nur über die Funktion VCI_ResetCan verlassen werden.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

can_num (in)
Nummer der CAN-Controllers (0..n).

p_sts (out)
Pointer auf die Statusdaten.

Returnwert: VCI-Return Codes.

2.8.3 VCI_InitCan

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

Beschreibung: Initialisierung der Timing-Register. Die Werte entsprechen den Angaben für den Philips SJA1000. Für andere Controller werden die Werte entsprechend umgesetzt. Der angegebene CAN-Controller wird zu diesem Zweck in den Zustand Init-Mode gesetzt und Muss dann über VCI_StartCan wieder gestartet werden.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

can_num (in)
Nummer der CAN-Controllers (0..3).

bt0 (in)
Wert für das Timing-Register 0.

bt1 (in)
Wert für das Timing-Register 1.

Die Werte für 'bt0' und 'bt1' zur Einstellung der gängigen Übertragungsraten (**bei 16MHz Taktfrequenz am CAN-Controller**) sind in der nachfolgenden Tabelle aufgeführt:

Übertragungsrate 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 mit 11Bit Identifier.

VCI_29B:
Extended CAN Frame Format mit 29Bit Identifier.

VCI_LOW_SPEED:
Low speed Busankopplung (wenn vorhanden).

VCI_TX_ECHO:
Empfang von Sendenachrichten (Self reception).

VCI_TX_PASSIV:
Passive mode of CAN controller („Listen only" mode).

VCI_ERRFRM_DET:
Erkennung von Error-Frames.

Außer VCI_11B und VCI_29B können alle Einstellungen kombiniert werden.

Ein Beispiel für Standard-Modus mit Self-Reception und Error-Frame-Detection:

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

Returnwert: VCI-Return Codes.

2.8.4 VCI_SetAccMask

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

Beschreibung: Setzen der Acceptance-Mask-Register des CAN-Controllers für eine globale Nachrichten-Filterung im 11-Bit- oder 29-Bit-Betrieb. (ggf. wird diese Controller-Funktion durch Software ersetzt). Der Filter arbeitet über alle Identifier-Bits. Er ist ganz geöffnet (0x0UL, 0x0UL), solange diese Funktion nicht ausgeführt wird. Der angegebene CAN-Controller wird zu diesem Zweck in den Zustand Init-Mode gesetzt und muss dann über VCI_StartCan wieder gestartet werden.

Mit den Variablen acc_code und acc_mask können einzelne CAN-IDs oder ganze ID-Gruppen definiert werden.

Beispiele:

- Es soll nur der CAN-ID 100 empfangen werden:
  acc_code = 100 und acc_mask = 0xffffffff
  0xffffffff -> alle Bits von acc_code sind relevant
- Es sollen die CAN-IDs 100-103 empfangen werden:
  acc_code = 100 und acc_mask = 0xfffffffc
  0xfffffffc -> alle Bits von acc_code sind relevant bis auf die untersten beiden (00, 01, 10, 11).

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

can_num (in)
Nummer der CAN-Controllers (0..3).

acc_code (in)
Wert für das Acceptance-Code-Register

acc_mask (in)
Wert für das Acceptance-Mask-Register
(0 - don't care; 1 - relevant)

Returnwert: VCI-Return Codes.

2.8.5 VCI_ResetCan

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

Beschreibung: Rücksetzen des CAN-Controllers und damit Stoppen der Kommunikation über den angegebenen CAN- Controller Außerdem wird über diese Funktion das Statusregister des CAN-Controllers gelöscht sowie die diesem CAN-Controller zugeordneten Queues

und Buffer neu initialisiert.
Der CAN-Controller verliert dabei nicht seine Konfiguration, sondern kann über VCI_StartCan wieder in Betrieb genommen werden.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

can_num (in)
Nummer der CAN-Controllers (0..3).

Returnwert: VCI-Return Codes.

2.8.6 VCI_StartCan

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

Beschreibung: Starten des angegebenen CAN-Controllers

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

can_num (in)
Nummer der CAN-Controllers (0..3).

Returnwert: VCI-Return Codes.

2.9 Funktionen zur Queue und Buffer Konfiguration

2.9.1 VCI_ConfigQueue

Funktion: 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);

Beschreibung: Anlegen einer Transmit- oder Receivequeue. Als Ergebnis wird ein Handle auf die Queue zurückgeliefert, unter dem die Queue angesprochen werden kann.

Anschließend müssen im Fall einer Receive-Queue alle gewünschten CAN-Nachrichten mit VCI_AssignTxQueObj angemeldet werden.

Für Receivequeues existieren 3 unterschiedliche Möglichkeiten der Verarbeitung von Queuenachrichten:

- Anlegen einer Queue für Pollingbetrieb über VCI_ReadQueObj. Dazu werden die Parameter int_limit und int_time auf null gesetzt.
- Anlegen einer Queue zur Interruptverarbeitung von Nachrichten. Dazu wird int_limit sinnvollerweise auf Eins (max. 13 Nachrichten) gesetzt. Der Interrupt-Callbackhandler wird aufgerufen, sobald die Anzahl Nachrichten in der Empfangsqueue int_limit erreicht oder überschreitet. Eine oder mehrere Nachrichten werden hierbei direkt

an den Interrupt-Callbackhandler übergeben.
Der Interrupt-Callbackhandler wird auch aufgerufen, wenn nach Ablauf der int_time Nachrichten empfangen, aber das int_limit nicht erreicht worden ist.
Die Queue kann hierbei nicht über ReadQueObj gelesen werden.
Die Nachrichten müssen über den Pointer kopiert werden.

