USB-CANmodul

System Manual
Version 2.05

Edition September 2019

Document No.: L-487e_02_05
## Document History

<table>
<thead>
<tr>
<th>Date/Version</th>
<th>Section/Table/Figure</th>
<th>Changes</th>
<th>Author/Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>13th June 2016</td>
<td>All</td>
<td>Continued from L-487e_30 (as V1.30). New style sheet for manuals. Obsolete hardware versions removed.</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>V2.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09th August 2016</td>
<td>2.1.3 2.1.4 2.2 2.3</td>
<td>Sections added for Multiport devices USB-CANmodul8 and USB-CANmodul16 Voltage limitations added for the CAN connector. Termination resistors of Multiport devices are changeable via switches at the front panel.</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>V2.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02nd February 2017</td>
<td>2.3</td>
<td>Section added explaining the CAN-channel assignment of Multiport devices.</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>V2.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17th Mai 2017</td>
<td>3.1 3.2</td>
<td>Add note for CAN cable to be used in case of highly electromagnetic disturbed applications. Add note for handling USB reconnections of logical devices in an application.</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>V2.02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 1</td>
<td></td>
<td>List of available hardware variants updated</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>3.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 14</td>
<td></td>
<td>Add constants for usbcani23.sys and usbcanlex.sys. Add availability of the constants according to the installed driver version.</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>Table 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Figure 11</td>
<td></td>
<td>Add figure showing the front and back view of the USB-CANmodul16</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>Figure 2 and</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Figure 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Figure 9 and</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.1 to 2.1.4,</td>
<td></td>
<td></td>
<td>Dietzsch</td>
</tr>
<tr>
<td>2.2 to 2.6 and</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.1 to 3.10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17th July 2019</td>
<td>2.1.1</td>
<td>Removed the technical property of the 120 ohm termination register.</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>V2.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26th September</td>
<td>3.1</td>
<td></td>
<td>Dietzsch</td>
</tr>
<tr>
<td>2019 V2.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29th September</td>
<td>2.1</td>
<td></td>
<td>Dietzsch +</td>
</tr>
<tr>
<td>2017 V2.03</td>
<td></td>
<td></td>
<td>KüNZel</td>
</tr>
<tr>
<td>29th August 2018</td>
<td>3.1</td>
<td></td>
<td>Dietzsch</td>
</tr>
<tr>
<td>V2.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17th July 2019</td>
<td>2.1.1</td>
<td></td>
<td>Dietzsch</td>
</tr>
<tr>
<td>V2.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26th September</td>
<td>3.1</td>
<td></td>
<td>Dietzsch</td>
</tr>
<tr>
<td>2019 V2.05</td>
<td>All</td>
<td>Changed “SYS TEC electronic GmbH” to “SYS TEC electronic AG”</td>
<td>Dietzsch</td>
</tr>
<tr>
<td>2.1.5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Product names used in this manual which are also registered trademarks have not been marked extra. The missing © mark does not imply that the trade name is unregistered. Nor is it possible to determine the existence of any patents or protection of inventions on the basis of the names used.

The information in this manual has been carefully checked and is believed to be accurate. However, it is expressly stated that SYS TEC electronic AG does not assume warranty or legal responsibility or any liability for consequential damages which result from the use or contents of this user manual. The information contained in this manual can be changed without prior notice. Therefore, SYS TEC electronic AG shall not accept any obligation.

Furthermore, it is expressly stated that SYS TEC electronic AG does not assume warranty or legal responsibility or any liability for consequential damages which result from incorrect use of the hard or software. The layout or design of the hardware can also be changed without prior notice. Therefore, SYS TEC electronic AG shall not accept any obligation.

© Copyright 2019 SYS TEC electronic AG, D-08468 Heinsdorfergrund

All rights reserved. No part of this manual may be reproduced, processed, copied or distributed in any way without the express prior written permission of SYS TEC electronic GmbH.

<table>
<thead>
<tr>
<th>Contact</th>
<th>Direct</th>
<th>Your Local Distributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address:</td>
<td>SYS TEC electronic AG Am Windrad 2 D-08468 Heinsdorfergrund GERMANY</td>
<td>Please find a list of our distributors under:</td>
</tr>
<tr>
<td>Ordering Information:</td>
<td>+49 (0) 37 65 / 38 600-0 <a href="mailto:info@systec-electronic.com">info@systec-electronic.com</a></td>
<td><a href="https://www.systec-electronic.com/distributors">https://www.systec-electronic.com/distributors</a></td>
</tr>
<tr>
<td>Technical Support:</td>
<td>+49 (0) 37 65 / 38 600-0 <a href="mailto:support@systec-electronic.com">support@systec-electronic.com</a></td>
<td></td>
</tr>
<tr>
<td>Fax:</td>
<td>+49 (0) 37 65 / 38 600 4100</td>
<td></td>
</tr>
<tr>
<td>Web Site:</td>
<td><a href="https://www.systec-electronic.com/">https://www.systec-electronic.com/</a></td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

1 Introduction ................................................................................................................................. 8

2 Hardware Description .................................................................................................................. 9
  2.1 Hardware Variants ................................................................................................................... 9
    2.1.1 The USB-CANmodul1 ..................................................................................................... 10
    2.1.2 The USB-CANmodul2 ..................................................................................................... 12
    2.1.3 The USB-CANmodul8 ..................................................................................................... 15
    2.1.4 The USB-CANmodul16 .................................................................................................... 18
    2.1.5 Legacy devices ............................................................................................................... 20
  2.2 CAN connector ......................................................................................................................... 21
  2.3 Termination resistor for high-speed CAN Transceiver ............................................................ 21
  2.4 CAN-port with low-speed CAN Transceiver .......................................................................... 23
  2.5 Expansion Port ....................................................................................................................... 24
  2.6 LEDs on the USB-CANmodul ................................................................................................. 26

3 Getting Started ............................................................................................................................ 28
  3.1 System requirements ............................................................................................................... 28
  3.2 Installation of the driver under Windows-OS ........................................................................ 29
  3.3 Updating an existing installation ............................................................................................ 34
  3.4 Verifying the Device Installation ........................................................................................... 36
  3.5 Device Number Allocation ..................................................................................................... 37
  3.6 Connection to a CAN Network ............................................................................................... 38
  3.7 Starting PCANView (USBCAN) ............................................................................................. 38
  3.8 Creating a debug file from DLL ............................................................................................. 42
  3.9 Activation of the network driver ............................................................................................. 43
  3.10 Completely uninstall the driver ............................................................................................ 43

4 Software Support for Windows OS ............................................................................................. 45
  4.1 File Structure ........................................................................................................................... 45
  4.2 Tools for the USB-CANmodul ................................................................................................. 46
    4.2.1 USB-CANmodul Control for Windows ........................................................................... 46
    4.2.2 PCANView (USBCAN) for Windows ................................................................................. 47
    4.2.3 ConvertUcanTrace for Windows ...................................................................................... 51
  4.3 Description of the USBCAN32.DLL / USBCAN64.DLL ......................................................... 52
    4.3.1 The concept of the DLL .................................................................................................. 52
    4.3.2 API Functions of the DLL ............................................................................................... 55
      4.3.2.1 General API functions ............................................................................................... 55
      4.3.2.2 API Functions for automatic transmission ............................................................... 96
      4.3.2.3 API Functions for the CAN port .............................................................................. 39
      4.3.2.4 API Functions for the expansion port ..................................................................... 103
    4.3.3 Error codes of the API functions ...................................................................................... 105
  4.4 Baud Rate Configuration ........................................................................................................... 110
    4.4.1 Baud Rate Configuration for first and second generation USB-CANmodul .................. 110
    4.4.2 Baud Rate Configuration for third generation USB-CANmodul ................................. 112
    4.4.3 Baud Rate Configuration for fourth generation USB-CANmodul ............................... 117
    4.4.4 Use of user-defined CAN baud rates ............................................................................ 120
  4.5 CAN Messages Filter Function ............................................................................................... 121
  4.6 Using multiple CAN-channels ................................................................................................. 124
  4.7 Using the Callback Functions ................................................................................................. 125
    4.7.1 Connect Control Callback Function ............................................................................... 125
    4.7.2 Event Callback Function ............................................................................................... 127
    4.7.3 Enumeration Callback Function .................................................................................... 130
    4.7.4 Assignment of CAN-channels of Multiport devices ..................................................... 133

5 Software support for Linux OS ................................................................................................... 135

6 Known issues .............................................................................................................................. 136

Index ............................................................................................................................................. 137
List of Tables

Table 1: Overview of hardware variants .......................................................... 9
Table 2: Pinout of the power input connector of order code 3204010-XX .......... 17
Table 3: Overview of supported legacy devices .............................................. 20
Table 3: Pinout of the CAN DB-9 Plug .......................................................... 21
Table 4: Recommended cable parameters ......................................................... 22
Table 5: Signals available for low-speed or single-wire CAN port .................... 23
Table 6: Control input for single-wire CAN port ............................................. 23
Table 7: Expansion Port Pin Assignment on USB-CANmodul2 ......................... 24
Table 8: Properties of port expansion on USB-CANmodul2 ............................. 24
Table 9: States of the LEDs on the USB-CANmodul devices ............................ 27
Table 11: Available API functions according the software state ....................... 54
Table 12: Constants for the debug level passed to function UcanSetDebugMode() ................................................................. 56
Table 14: Constants for the type of version information for function UcanGetVersionEx() ................................................................. 58
Table 14: Constants for selecting the CAN mode ............................................. 69
Table 15: Constants for Reset Flags ............................................................... 71
Table 16: Constants as pre-defined combinations for Reset Flags .................... 72
Table 17: Constants for Product-Code / Hardware-Type ................................... 77
Table 18: Constants for CAN error status ....................................................... 84
Table 19: Constants for general error status .................................................... 84
Table 20: Constants for the CAN frame format ............................................... 89
Table 21: Constants for the flags parameter in function UcanGetMsgPending() ................................................................. 94
Table 22: Constants for the flags parameter in function UcanEnableCyclicCanMsg() ................................................................. 99
Table 23: Constants for low-speed CAN port .................................................. 100
Table 24: Error codes of the API functions ..................................................... 105
Table 25: Constants for CAN baud rates for first and second generation .......... 110
Table 26: Constants for CAN baud rates for third generation .......................... 113
Table 27: Constants for CAN baud rates for fourth generation (CPU frequency = 96 MHz) ................................................................. 117
Table 28: Constants for CAN baud rates for fourth generation (CPU frequency = 120 MHz) ................................................................. 117
Table 29: Examples for user-defined CAN baud rates ...................................... 120
Table 30: CAN message filter mechanism for only accepted CAN messages ........ 121
Table 31: Constants for acceptance filter for receiving all CAN messages .......... 123
Table 32: Constants for CAN-channel selection .............................................. 124
Table 33: Constants for the event informed with the connect control callback functions ............ 125
Table 34: Constants for the event informed with the event callback functions .... 127
Table 35: Assignment of CAN-channels of Multiport devices ........................... 133
List of Figures

Figure 1: Top view of the USB-CANmodul1 ......................................................... 10
Figure 2: Product sticker of the USB-CANmodul1 ............................................. 11
Figure 3: Top view of the USB-CANmodul2 ...................................................... 12
Figure 4: Product sticker of the USB-CANmodul2 ............................................. 13
Figure 5: Position of expansion plug and jumpers on USB-CANmodul2 .......... 14
Figure 6: Internal structure of the USB-CANmodul8 ....................................... 15
Figure 7: Front and back view of the USB-CANmodul8 in table case ............... 15
Figure 8: Product sticker of the USB-CANmodul8 .......................................... 16
Figure 9: Power input of order code 3204010-XX .......................................... 16
Figure 10: Internal structure of the USB-CANmodul16 .................................... 18
Figure 11: Front and back view of the USB-CANmodul16 ................................ 18
Figure 12: Product sticker of the USB-CANmodul16 ....................................... 19
Figure 13: Termination resisters on CAN bus .................................................. 21
Figure 14: Simple example circuit for Expansion Port ...................................... 25
Figure 15: Traffic LED after one CAN message on CAN bus ......................... 26
Figure 16: Traffic LED after more CAN messages on CAN bus ....................... 26
Figure 17: Blink cycles of the state LED .......................................................... 26
Figure 18: USB-CANmodul Control Check for Update ................................... 34
Figure 19: USB-CANmodul Control Start Download ...................................... 35
Figure 20: Updating an existing installation .................................................... 35
Figure 21: Device Manager with the USB-CANmodul ...................................... 36
Figure 22: USB-CANmodul Control tab-sheet Hardware ................................. 37
Figure 23: Device number changing dialog box ............................................ 37
Figure 24: Dialog box with hardware configuration in PCANView .................... 38
Figure 25: Dialog box for message filter configuration in PCANView ............... 39
Figure 26: PCANView (USBCAN) main window ............................................ 40
Figure 27: Dialog box for entering a new transmit message ......................... 40
Figure 28: Transmission and reception of CAN messages in PCANView .......... 41
Figure 29: Debug settings in USB-CANmodul Control .................................. 42
Figure 30: Activation of higher performance .................................................. 46
Figure 31: Dialog box for manipulating the port expansion and the CAN port .... 46
Figure 32: Extended system menu of PCANView ......................................... 46
Figure 33: Configuration of cyclic CAN messages in PCANView (USBCAN) .... 49
Figure 34: Software State Diagram ................................................................. 52
Figure 35: Example for parallel mode of cyclic CAN messages ...................... 96
Figure 36: Example for Sequential mode of cyclic CAN messages ................. 96
Figure 37: Structure of baud rate register BTR0 ............................................. 111
Figure 38: Structure of baud rate register BTR1 ............................................. 111
Figure 39: General structure of one bit on the CAN-bus (source: NXP SJ1000 manual) .......................................................... 112
Figure 40: Structure of baud rate register dwBaudrate of third generation modules .......................................................... 113
Figure 41: General structure of one bit on the CAN-bus (source: Atmel AT91SAM7A3 manual) .......................................................... 114
Figure 42: Structure of baud rate register dwBaudrate for fourth generation modules .......................................................... 118
Figure 43: General structure of one bit on the CAN-bus (source: STM32F205xx manual) .......................................................... 118
Figure 44: CAN message filter mechanism used within the USB-CANmodul .......................................................... 121
Figure 45: CAN message filter corresponding bits for 11-bit CAN-ID .................. 121
Figure 46: CAN message filter corresponding bits for 29-bit CAN-ID ............... 122
1 Introduction