'VCI_ReadQueObj()' kann hier nicht benutzt werden.
Diese Betriebsart garantiert sehr kurze Reaktionszeiten, ist aber wegen einer relativ geringen Effektivität nicht für größere Datenraten geeignet.

- Anlegen einer Queue mit Eventbetrieb. Der Interrupt-Callbackhandler wird lediglich zur Signalisierung eines Nachrichtenempfangs benutzt. Dies kann in einer Multitasking-Umgebung zum Anstoßen einer Task benutzt werden, die daraufhin die empfangenen CAN-Objekte mittels VCI_ReadQueObj aus der Empfangsqueue ausliest.
  Das int_limit ist für den Eventbetrieb auf einen Wert größer 13 zu setzen. Das Aufrufkriterium für den Interrupt-Callbackhandler ist wieder das Erreichen oder Überschreiten des int_limit oder der Ablauf von int_time bei Nichterreichen des int_limit, obwohl mindestens ein CAN-Objekt empfangen wurde.

  Diese Betriebsart besitzt die höchste Effektivität und wird daher für den Empfang größere Datenmengen bei hoher Datenrate empfohlen.

  Dem Anwender liegt es offen, die Möglichkeiten des Callbackhandlers zu nutzen. Wünscht er keine Benachrichtigung, muss er beim Aufruf der Funktion VCI2_PrepareBoard einen NULL-Zeiger anstatt eines Funktionszeigers übergeben.

  Der Callbackhandler wird im Kapitel 2.2.1näher beschrieben.

  Bei den Parametern zur Zeitangabe muss die vom Interface unterstützte Auflösung beachtet werden.

  Der CAN-Controller, der der Queue zugeordnet ist muss sich für die Konfiguration der Queues im Init-Mode befinden!

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

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

que_type (in)
Queue-Typ (VCI_TX_QUE, VCI_RX_QUE).

que_size (in)
Größe der Queue in CAN- Nachrichten (muss >= 20 sein!)

int_limit (in)
Anzahl CAN-Nachrichten, nach der ein Interrupt ausgelöst wird (Aufruf des Interrupt-Callbackhandlers). Für eine Transmitqueue wird int_limit nicht berücksichtigt und kann auf 0 gesetzt werden.

0 =	Kein Interrupt auslösen.
<= 13	Die empfangenen Nachrichten werden sofort mit dem Interrupt-Callbackhandler übergeben.
>13	Die empfangenen Nachrichten müssen mit Hilfe der Funktion 'VCI_ReadQueObj()' ausgelesen werden.

int_time (in)
Zeit in ms, nach der für eine Receivequeue ein Interrupt ausgelöst wird (Aufruf des Interrupt-Callbackhandlers), wenn 'int_limit' nicht erreicht wird. Die CAN- Nachrichten werden je nach Größe von 'int_limit' direkt mit dem Interrupt-Callbackhandler übertragen oder müssen gepollt werden. Wird für eine Receivequeue für int_time der Wert Null übergeben, so wird dieser intern auf 500ms gesetzt, um eine vollständige CPU-Auslastung zu verhindern.

Bei einer Konfiguration einer Transmitqueue wird dieser Parameter nicht berücksichtigt und kann somit auf 0 gesetzt werden.

ts_res (in)
Gewünschte Auflösung in μs der Nachrichten-Timestamps für eine Receivequeue. Bei einer Konfiguration einer Transmitqueue wird dieser Parameter nicht berücksichtigt und kann somit auf 0 gesetzt werden.

p_que_hdl (out)
Handle der Queue.

Returnwert: VCI-Return Codes.

Beispiel: Pollen einer Empfangsqueue

```
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);
      }
    }
    ...
  }
}
```

Beispiel: Empfangsqueue mit Interruptverarbeitung

```
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 )
    {
      ...
    }
  }
}
```

Beispiel: Empfangsqueue mit Eventbetrieb

```
#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", asObj[i].id);
            }
          } while( 0 < lReadCount )
        }
      } } } }
```


2.9.2 VCI_AssignRxQueObj

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

Beschreibung: Zuordnen / Sperren von Nachrichten zur angegebenen Receivequeue. Über die Maske sind direkt Identifiergruppen definierbar.

Achtung: Im 29-Bit-Betrieb sind nicht beliebig viele Identifier definierbar. Je nach Hardware werden unterschiedliche Filtermechanismen genutzt. Daher ist die Anzahl der definierbaren Filter begrenzt.

Die Benutzung von 'id' und 'mask' wird in 'VCI_SetAccMask' analog erklärt.

Der CAN-Controller, der der Queue zugeordnet ist muss sich für die Konfiguration der Queue im Init-Mode befinden !!

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

que_hdl (in)
Queue-Handle.

mode (in)
Freigeben/Sperren der Nachricht(en)
(VCI_ACCEPT, VCI_REJECT).

id (in)
Identifier der Nachricht(en).

mask (in)
Maske zur Bestimmung der relevanten Identifierbits. (0 - don't care; 1 - relevant)

Returnwert: VCI-Return Codes.

2.9.3 VCI_ResetTimeStamp

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

Beschreibung: Rücksetzen des Timers für die Timestamps der ReceiveQueues.

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

Returnwert: VCI-Return Codes.

2.9.4 VCI_ConfigBuffer

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

Beschreibung: Einrichten eines Receive- oder Remotebuffers. Der Zugriff auf diesen Buffer erfolgt über den zurück gelieferten Handle. Handles werden als aufsteigende Nummern von Null an vergeben (0,1,2,...n).

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

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

type (in)
Receive oder Remotebuffer
(VCI_RX_BUF, VCI_RMT_BUF).

id (in)
Identifier.

p_buf_hdl (out)
Handle auf den Buffer.

Returnwert: VCI-Return Codes.

2.9.5 VCI_ReConfigBuffer

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

Beschreibung: Ändern des Identifiers eines Receive- oder Remotebuffers. Der Zugriff auf diesen Buffer erfolgt über den Handle.

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

buf_hdl (in)
Buffer-Handle.

type (in)
Receive oder Remotebuffer
(VCI_RX_BUF, VCI_RMT_BUF).

id (in)
Identifier.

Returnwert: VCI-Return Codes.

2.10 Empfangen von Nachrichten

2.10.1 VCI_ReadQueStatus

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

Beschreibung: Lesen des Status der angegebenen Queue.

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

que_hdl (in)
Handle der Queue.

Returnwert:

>0	Anzahl der Queueeinträge.
=0	Queue leer (VCI_QUE_EMPTY),
<0	VCI-Return Codes.

2.10.2 VCI_ReadQueObj

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

Beschreibung: Lesen des/der vordersten Eintrages/Einträge einer Receivequeue. Die Anzahl der zu lesenden Einträge wird über ´count´ angegeben. Dabei werden aber nur so viele Einträge gelesen, wie in der Queue vorhanden sind oder durch die Schnittstelle pro Lesevorgang unterstützt werden. Das bedeutet, dass die Queue solange gelesen werden muss, bis als Returnwert VCI_QUE_EMPTY geliefert wird.
Wird im Statusbyte der Nachricht Bit 7 (0x80) gesetzt (= Queue-Overrun), so konnte nach dieser Nachricht keine weitere Nachricht in der Receivequeue eingetragen werden, da diese bereits voll war. Dies bedeutet, dass Nachrichten verlorengegangen sind.

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

que_hdl (in)
Handle der Queue.

count (in)
Maximale Anzahl von Nachrichten, die gelesen werden sollen. (max. = 13)

p_obj (out)
Pointer auf die zu lesende(n) Nachricht(en).

Returnwert:

>0	Anzahl der gelesenen Queueeinträge.
=0	Queue leer (VCI_QUE_EMPTY).
<0	VCI-Return Codes.

2.10.3 VCI_ReadBufStatus

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

Beschreibung: Lesen des Bufferstatus, ohne diesen zu verändern. Der Bufferstatus stellt die Anzahl der Empfangsvorgänge auf diesen Buffer seit dem letzten Lesevorgang dar.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

buf_hdl (in)
Buffer-Handle.

Returnwert:

=0	VCI_OLD	keine Neue Daten empfangen.
>0		Anzahl, wie oft die Daten nach dem letzten Lesevorgang empfangen wurden.
<0		VCI-Return Codes.

2.10.4 VCI_ReadBufData

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

Beschreibung: Lesen der Bufferdaten und Rücksetzen des Bufferstatus. Als Return wird der Status (Anzahl der Empfangsvorgänge seit dem letzen Lesen) geliefert. Wird dieser Wert aufaddiert, so erhält man die absolute Anzahl der Empfangsvorgänge seit Programmstart.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

buf_hdl (in)
Buffer-Handle.

p_data (out)
Pointer auf die zu lesenden Daten.

p_len (out)
Pointer auf die Anzahl der Datenbytes.

Returnwert:

=0	VCI_OLD	keine Neue Daten empfangen.
>0		Anzahl, wie oft die Daten nach dem letzten Lesevorgang empfangen wurden.
<0		VCI-Return Codes.

2.11 Senden von Nachrichten

2.11.1 VCI_TransmitObj

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

Beschreibung: Senden einer CAN-Nachricht über die angegebene Sendequeue. Wird VCI_QUE_FULL zurückgeliefert, so ist die angegebene Sendequeue im Moment voll und der Sendeauftrag muss (später) wiederholt werden. Wird VCI_TX_ERR geliefert, so ist der CAN-Controller nicht in der Lage, Nachrichten zu senden.

Mögliche Ursache sind fehlende Teilnehmer, fehlende Busabschlüsse oder falsche Baudrate. Überprüfen Sie ihren CAN-Anschluss und die Verkabelung.