Unveiled in 1995, the Universal Serial Bus (USB) connectivity standard provides a simple and convenient way to connect various peripheral devices to a host-PC. It will replace a wide variety of serial and parallel connections. The USB standard allows up to 127 devices to be connected to the PC without using multiple connector types, without interrupt conflicts (IRQs), hardware address adjustments (jumpers) or channel changes (DMA). USB provides powerful true hot plug-and-play capability; i.e., dynamic attach and recognition for new devices. It allows the user to work with those devices immediately without restarting the operating system.

The USB-CANmodul takes advantage of this communication standard and provides an easy to use portal from a host-PC to a CAN network. Connecting the USB-CANmodul to the host-PC is simple. The included USB cable supports the two types of USB connectors, type A and type B. The type A plug connects to the host computer or an upstream hub. Type B plug connects downstream to the USB-CANmodul. The USB interface enables data transfer with a rate of up to 12 MBit/s. With a uniform connector for all device types, the system is absolutely user friendly.

Once the USB-CANmodul is connected to the host-PC, the operating system reads the configuration data and automatically loads the device driver. All CAN messages are transferred transparently through the USB Bus. CAN Baud Rates of up to 1 mbps are supported. The transmitted and received CAN messages are buffered by the USB-CANmodul. The device supports CAN messages according to CAN 2.0A and 2.0B specifications (11- and 29-Bit identifiers). Connection to the CAN bus meets the CiA Standard DS 102 (DB-9) and features optional optical isolation of the CAN signals.

Drivers for LabView (contributed), Windows 7, 8 and 8.1 as well as Linux are provided for the USB-CANmodul. The USB configuration tool for Windows enables connectivity and management of more than one device on the USB bus. This USB network is configured using device numbers which are assigned by the user and are stored in an EEPROM. The functions for data exchange with the USB-CAN application are available through a DLL (Dynamic Linked Library). The enclosed demo program shows the easy handling of the DLL API functions.

For more information, optional products, updates et cetera, we recommend you to visit our website: http://www.systec-electronic.com. The content of this website is updated periodically and provides to you downloads of the latest software releases and manual versions.

The document describes all hardware variants of the USB-CANmodul, the installation of the device drivers and the API interface. There are no additional manuals needed for the USB-CANmodul.
2 Hardware Description

2.1 Hardware Variants

*Table 1* lists the available hardware variants this manual is related to. All these hardware variants belong to the fourth hardware generation (G4). Older hardware variants are not documented within the scope of this manual - they are described in older manual versions of L-487.

Since driver version V6.05 the legacy USB-CANmodul devices are supported too. Refer to section 2.1.5.

*Table 1:* Overview of hardware variants

<table>
<thead>
<tr>
<th>Order code / name</th>
<th>CH0</th>
<th>CH1</th>
<th>IO port</th>
<th>Housing</th>
<th>Galv. isolation</th>
<th>Power supply</th>
<th>Max. current over USB</th>
</tr>
</thead>
<tbody>
<tr>
<td>3204001-01 <em>USB-CANmodul1</em></td>
<td>82C251</td>
<td>-</td>
<td>No</td>
<td>Small table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>150mA</td>
</tr>
<tr>
<td>3204003-01 <em>USB-CANmodul2</em></td>
<td>82C251</td>
<td>82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
</tr>
<tr>
<td>3204007-01 <em>USB-CANmodul2</em></td>
<td>82C251</td>
<td>82C251</td>
<td>Yes</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
</tr>
<tr>
<td>3204008-01 <em>USB-CANmodul2</em></td>
<td>AU5790</td>
<td>82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
</tr>
<tr>
<td>3204019-01 <em>USB-CANmodul2</em></td>
<td>TJA1054</td>
<td>82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
</tr>
<tr>
<td>3204004-01 <em>USB-CANmodul8</em></td>
<td>82C251</td>
<td>82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>External 100 - 240 VAC</td>
<td>0mA</td>
</tr>
<tr>
<td>3204010-01 <em>USB-CANmodul8</em></td>
<td>82C251</td>
<td>82C251 up to CH7</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>External 9 - 32 VDC</td>
<td>0mA</td>
</tr>
<tr>
<td>3304000-01 <em>USB-CANmodul8</em></td>
<td>82C251</td>
<td>82C251 up to CH7</td>
<td>No¹</td>
<td>Open frame</td>
<td>Yes</td>
<td>External 100 - 240 VAC</td>
<td>0mA</td>
</tr>
<tr>
<td>3404002-01 <em>USB-CANmodul8</em></td>
<td>82C251</td>
<td>82C251 up to CH7</td>
<td>No¹</td>
<td>19” rack-mounted</td>
<td>Yes</td>
<td>External 100 - 240 VAC</td>
<td>0mA</td>
</tr>
<tr>
<td>3404001-01 <em>USB-CANmodul16</em></td>
<td>82C251</td>
<td>82C251 up to CH15</td>
<td>No¹</td>
<td>19” rack-mounted</td>
<td>Yes</td>
<td>External 100 - 240 VAC</td>
<td>0mA</td>
</tr>
</tbody>
</table>

¹ The IO port is available at the PCB but the connector is not available on the case.