Für ungültige Parameterwerte, wie zu große ID (>7FFh bei 11-bit Mode; >1FFFFFFFh bei 29-bit Mode) gibt VCI_TransmitObj VCI_PARA_ERR zurück. Die genaue Fehlerursache wird über den Exception Callback angezeigt.

Parameter:

board_hdl (in)
Handle des zuvor angemeldeten Boards.

que_hdl (in)
Queue-Handle.

id (in)
Identifier der Sende-Nachricht.

len (in)
Anzahl der Datenbytes.

p_data (in)
Pointer auf die Sendedaten.

Returnwert: VCI-Return Codes.

2.11.2 VCI_RequestObj

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

Beschreibung: Senden einer Anforderungsnachricht (Remoteframe) über die angegebene Sendequeue. Wird VCI_QUE_FULL zurückgeliefert, so ist die angegebene Sendequeue im Moment voll und der Sendeauftrag muss (später) wiederholt werden. Wird VCI_TX_ERR geliefert, so ist der CAN-Controller nicht in der Lage, Nachrichten zu senden.

Mögliche Ursache sind fehlende Teilnehmer, fehlende Busabschlüsse oder falsche Baudrate. Überprüfen Sie ihren CAN-Anschluss und die Verkabelung.

Für ungültige Parameterwerte, wie zu große ID (>7FFh bei 11-bit Mode; >1FFFFFFFh bei 29-bit Mode) gibt VCI_RequestObj VCI_PARA_ERR zurück. Die genaue Fehlerursache wird über den Exception Callback angezeigt.

Werden vom CAN-Controller keine Remoteframes unterstützt liefert VCI_RequestObj VCI_SUPP_ERR.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

que_hdl (in)
Queue-Handle.

id (in)
Identifier der Sende-Nachricht.

len (in)
Anzahl der Datenbytes.

Returnwert: VCI-Return Codes.

2.11.3 VCI_UpdateBufObj

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

Beschreibung: Aktualisieren von Daten in einen Remotebuffer, die über das CAN-Netzwerk durch einen anderen CAN-Controller angefordert werden können.

Parameter: **board_hdl (in)**
Handle des zuvor angemeldeten Boards.

buf_hdl (in)
Buffer-Handle.

len (in)
Anzahl der Datenbytes.

p_data (in)
Pointer auf die Daten.

Returnwert: VCI_OK, VCI_QUE_ERR, VCI_HWSW_ERR, VCI_SUPP_ERR, VCI_PARA_ERR.

2.12 Verwendete Datentypen

Die genaue Spezifizierung der verwendeten Datentypen entnehmen Sie bitte dem File VCI.H. Nachfolgend erfolgt die Erklärung der wichtigsten Strukturen.

2.12.1 VCI-CAN-Objekt

Das Senden von CAN-Nachrichten über Transmitqueues sowie das Lesen von CAN-Nachrichten aus Receivequeues erfolgt über den Datentyp VCI_CAN_OBJ:

VCI_CAN_OBJ	***Beschreibung***
~.time_stamp	Empfangszeitstempel für Receivequeue-Nachrichten. Die Auflösung wird über die Funktion VCI_ConfigQueue vorgegeben. Bitte beachten Sie, dass es unabhängig von der Formatierung des Zeitstempels nach (2 ^ 32 * TimeStampResolution) zu einem Überlauf kommt (> 12 Stunden). Der Zeitstempel kann über die Funktion VCI_ResetTimeStamp wieder auf null gesetzt werden.
~.id	11/29-Bit-Identifier der CAN-Nachricht (immer rechtsbündig)
~.len	Anzahl der Datenbytes der CAN-Nachricht(0-8 Bytes)
~.rtr	1= Remote-Request (Datenanforderungsnachricht); die nachfolgenden Datenbytes sind dabei ohne Bedeutung 0 = Datenrahmen (Daten)
~.res	nicht genutzt
~.a_data[8]	8-Byte-Array für die Datenbytes der Nachricht
~.sts	Status der Nachricht: 0 = OK; 0x80 = Queue-Overrun (Nach dieser Nachricht konnte keine weitere Nachricht in der Receivequeue eingetragen werden, da diese bereits voll war -> Datenverlust !)

2.12.2 VCI-Board-Informationen

Das Lesen der Board-Informationen erfolgt über die Struktur VCI_BOARD_INFO:

VCI_BOARD_INFO	***Beschreibung***
~.hw_version	Hardware-Version als HEX-Wert (z. B: 0x0100 für V1.00)
~.fw_version	Firmware-Version als HEX-Wert
~.dd_version	Device-Driver-Version als HEX-Wert (nur für PCMCIA-Karten)
~.sw_version	Versionsnummer der PC-Software als HEX-Wert
~.can_num	Anzahl der vom Board unterstützten CAN-Controller
~.time_stamp_res	kleinstmögliche Timestamp Auflösung in 100 nsec
~.timeout_res	kleinstmögliche Timeout Auflösung für die Receivequeues

~.mem_pool_size	Größe der Speicherpools für die Einrichtung von Queues und Buffers (Bei der VCI_V2 immer 0, da nicht mehr relevant !)
~.irq_num	Interruptnummer zur Kommunikation mit dem PC/CAN-Interface
~.board_seg	eingestellte Board-Adresse/Segment/Portnummer
~.serial_num	16-Zeichen-String mit der Seriennummer des Boards
~.str_hw_type	Null-terminierter String mit der Hardware-Kennung

2.12.3 VCI-Board-Status

Das Lesen des Board-Status erfolgt über die Struktur VCI_BRD_STS:

VCI_BRD_STS	***Beschreibung***
~.sts	Bit-kodierte Informationen zum Boardstatus: Bit 0: RxQueue Overrun; in einer konfigurierten Receivequeue ist ein Overrun aufgetreten (Queue war bereits voll und eine weitere Nachricht konnte nicht eingetragen werden.) Weitere Informationen über VCI_ReadQueStatus und VCI_ReadQueObj. Bit 4: CAN0-Running Bit 5: CAN1-Running Bit 6: CAN2-Running Bit 7: CAN3-Running Status-Bit für die CAN-Controller des Boards (es werden bis max. 4 CAN-Controller pro Board unterstützt). Initialisierte, gestartete und korrekt arbeitende CAN-Controller werden auf ´1´ gesetzt. Befindet sich der CAN-Controller im Bus-Off-Status oder Init-Mode, oder trat ein CAN-Data-Overrun oder RemoteQueue-Overrun auf, so wird das Bit auf ´0´ gesetzt. Die genaue Ursache muss dann über VCI_ReadCanStatus ermittelt werden. Damit kann sehr schnell ein Überblick über den Zustand der CAN-Controller gewonnen werden, ohne die CAN-Stati lesen zu müssen.
~.cpu_load	durchschnittliche CPU-Auslastung in % (0-100)

2.12.4 VCI-CAN-Informationen

Das Lesen der CAN-Informationen erfolgt über die Struktur VCI_CAN_INFO:

VCI_CAN_INFO	*Beschreibung*
~.can_type	Typ des CAN-Controllers entsprechend VCI_CAN_TYPE
~.bt0	eingestellter Wert für das Bit Timing Register 0
~.bt1	eingestellter Wert für das Bit Timing Register 1
~.acc_code	eingestellter Wert für das Acceptance-Code-Register
~.acc_mask	eingestellter Wert für das Acceptance-Mask-Register

2.12.5 VCI-CAN-Status

Das Lesen des CAN-Status erfolgt über die Struktur VCI_CAN_STS:

VCI_CAN_STS	*Beschreibung*
~.sts	Bit-kodierte Informationen zum CAN-Status (1 = true): Bit 0: nicht genutzt, Bit 1: nicht genutzt, Bit 2: RemoteQueueOverrun - In der internen Queue zur Bearbeitung der Remoteanforderungen ist ein Überlauf aufgetreten, Bit 3: CAN-TX-Pending - Es läuft gerade ein Sendevorgang. Dauert dieser Zustand an, ohne dass erneut Daten gesendet werden, so kann der CAN-Controller die Daten nicht absetzen, (fehlender Teilnehmer) Bit 4: CAN-Init-Mode - CAN befindet sich im Initialisierungszustand und kann über VCI_StartCan in den Running-Mode gesetzt werden, Bit 5: CAN-Data-Overrun - Im CAN-Controller (oder der CAN-Controller nahen Software) ist ein Überlauf von CAN-Nachrichten aufgetreten, Bit 6: CAN-Error-Warning-Level - Der CAN-Controller hat wegen häufiger Störungen den Error-Warning-Level erreicht, Bit 7: CAN-Bus-Off-Status, - Der CAN-Controller hat sich wegen Busstörungen ganz vom Bus abgeschaltet. Die Bits 4 - 7 werden direkt aus den CAN-Controllern geliefert. (Zu weiteren Informationen zu diesen Bits lesen Sie bitte die Datenblätter zu den CAN-Controllern Phillips 82C200 oder Intel 82527). Ist im CAN-Controller ein Fehler aufgetreten (Bit 2,5 und 7), so kann dieser Zustand nur über die Funktion VCI_ResetCan verlassen werden.
~.bus_load	Buslast in Prozent. Dieses Feature wird nur von aktiven CAN-Interfaces unterstützt. Stuffbits werden nicht mit eingerechnet.

3 Registrierungsfunktionen (XAT*xx*Reg.DLL)

Mit der VCI_V2 wurde die Möglichkeit eingeführt PC/CAN-Interfaces, die über das VCI angesprochen werden können, beim System unter einer **eindeutigen Indexnummer** zu registrieren.

Die VCI_V2 stellt nun mit der XAT*xx*Reg.DLL (*xx* steht für Versionsnummer, z. B. 10) eine Schnittstelle zur Verfügung um an diese Registrierungsinformationen zu gelangen.

Die Schnittstelle zu dieser DLL ist in der gleichnamigen Header-Datei XAT*xx*Reg.h enthalten. Zum Einbinden der XAT*xx*Reg.DLL gelten die gleichen Hinweise wie für die VCI-DLL in Kapitel 4.