---

**Note for EMC:**

In case of highly electromagnetic disturbed applications we advise to use a proper mounting location. Please separate the power and control wires/components to suite the general rules of electrical installation design.
2.1.1 The USB-CANmodul1

The USB-CANmodul1 is a cost optimized variant of the sysWORXX USB-CANmodul series including only one CAN-channel. This device has a galvanic isolation and built in a high-speed CAN transceiver. There is no Expansion Port for connecting digital inputs or outputs.

![Top view of the USB-CANmodul1](image.png)

The modules since revision -01 belong to the fourth generation. All older revisions are obsolete and are not documented within the scope of this manual. To find out the revision number of the USB-CANmodul1 have a look at the sticker at the ground of the case. The number behind the hyphen specifies the revision number of the USB-CANmodul (refer to Figure 2). At older stickers the revision is marked with the prefix “Rev.” at which revision 05 belongs to the fourth generation.

Technical Data:
- More compact case with dimensions of 78x45x18 (LxWxH in mm), weight approx. 40g
- Single CAN interface (ISO 11858-1/2, Standard Frames, Extended Frames, Remote Frames), SUB-D9 connector
- Fast 32-bit MCU, enhanced firmware
- USB bus powered, current consumption max. 150mA
- USB 1.1 Full-Speed (12Mbit/s), compatible to USB 2.0 and USB 3.0, Mini-USB Type B connector
- High-speed CAN transceiver 82C251 (according ISO 11898-2)
- CAN bitrate 10kbps to 1Mbps
- Galvanic isolation
- Operating temperature: -40°C to +85°C

Refer to section 2.2 for information about the pinout of the CAN connectors.
Refer to section 2.3 for information about the termination resistors for high-speed CAN transceivers.
Figure 2: Product sticker of the USB-CANmodul1

(old sticker)

(new sticker)
2.1.2 The USB-CANmodul2

The USB-CANmodul2 is an extended variant of the sysWORXX USB-CANmodul series including two CAN-channels. This device has a galvanic isolation and built in two high-speed CAN transceiver 82C251. There are variants with built in alternatively CAN transceivers at the 1st CAN channel (e.g. TJA1054 or AU5790). But the 2nd CAN channel always has built in the high-speed CAN transceiver 82C251 (refer to Table 1 for detailed information).

There is Expansion Port for connecting digital inputs or outputs. With order number 3204007 you will get an USB-CANmodul2 including an Expansion Port which is described in section 2.5.

![Top view of the USB-CANmodul2](image)

Figure 3: Top view of the USB-CANmodul2

The modules since revision -01 belongs to the fourth generation. All older revisions are obsolete and are not documented within the scope of this manual. To find out the revision number of the USB-CANmodul2 have a look at the sticker at the ground of the case. The number behind the hyphen shows the revision number (refer to Figure 4). At older stickers the revision is marked with the prefix “Rev.” at which revision 03 belongs to the fourth generation.
Technical Data:

- Case dimensions of 100x78x30 (LxWxH in mm), weight approx. 110g
- Two CAN-channels, independently utilizable (ISO 11858-1/2, Standard Frames, Extended Frames, Remote Frames), SUB-D9 connectors
- Fast 32-bit MCU, enhanced firmware
- USB bus powered, current consumption max. 200mA
- USB 1.1 Full-Speed (12Mbit/s), compatible to USB 2.0 and USB 3.0, USB Type B connector
- High-speed CAN transceiver 82C251 for 2nd CAN channel (according ISO 11898-2)
- Alternative CAN transceivers for 1st CAN channel choose-able via order code (e.g. TJA1054 or AU5790 - low-speed or single-wire CAN according ISO 11898-3)
- CAN bitrate 10kbps to 1Mbps for high-speed CAN transceiver
- CAN bitrate 10kbps to 125bps for low-speed CAN transceiver TJA1054
- CAN bitrate 33,3kbps or 83,3bps for single-wire CAN transceiver AU5790
- Galvanic isolation
- 120 ohm termination resistor can be set at PCB via jumper
- Micro controller’s 8-bit user port (I/O with TTL level) provides for customer-specific extensions with order code 3204007
- Operating temperature: -40ºC to +85ºC
Refer to section 2.2 for information about the pinout of the CAN connectors.

Refer to section 2.3 for information about the termination resistors for high-speed CAN transceivers.

Refer to section 2.4 for information about the CAN port for low-speed CAN transceivers.

Refer to section 2.5 for information about the expansion port.

Figure 5: Position of expansion plug and jumpers on USB-CANmodul2
2.1.3 The USB-CANmodul8

The USB-CANmodul8 is a Multiport device with up to 8 CAN channels. This device is structured into 4 logical USBCAN devices with 2 CAN-channels each (four so-called “logical devices”). All 4 logical devices are combined by an USB-hub (see picture below).

![Internal structure of the USB-CANmodul8](image)

Each CAN channel has a galvanic isolation and a built in high-speed CAN transceiver 82C251.

![Front and back view of the USB-CANmodul8 in table case](image)

The modules since revision -01 belongs to the fourth generation. All older revisions are obsolete and are not documented within the scope of this manual. To find out the revision number of the USB-CANmodul8 have a look at the sticker at the background of the case. The number behind the order code shows the revision number (refer to Figure 8).
Figure 8: Product sticker of the USB-CANmodul8

Technical Data:

- Table-case dimensions of 200x225x75 (LxWxH in mm), weight approx. 1200g
- Eight CAN-channels, independently utilizable (ISO 11858-1/2, Standard Frames, Extended Frames, Remote Frames), SUB-D9 connectors
- Fast 32-bit MCU, enhanced firmware
- USB self-powered 100 – 240 VAC 50/60 Hz, max. 25W (optional 9 – 32 VDC)
- USB 1.1 Full-Speed (12Mbit/s), compatible to USB 2.0 and USB 3.0, USB Type B connector
- High-speed CAN transceiver 82C251 (according ISO 11898-2)
- CAN bitrate 10kbps to 1Mbps for high-speed CAN transceiver
- Galvanic isolation
- 120 ohm termination resistor can be set via switch at the front panel of the case
- Operating temperature: 0ºC to +55ºC

Refer to section 2.2 for information about the pinout of the CAN connectors.
Refer to section 2.3 for information about the termination resistors for high-speed CAN transceivers.
Refer to section 2.5 for information about the expansion port.