Im Folgenden werden nur diejenigen Funktionen der XAT*xx*Reg.DLL beschrieben, die für eine VCI-Applikation notwendig sind, um an Information über die im System registrierten PC/CAN-Interfaces zu kommen.

Die zur Verfügung gestellten Funktionen decken folgende Bereiche ab:

- Auflisten (Enumeration) aller registrierten PC/CAN-Interfaces sowie der zugeordneten Parameter
- Suchen eines bestimmten PC/CAN-Interfaces
- Aufrufen eines Hardware-Auswahldialogs
- Lesen der Konfiguration eines PC/CAN-Interfaces
- Auswahl/Abfragen eines systemweiten Default-PC/CAN-Interfaces

3.1 Typdefinitionen der Callbackhandler

Die XATxxReg.DLL benutzt den Callback-Mechanismus um alle verfügbaren Informationen über registrierte PC/CAN-Interfaces auszulesen.

3.1.1 Callback zum Auflisten der registrierten PC/CAN-Interfaces

An diese Funktion werden die registrierten PC/CAN-Interfaces nach Aufruf der Funktion XAT_EnumHwEntry übergeben.

Typdefinition: 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)**
Art des Eintrags.

- 0 -> Hardware-Eintrag
- 1 -> Hardware -Parameter

hw_key (in)
Eindeutige Indexnummer, unter der das PC/CAN-Interface beim System registriert ist.

name (in)
Für Hardware-Parameter:
Name des Eintrags

value (in)
Für Hardware-Parameter:
Wert des Eintrags

valuehex (in)
Für Hardware-Parameter:
Hex-Wert des Eintrags

vp_context (in)
Void* auf den Kontext , der in der Funktion XAT_EnumHwEntry übergeben wurde.

Returnwerte: keine

3.2 Funktionsdefinitionen

3.2.1 XAT_SelectHardware

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

Beschreibung: Zeigt einen Dialog zum Auswählen der PC/CAN-Interfaces an. Die vom Benutzer ausgewählte Konfiguration wird in einer Struktur auf die der Parameter pConfig zeigt, abgelegt.

Parameter: **hwndOwner (in)**
Window-Handle des Elternfensters des Dialogs. Normalerweise wird hier der Handle des Hauptfensters der Applikation übergeben.

pConfig (in/out)
Zeiger auf eine Datenstruktur, in die die vom Benutzer ausgewählte Boardkonfiguration geschrieben wird.

Returnwert: 0 -> Benutzer hat CANCEL gedrückt
1 -> Benutzer hat OK-Button gedrückt
-1 -> Fehler (GetLastError()-Funktion verwenden für erweiterten Fehlercode)

Beispiel:

```
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

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

Beschreibung: Liest die Konfiguration des PC/CAN-Interfaces, das unter der eindeutigen Indexnummer dw_key beim System registriert ist. Die Konfiguration wird in der Struktur auf die der Zeiger pConfig zeigt abgelegt.

Parameter: **dw_key (in)**
Eindeutige Indexnummer, unter der das PC/CAN-Interface beim System registriert ist.

pConfig (out)
Zeiger auf eine Datenstruktur, in die die Boardkonfiguration geschrieben wird..

Returnwert: ERROR_SUCCESS – kein Fehler
andernfalls HRESULT Fehlercode

Beispiel:

```
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

Funktion: HRESULT XATREG_CALLATTR **XAT_EnumHwEntry**
(XATREG_ENUM_CALLBACK fp_callback, void * vp_context);

Beschreibung: Enumeriert alle registrierten IXXAT PC/CAN-Interfaces. Für jeden Eintrag wird die im Parameter fp_callback übergebene Callback-Funktion aufgerufen.

Parameter: **fp_callback (in)**
Pointer auf die Callback-Funktion, die für jeden Eintrag aufgerufen wird.

vp_context (in)
Optionaler Kontext. Wird an die Callback-Funktion weitergereicht.