The order code 3204010 since hardware revision -01 does not have a port for a rubber connector. Instead of this it has a green screw terminal with three pins. Refer to Figure 9 and Table 2 for detailed information.

Figure 9: Power input of order code 3204010-XX
Table 2: Pinout of the power input connector of order code 3204010-XX

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Vcc +9...32VDC (max. 24W)</td>
</tr>
<tr>
<td>2</td>
<td>GND</td>
</tr>
<tr>
<td>3</td>
<td>Earth grounding</td>
</tr>
</tbody>
</table>
2.1.4 The USB-CANmodul16

The USB-CANmodul16 is a Multiport device with up to 16 CAN channels. This device is structured into 8 logical USBCAN devices with 2 CAN-channels each (eight so-called “logical devices”). All 8 logical devices are combined by two USB-hubs (see picture below).

![Internal structure of the USB-CANmodul16](image)

Each CAN channel has a galvanic isolation and a built in high-speed CAN transceiver 82C251.

![Front and back view of the USB-CANmodul16](image)
The modules since revision -01 belong to the fourth generation. All older revisions are obsolete and are not documented within the scope of this manual. To find out the revision number of the USB-CANmodul16 have a look at the sticker at the background of the case. The number behind the order code shows the revision number (refer to Figure 12).

![Product sticker of the USB-CANmodul16](image)

**Figure 12: Product sticker of the USB-CANmodul16**

**Technical Data:**
- Dimensions of 19" rack-mounted case of 250x485x45 (LxWxH in mm), weight approx. 2000g
- Sixteen CAN-channels, independently utilizable (ISO 11858-1/2, Standard Frames, Extended Frames, Remote Frames), SUB-D9 connectors
- Fast 32-bit MCU, enhanced firmware
- USB self-powered 100 – 240 VAC, current consumption max. 500mA
- 2 x USB 1.1 Full-Speed (12Mbit/s), compatible to USB 2.0 and USB 3.0, USB Type B connector
- High-speed CAN transceiver 82C251 (according ISO 11898-2)
- CAN bitrate 10kbps to 1Mbps for high-speed CAN transceiver
- Galvanic isolation
- 120 ohm termination resistor can be set via switch at the front panel of the case
- Operating temperature: 0ºC to +55ºC

Refer to section 2.2 for information about the pinout of the CAN connectors.
Refer to section 2.3 for information about the termination resistors for high-speed CAN transceivers.
Refer to section 2.5 for information about the expansion port.
2.1.5 Legacy devices

Since driver version V6.05 the legacy USB-CANmodul devices are supported too. Table 3 gives a short overview of these legacy devices.

Any hardware properties are equal to the actual devices of the 4th generation. For more information refer to an older version of this manual or contact support@systec-electronic.com.

Table 3: Overview of supported legacy devices

<table>
<thead>
<tr>
<th>Order code / name</th>
<th>CH0</th>
<th>CH1</th>
<th>IO port</th>
<th>Housing</th>
<th>Galv. isolation</th>
<th>Power supply</th>
<th>Max. current over USB</th>
</tr>
</thead>
<tbody>
<tr>
<td>3204000 Rev.01 to Rev.04 USB-CANmodul1</td>
<td>82C251</td>
<td>-</td>
<td>No</td>
<td>Small table case</td>
<td>No</td>
<td>USB powered</td>
<td>110mA</td>
</tr>
<tr>
<td>3204001 Rev.01 to Rev.04 USB-CANmodul1</td>
<td>82C251</td>
<td>-</td>
<td>No</td>
<td>Small table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>110mA</td>
</tr>
<tr>
<td>3204002 Rev. 01 to Rev. 02 USB-CANmodul2</td>
<td>82C251 82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>No</td>
<td>USB powered</td>
<td>200mA</td>
<td></td>
</tr>
<tr>
<td>3204003 Rev. 01 to Rev. 02 USB-CANmodul2</td>
<td>82C251 82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
<td></td>
</tr>
<tr>
<td>3204007 Rev. 01 to Rev. 02 USB-CANmodul2</td>
<td>82C251 82C251</td>
<td>Yes</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
<td></td>
</tr>
<tr>
<td>3204008 Rev. 01 to Rev. 02 USB-CANmodul2</td>
<td>AU5790 82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
<td></td>
</tr>
<tr>
<td>3204019 Rev. 01 to Rev. 02 USB-CANmodul2</td>
<td>TJA1054 82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>USB powered</td>
<td>200mA</td>
<td></td>
</tr>
<tr>
<td>3404000 USB-CANmodul8</td>
<td>82C251 82C251</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>External 100 - 240 VAC</td>
<td>0mA</td>
<td></td>
</tr>
<tr>
<td>3004006 USB-CANmodul16</td>
<td>82C251 82C251 up to CH15</td>
<td>No¹</td>
<td>19&quot; rack-mounted</td>
<td>Yes</td>
<td>External 100 - 240 VAC</td>
<td>0mA</td>
<td></td>
</tr>
<tr>
<td>3404001 USB-CANmodul16</td>
<td>82C251 82C251 up to CH15</td>
<td>No¹</td>
<td>Table case</td>
<td>Yes</td>
<td>External 100 - 240 VAC</td>
<td>0mA</td>
<td></td>
</tr>
</tbody>
</table>
2.2 CAN connector

No external CAN supply voltage is necessary for all USB-CANmodul types with high-speed CAN transceivers. The low-speed versions require an external supply voltage for the CAN transceiver. Be sure to note the limitations for the CAN transceivers when connecting the external supply voltage.

The pin assignment for the DB-9 CAN plug is shown in the table below:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Pinout of DB-9 plug with 82C251, TJA1054 (differential)</th>
<th>Pinout of DB-9 plug with AU5790 (single-wire) only available with USB-CANmodul2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N/C</td>
<td>N/C</td>
</tr>
<tr>
<td>2</td>
<td>CAN-L</td>
<td>N/C</td>
</tr>
<tr>
<td>3</td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>4</td>
<td>N/C</td>
<td>N/C</td>
</tr>
<tr>
<td>5</td>
<td>CAN shield</td>
<td>CAN shield</td>
</tr>
<tr>
<td>6</td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>7</td>
<td>CAN-H</td>
<td>CAN-H</td>
</tr>
<tr>
<td>8</td>
<td>N/C</td>
<td>N/C</td>
</tr>
<tr>
<td>9</td>
<td>N/C for USB-CANmodul1 N/C for USB-CANmodul8/16 V_{BAT} (+7 to +27 VDC)* for USB-CANmodul2</td>
<td>V_{BAT} (+5.3 to +16 VDC)*</td>
</tr>
</tbody>
</table>

**Note:**
The value for $V_{BAT}$ depends on the alternative CAN transceiver that populates the device.

The Common Mode Voltage between GND (Pin 3 or 6) and CAN-L (Pin 2) or CAN-H (Pin 7) is limited to max. +12 VDC for using the high-speed CAN transceiver 82C251.