Returnwert: ERROR_SUCCESS – kein Fehler
andernfalls HRESULT Fehlercode

Beispiel:

```
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

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

Beschreibung: Suche nach einem bestimmten registrierten PC/CAN-Interface. Die Funktion XAT_FindHwEntry stellt hierzu verschiedene Sucharten zur Verfügung, die über den Parameter b_typ ausgewählt werden können:

Parameter: **b_typ (in)**
Der Parameter b_typ entscheidet über die Art der Suche, die durchgeführt werden soll:

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION
  Suche eines Boards (eindeutige Board-Indexnummer) anhand dessen Boardtyps und Boardtyp bezogenen Index (z.B. Suche nach dem zweiten registrierten USB-to-CAN).
- XATREG_FIND_RELATIVE_BTYPE_POSITION
  Suche nach dem Boardtyp bezogenen Index eines bestimmten Boards anhand dessen eindeutiger Board-Indexnummer (z.B. Das wievielte registrierte USB-to-CAN ist das spezifizierte Board?)
- XATREG_FIND_ADDRESS
  Suche nach dem Board mit der übergebenen Adresse. Diese Suche ist ausschließlich für ISA-Karten sinnvoll.
- XATREG_FIND_ENTRY_WITH_VALUE
  Such nach einem Board anhand dessen Boardtyp und einer Boardparameter/Boardparameterwert-Kombination (z.B. Suche nach der tinCAN mit IRQ 15). Die für den jeweiligen Boardtyp verfügbaren Parameter sind im IXXAT Interfaces Applet in der Systemsteuerung ersichtlich.

p_dw_key (in/out)

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION:
  (out) Liefert den eindeutigen Index des gefundenen Boards zurück.
- XATREG_FIND_RELATIVE_BTYPE_POSITION
  (in) Eindeutiger Index des Boards für das der Boardtyp bezogenen Index gesucht wird.
- XATREG_FIND_ADDRESS
  (out) Liefert den eindeutigen Index des gefundenen Boards zurück.

- XATREG_FIND_ENTRY_WITH_VALUE
  (out) Liefert den eindeutigen Index des gefundenen Boards zurück.

p_i_boardtyp (in/out)

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION:
  (in) Typ des gesuchten Boards.
- XATREG_FIND_RELATIVE_BTYPE_POSITION
  (in) Typ des gesuchten Boards.
  (out) Boardtyp bezogener Index des gefundenen Boards.
- XATREG_FIND_ADDRESS
  (in) Typ des gesuchten Boards.
- XATREG_FIND_ENTRY_WITH_VALUE
  (in) Typ des gesuchten Boards.

ca_entryname (in)

Wird ausschließlich für XATREG_FIND_ENTRY_WITH_VALUE ausgewertet und bestimmt die Bezeichnung des in dw_arg übergebenen Wert des Boardparameters.

dw_arg (in)

- XATREG_FIND_BOARD_AT_RELATIVE_POSITION:
  (in) Boardtyp bezogener Index des gesuchten Boards.
- XATREG_FIND_RELATIVE_BTYPE_POSITION
  Parameter nicht relevant.
- XATREG_FIND_ADDRESS
  (in) Adresse des gesuchten Boards.
- XATREG_FIND_ENTRY_WITH_VALUE
  (in) Wert des über ca_entryname bezeichneten Boardparameters.

Returnwert: ERROR_SUCCESS – kein Fehler
andernfalls HRESULT Fehlercode

Beispiel für XATREG_FIND_BOARD_AT_RELATIVE_POSITION:

Suchen des zweiten registrierten USB-to-CAN

```
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 )
                                  , ...);
    ...
  }
}
```

Beispiel für XATREG_FIND_RELATIVE_BTYPE_POSITION:

Das wievielte registrierte USB-to-CAN ist mein aktuell benutztes?

```
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 );
}
```


Beispiel für XATREG_FIND_ADDRESS:
Suchen der installierten iPC-I 320 mit der Adresse 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.
}
```

Beispiel für XATREG_FIND_ENTRY_WITH_VALUE:
Suchen der installierten tinCAN mit 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

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

Beschreibung: Setzt den Default-Hardwareeintrag auf das PC/CAN-Interface mit der Indexnummer dw_key.

Parameter: **dw_key (in/out)**
Indexnummer des PC/CAN-Interface.

Returnwert: ERROR_SUCCESS – kein Fehler
andernfalls HRESULT Fehlercode

3.2.6 XAT_GetDefaultHwEntry

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

Beschreibung: Bestimmt den Default-Hardwareeintrag.

Parameter: **p_dw_key (in/out)**
Zeiger auf ein DWORD in dem die Indexnummer des PC/CAN-Interfaces abgelegt wird.

Returnwert: ERROR_SUCCESS – kein Fehler
andernfalls HRESULT Fehlercode

Beispiel:

```
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)
                                    ...);
      ...
    }
  }
```

3.3 Verwendete Datentypen

3.3.1 XAT_BoardCFG

Das Lesen der Informationen über registrierte PC/CAN-Interfaces erfolgt über die Struktur XAT_BoardCFG:

XAT_BoardCFG	*Beschreibung*
~.board_no	Eindeutige Indexnummer
~.board_type	Typ des PC/CAN-Interfaces
~.sz_brd_name[255]	Name
~.sz_manufacturer[50]	Hersteller
~.sz_brd_info[50];	Zusatzinformation
~.sz_CardAddString[255];	Kartenspezifische Information

3.3.2 HRESULT Fehlercodes

Da die Funktionen der XATxxReg.DLL auf den Registry-Funktionen von Microsoft aufsetzen, wird bei einem Fehler der Fehlercode der Registry-Funktionen zurückgegeben. Dieser Fehler kann mit der Win32-API-Funktion FormatMessage() in lesbaren Text umgewandelt werden.

4 Hinweise zur Verwendung der VCI-DLLs

Das Virtual CAN Interface für Windows ist als Dynamic Link Library (DLL) implementiert.