2.3 Termination resistor for high-speed CAN Transceiver

Please note that there always has to be connected two termination resistors with value 120 Ohms, if you are using an USB-CANmodul with a high-speed CAN transceiver 82C251. These has to be connected to both ends of the CAN bus:

![Termination resistors on CAN bus](image-url)
Note:
When using a special version of the device featuring a low-speed CAN transceiver (e.g. TJA1054) no terminating resistor must be used because it is already integrated in the device.

On USB-CANmodul2 USB-CANmodul8 and USB-CANmodul16 a termination resistor with 120 Ohms is already built in for each high-speed CAN-channel. You can enable or disable it by closing a jumper (JP200 and JP300 for USB-CANmodul2 — refer to Figure 5) or by switching a switch on front panel (USB-CANmodul8, USB-CANmodul16). The default state of the termination resistor is: disabled.

If you decide to enable the termination resistor, change the appropriate switch to ON or close the appropriate jumper.

The current state of the termination resistor can be indirectly read back by software only on USB-CANmodul2 (by calling function UcanReadCanPort() and/or UcanReadCanPortEx() or by opening the Ports Dialog in Control Panel Application USB-CANmodul Control – refer to Figure 31). Please note that the jumper JP104 must have the same state as JP200 (for CAN-channel 0) and the jumper JP105 must have the same state as JP300 (for CAN-channel 1). Otherwise the read state of the termination resistor is not correct. The reason of this solution is the optical isolation of the CAN-channels.

Refer to Table 5 for recommended cable parameters of the CAN bus.

Table 5: Recommended cable parameters

<table>
<thead>
<tr>
<th>max. cable length [m]</th>
<th>max. bit rate [kbps]</th>
<th>specific resistance [kΩ/m]</th>
<th>Cable cross-section [mm²]</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>1000</td>
<td>70</td>
<td>0,25..0,34</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
<td>&lt;60</td>
<td>0,34..0,60</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>&lt;40</td>
<td>0,50..0,60</td>
</tr>
<tr>
<td>1000</td>
<td>20</td>
<td>&lt;26</td>
<td>0,75..0,80</td>
</tr>
</tbody>
</table>

Note:
In case of highly electromagnetic disturbed applications we advise to well ground each side of the shield. Refer to the following standard: ISO11898-2:2003
- Section 3.10 - Physical media
- Section 7.5.1 - Physical medium specification – General
- Table 9 - Physical media Parameters of a pair of wires (shielded or unshielded)
2.4 CAN-port with low-speed CAN Transceiver

This section is not the scope of USB-CANmodul1.

The high-speed CAN transceiver 82C251 is implemented in the standard configuration of the device. As an alternative, other CAN transceiver can be populated on the USB-CANmodul2. In this case only the behavior on the CAN bus changes, not the behavior in relation to the software. From the software point of view (e.g. using the included tool PCANView (USBCAN)) any transceiver can be used.

The optional low-speed transceiver TJA1054 or the single-wire transceiver AU5790 have multiple signals for setting the operating mode of the transceivers and displaying the operating state. The following signals are supported:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Name</th>
<th>Meaning</th>
<th>Type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>Enable</td>
<td>Enable control</td>
<td>high-active output</td>
<td>high level</td>
</tr>
<tr>
<td>/STB</td>
<td>Standby</td>
<td>Stand-by control</td>
<td>low-active output</td>
<td>high level</td>
</tr>
<tr>
<td>/ERR</td>
<td>Error</td>
<td>error, wake-up and power-on indication output</td>
<td>low-active input</td>
<td>high level</td>
</tr>
<tr>
<td>TRM</td>
<td>Termination</td>
<td>termination resistor</td>
<td>high-active input</td>
<td>low level</td>
</tr>
</tbody>
</table>

Note:

It is only possible to read the state of the termination resistor by software using USB-CANmodul2. It reads the state of jumpers JP104 and JP105 (refer to Figure 5). Make sure jumper JP104 has the same state as jumper JP200 and make sure jumper JP105 has the same state as jumper JP300 for the correct use of this feature.

The signals /STB and EN of low-speed channel of USB-CANmodul2 (order code 3204019) cannot be separately switched. They are both interconnected at the TJA1054.

The single-wire CAN transceiver AU5790 does not have an error output. Its operation modes are described in Table 7.

<table>
<thead>
<tr>
<th>/STB</th>
<th>EN</th>
<th>Description</th>
<th>CANH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Sleep mode</td>
<td>0 V</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Wake-up transmission mode</td>
<td>0 V, 12 V</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>High-speed transmission mode (83,3 kbps)</td>
<td>0 V, 4 V</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Normal transmission mode (33,3 kbps)</td>
<td>0 V, 4 V</td>
</tr>
</tbody>
</table>

Use the API functions UcanWriteCanPort() and/or UcanWriteCanPortEx() to change the output state of signals listed in Table 6 and Table 7. To read the input states of the signals listed in Table 6 use the API functions UcanReadCanPort() and/or UcanReadCanPortEx().
2.5 Expansion Port

Only the USB-CANmodul2 features an 8-bit port for functional expansion which can be used to add digital inputs (e.g. push buttons) and digital outputs (e.g. LEDs) to the device. An additional 2*5-pin header connector in 2.54 mm pitch (male, X400 – refer to Figure 5) is provided on the USB-CANmodul2 with order code 3204007. The connector X400 has the following pinout:

Table 8: Expansion Port Pin Assignment on USB-CANmodul2

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB0</td>
<td>1</td>
<td>2</td>
<td>PB1</td>
</tr>
<tr>
<td>PB2</td>
<td>3</td>
<td>4</td>
<td>PB3</td>
</tr>
<tr>
<td>PB4</td>
<td>5</td>
<td>6</td>
<td>PB5</td>
</tr>
<tr>
<td>PB6</td>
<td>7</td>
<td>8</td>
<td>PB7</td>
</tr>
<tr>
<td>GND</td>
<td>9</td>
<td>10</td>
<td>Vcc Output</td>
</tr>
</tbody>
</table>

The microcontroller's port pins are connected directly to the expansion port, there is no protective circuit. Make sure that external circuitry connected to this port does not exceed the maximum load tolerance of the corresponding port pins (refer to Table 9)! The port pins can be configured to be used as inputs or outputs.