- die DLL wird nicht wie eine normale C-Bibliothek eingebunden, sondern zur Laufzeit der Applikation geladen und dynamisch mit dieser verbunden; die Funktionen der DLL befinden sich also in einem eigenen kompilierten Modul und müssen in einer bestimmten Weise eingebunden werden; das Einbinden wird in Abschnitt 3.1 erklärt.
- die Funktion **VCI_Init()** sollte unter Windows in normalem Betrieb nicht genutzt werden; für die Entwicklung in einer Interpreterumgebung kann es jedoch hilfreich sein mit **VCI_Init()** das VCI explizit zurückzusetzen; dies sollte aber nicht für die Releaseversion der Applikation gelten; dort muss 'VCI_CancelBoard' benutzt werden. Siehe auch die Beschreibung der Funktion VCI_Init().

4.1 Allgemeine Hinweise

In der Installation der VCI_V2 sind Header-Dateien und Beispielprogramme für folgende Entwicklungsumgebungen enthalten:

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

Es ist auch möglich die VCI mit anderen Entwicklungsumgebungen anzusprechen, wir verweisen hier auf die Dokumentation des jeweiligen Systems. Um die VCI in diesem Fall statisch zu linken brauchen Sie eine Importbibliothek, die zum ihrer Entwicklungsumgebung kompatibel sein muss. Meist werden kleine Tools mitgeliefert, mit dem die Funktionssignaturen der DLL ausgelesen, und die Importbibliothek manuell erstellt werden kann. Ist dies nicht möglich, bleibt immer noch die Möglichkeit die VCI Funktionen dynamisch zu importieren (siehe Kap. 4.2.2).

Benutzer von Script-Sprachen und Visualisierungs-Software (z.B. Labview) sollten herausfinden, inwieweit die verwendete Software mit COM-Komponenten zurechtkommt, und den VCIWrapper benutzen (siehe Kap. 4.3).

Performancedaten zu den verschiedenen IXXAT CAN-Interfaces finden Sie auf unserer Website http://www.ixxat.de.

4.2 Einbinden der DLL in eine Applikation

Die Einbindung der DLL kann auf verschiedene Weise erfolgen:

- Impliziter Import über Importbibliothek
- Dynamischer Import

Der Header 'VCI2.H' beinhaltet die Prototypen für die exportierbaren Funktionen.

4.2.1 Impliziter Import während des Linkens

Die DLL kann durch Einfügen der Import-Library in ein Projekt-File der Applikation eingebunden werden. Die Import-Library trägt den gleichen Namen wie die DLL mit der Dateierweiterung ".LIB". In ihr befinden sich die Einträge, die der Linker dazu nutzt, eine "Relocation Table" zu erstellen. Hier werden während der Laufzeit die Adressen der Funktionen der DLL eingetragen. Bei diesem Verfahren wird die Bibliothek während des Startens der Applikation geladen. In der Installation sind Libraries für Microsoft Visual C++ 5.0, Microsoft Visual C++ 6.0 sowie Borland C++ Builder enthalten. Mittels der ebenfalls in der Installation enthaltenen Modul-Definition-Datei (Endung ".DEF") können Import-Libraries für andere Compiler erzeugt werden. Wie sie hierzu verfahren entnehmen Sie bitte der Dokumentation ihrer Entwicklungsumgebung.

4.2.2 Dynamischer Import während der Laufzeit

Beim dynamischen Import wird die DLL nicht schon beim Start der Applikation geladen, sondern erst dann, wenn sie tatsächlich benötigt wird. Danach kann die DLL ebenso wieder geschlossen werden, ohne gleichzeitig die Applikation zu beenden. Dieser Import wird von Hand in der Applikation selbst vorgenommen.

Dazu werden die Windows-API-Funktionen

- LoadLibrary
- GetProcAdress
- FreeLibrary

verwendet. Nähere Informationen finden Sie in der Dokumentation der Windows-API. Das folgende Codefragment verdeutlicht die Vorgehensweise beim dynamischen Laden:

```
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 Hinweis für VisualBasic–Entwickler

Die VCI war ursprünglich nur als C-API verfügbar. Es gibt zwar Möglichkeiten DLL-Funktionen von VisualBasic aus zu nutzen, doch treten dabei immer wieder Probleme auf. Dies hat folgende Gründe:

- Der Debugger von VisualBasic ist nicht Multithreading ausgelegt. Da die VCI aber intern mehrere Threads benutzt, stürzt die IDE von VisualBasic im Debug-Modus öfters ab.
- Es können Probleme mit dem Alignment von Strukturen auftreten, die an Funktionen der VCI-DLL übergeben werden.
- Callback-Funktionen werden durch VisualBasic nur eingeschränkt genutzt werden. Die Verwendung von Callbacks wurde von der VisualBasic Version 5.0 zur Version 6.0 noch weiter limitiert.

Alle diese Probleme kann man durch die Verwendung einer COM-Komponente vermeiden, das die Zugriffe auf die VCI kapselt. Die IXXAT Automation GmbH hat bereits eine solche VCIWrapper-Komponente entwickelt.

Der VCIWrapper kann über den IXXAT Webserver unter http://www.ixxat.de heruntergeladen werden. In der Installation sind ein Benutzerhandbuch sowie ein Beispielprogramm enthalten, dem Sie die Verwendung des VCIWrappers entnehmen können.