Use the API functions UcanConfigUserPort() to configure the port direction of signals listed in Table 8. To read the input states of the expansion port use the API functions UcanReadUserPort() and/or UcanReadUserPortEx(). As well as use the API functions UcanWriteUserPort() to change the output state of signals which are configured as output signal.

Table 9: Properties of port expansion on USB-CANmodul2

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Condition</th>
<th>min.</th>
<th>typ.</th>
<th>max.</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vih</td>
<td>Input High Voltage</td>
<td></td>
<td>2.0</td>
<td>5.5</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>vil</td>
<td>Input Low Voltage</td>
<td></td>
<td>-0.3</td>
<td>0.8</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Vol</td>
<td>Output Low Voltage</td>
<td>Iout = 8 mA</td>
<td>2.9</td>
<td></td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>CIN</td>
<td>Input Pin Capacitance</td>
<td></td>
<td></td>
<td>5</td>
<td>pF</td>
<td></td>
</tr>
<tr>
<td>Iout</td>
<td>Output Current</td>
<td></td>
<td>8.0</td>
<td></td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>Vcc</td>
<td>Supply Voltage</td>
<td></td>
<td>3.2</td>
<td>3.4</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

A user circuit of the Expansion Port depends on the necessity to which level the hardware of USB-CANmodul has to be protected against destruction. You find an example of a user circuit without protection in the next figure.
Please note that if Vcc is used as power supply for your circuit, the total current of an USB device may not exceed 500 mA (during plug-in the total current actually may not exceed 100 mA). If bus powered USB hubs are used, there could be problems even below 500 mA. Some USB hubs share its power supply with the number of available USB ports. Please note that there could also be problems below 500 mA if other USB devices are connected to these ports. Thus, we advise to implement a galvanic decoupled circuit that has its own power supply.
2.6 LEDs on the USB-CANmodul

Each USB-CANmodul device has a yellow power LED. It illuminates as soon as the power is connected to the device (by connecting the device to a PC using an USB cable).

Additionally the USB-CANmodul has a traffic LED and a status LED for each available CAN channel. The green traffic LED is switched off as long as the CAN interface is not initialized. After initialization it blinks when a CAN message is transferred on the CAN bus (for sending or reception). Figure 15 and Figure 16 shows the principle of switching the traffic LED after the transmission of CAN messages on the CAN bus. Each CAN messages starts a 256 ms cycle blinking the traffic LED.

![Traffic LED after one CAN message on CAN bus](image)

**Figure 15:** Traffic LED after one CAN message on CAN bus

![Traffic LED after more CAN messages on CAN bus](image)

**Figure 16:** Traffic LED after more CAN messages on CAN bus

The state of each CAN-channel on the USB-CANmodul is displayed via a red LED. In order to distinguish the states, different blinking cycles were defined respectively. A description of the power LED and the state LED is shown in the Table 10. Figure 17 shows the different blinking cycles.

![Blink cycles of the state LED](image)

**Figure 17:** Blink cycles of the state LED
Table 10: States of the LEDs on the USB-CANmodul devices

<table>
<thead>
<tr>
<th>USB-CANmodul connected?</th>
<th>LED yellow (power)</th>
<th>LED red (state)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>off</td>
<td>off</td>
<td>No voltage is supplied to the device.</td>
</tr>
<tr>
<td>no&lt;sup&gt;1&lt;/sup&gt;</td>
<td>on</td>
<td>blinking cycle 1</td>
<td>USB cable not connected.</td>
</tr>
<tr>
<td>yes</td>
<td>on</td>
<td>blinking cycle 1</td>
<td>Device logs in to the host-PC.</td>
</tr>
<tr>
<td>yes</td>
<td>on</td>
<td>on</td>
<td>Log-in successful, CAN-channel is not initialized, no error.</td>
</tr>
<tr>
<td>yes</td>
<td>on</td>
<td>off</td>
<td>CAN-channel is initialized, no error.</td>
</tr>
<tr>
<td>yes</td>
<td>on</td>
<td>blinking cycle 2</td>
<td>A CAN-bus error occurred on this channel.</td>
</tr>
<tr>
<td>yes</td>
<td>on</td>
<td>blinking cycle 3</td>
<td>Firmware update running. The device must not be powered-off or disconnected while the firmware update is running.</td>
</tr>
</tbody>
</table>

<sup>1</sup> Only occurs on self-powered USB-CANmodul devices
3 Getting Started

Ensure that the individual components are not damaged. The delivery content of the USB-CANmodul includes:

- One USB-CANmodul device
- One Installation CD-ROM with electronic version of the latest systems manual and all software and drivers
- One USB cable

3.1 System requirements

The system requirements are:

- Processor: 1 GHz or faster (2 GHz or faster recommended)
- RAM: 1 GB or more (2 GiB or more recommended)
- Approx. 32 MB free disc space
- USB port according USB 1.1 spec. or higher
- Microsoft Windows 7 or higher.
  Since driver version V6.00 Windows 10 is supported (valid for modules since hardware revision -01 – refer to section 2.1).
- Windows XP is not supported any more.

Note:

For Windows 7 the Microsoft Hotfix KB3033929 is required! To check whether the Microsoft Hotfix KB3033929 is already installed use the following command at the console:

```
wmic qfe get hotfixid
```

(This will list all installed hotfixes)

Win driver version V6.05 the kernel drivers are tested for Windows 10 Professional version 1903 Build 18362 and signed by the Microsoft Hardware Developer Dashboard.
3.2 Installation of the driver under Windows-OS

Follow the steps below to install the device driver to your Windows PC:

1) Start your computer.

2) Insert the USB-CANmodul Utility CD-ROM in your CD-ROM drive. If you have downloaded the driver from our homepage then continue with step 4).

3) Open the Windows Explorer and locate the following path: 
   "<CD-ROM>:\Products\USB-CANmodul_xxxxx\Software\SO-387"
   "xxxxxx" specifies the order code listed in Table 1.

4) Execute file SO-387.exe, which will start the setup tool. NOTE: You will need administrator rights to execute this file! Enter the administrator password if Windows asks for it. The following window will appear:

5) Click Next. Read and accept the License Agreement in the next window and click Next again.
6) Additional version information are displayed in next window. Click Next again.

7) Edit all the user information in next window and click Next again.
8) In the next window you select the destination location of the USB-CANmodul software. We recommend to use the predefined destination location. Click Next to continue.

9) The following important note is displayed. Read it carefully and select “YES”, that you understood the notice. Click the Next button again.
10) In the next window you select the type of installation you wish to perform (*Full Installation is recommended*).

11) Follow all further setup instructions to install the USB-CANmodul software until the **Additional settings** page appears. The first checkbox activates the check for newer driver version via Internet at each start of the USB-CANmodul. We recommend to activate this checkbox. If the second checkbox is activated, the kernel driver will not automatically update the firmware versions on the USB-CANmodul device. Click Next to continue.
12) Continue the installation by clicking the **Install** button. After the setup routine has installed all needed files the hardware assistant is called automatically to register the kernel drivers. The following windows may appear. Please check the checkbox for always trusting the software from company SYSTEC electronic GmbH and click to the **Install** button.

![Dialog window for software installation](image1)

**Note:** On Windows 10 this dialog window is not shown.

13) The next page displays the revision information. Click the **Next** button to continue.

![Revision history dialog](image2)

14) Finish the installation by clicking the **Finish** button.
15) Connect the USB-CANmodul to your computer using the included USB cable. Windows automatically detects the USB-CANmodul. The appropriate driver files will be found automatically. The firmware will now be downloaded to the USB-CANmodul. The red status LED blinks with a frequency of 4 Hertz 4:1 duty cycle to indicate this firmware update procedure (refer to Figure 17 and Table 10).

16) After successful download of the device firmware the red status LED will stay on.

**Note:**
Do not unplug the USB cable from the PC and/or the USB-CANmodul unless the firmware update procedure is not finished (refer to step 16).

### 3.3 Updating an existing installation

For updating the USB-CANmodul driver SYS TEC electronic AG provides a feature to download the driver via Internet using the tool USB-CANmodul Control (refer to Figure 18).

![USB-CANmodul Control Check for Update](image)

*Figure 18: USB-CANmodul Control Check for Update*

Click to the button **Check for Updates** to check for a newer driver version. If there is no newer version a dialog box appears with the message “**A newer version is not available**”. Otherwise a dialog box appears with the message “**A newer version is available. Please read the revision history and click ‘Start Download’**”. The tab-sheet **Update** displays the revision history of the new driver:
Figure 19: USB-CANmodul Control Start Download

After the download has finished a dialog box appears with the message "Download successfully finished. Click ‘Start Setup’". The button text on the bottom left changes to “Start Setup”. Click on that button to start the installation.

The setup tool starts with the following message box:

Figure 20: Updating an existing installation

Click on the “Yes”-button to uninstall the old driver and to install the newer one.

You also can download the latest driver from our homepage http://www.systec-electronic.com. Extract the downloaded ZIP file and execute the file SO-387.exe. Follow all steps described in section 3.2.
3.4 Verifying the Device Installation

Verification of correct device installation on your host-PC can be done by following the steps listed below:

- Open the System Control from the start menu of Windows.
- Choose the “Device Manager” at the top. It may be necessary to re-adjust the “View-By” mode (top right corner of the window) to “Large Icons” or “Small Icons”.
- Click on the tree node “USB-CAN-Hardware”. If the device “Systec USB-CANmodul Device Driver” or “Systec USB-CANmodul Network Driver” is shown in the list, the new USB device has been detected properly. This is shown in the figure below:

![Device Manager with the USB-CANmodul](image)

*Figure 21: Device Manager with the USB-CANmodul*

If the installation was not successful, check the installation steps as described above and try to re-install.
3.5 Device Number Allocation

With the help of device number allocation, it is possible to use more than one USB-CANmodul simultaneously on the host-PC. The device number identifies the individual USB-CANmodul on the API functions of the DLL.

- Click on Start → Programs → USB-CANmodul Utilities → Tools → USB-CANmodul Control. The following window will appear:

![USB-CANmodul Control tab-sheet Hardware](image)

*Figure 22: USB-CANmodul Control tab-sheet Hardware*

- Select/highlight one of the modules shown in the hardware list and then click on the Change... button.
- Enter a new device number in the input field or modify the device number using the Up or Down button. Click OK to exit this window.
- The new device number will only take affect and gets downloaded into the device after clicking the Apply or OK button.

![Device number changing dialog box](image)

*Figure 23: Device number changing dialog box*
Note:
The device number of USB-CANmodul devices which are grayed out in the list cannot be changed because they are exclusively used by other applications.

3.6 Connection to a CAN Network

The USB-CANmodul provides a DB-9 plug for connection to the CAN network. The pin assignment on this connector is in accordance to the CiA (CAN in Automation) specification. Connect your CAN network to this connector with an appropriate CAN bus cable. The pinout is described in Table 4 on page 21.

Note:
When using the standard version of the USB-CANmodul with on-board high-speed CAN transceivers (e.g. 82C251) a termination resistor of 120 Ohms at both ends of the CAN cable between CAN-L (pin 2 of the DB-9 plug) and CAN-H (pin 7 of the DB-9 plug) is required to ensure proper signal transmission. When using a special version of the device featuring a low-speed CAN transceiver (e.g. TJA1054) no terminating resistor must be used because it is already integrated in the device. It is necessary to use shielded cables if the CAN bus extension exceeds 3 meters. Refer to section 2.3.

3.7 Starting PCANView (USBCAN)

The included program PCANView (USBCAN) is a CAN bus monitor for Windows.

Note:
The tool PCANView (USBCAN) is free of charge which was distributed by the company Peak-System Technik GmbH. Related to the drivers of the USB-CANmodul there would never be an update to a newer version of the tool.

- Start the utility program using the Windows Start button and browse to Programs → USB-CANmodul Utilities → Tools → PCANView (USBCAN).
- The USB-CANmodul settings window will appear:

![PCANView (USBCAN) settings](image)

*Figure 24: Dialog box with hardware configuration in PCANView*
- Select the baud rate of your CAN network in the **Baudrate** box and the Device Number. The entry "**any**" selects the USB-CANmodul that is found first by Windows.

- If "**user**" is selected in the baud rate field, then the values for register BTR Ext can be entered directly. Refer to section 4.3.4 for detailed information.

- When using an USB-CANmodul2 please select the CAN-channel you want to use.

- When using the USB-CANmodul8 or USB-CANmodul16 then always two CAN channels are combined to a "logical device". Refer to section 4.3.8 for detailed information how to find the correct CAN channel via software.

- A high resolution time stamp for CAN-reports can be set via "**high res. timer**". If this feature is activated the column "**Period**" within the main window of PCANView has a resolution of 100µs (instead 1ms) for the standard variant.

- Click on the **OK** button to enable these settings.

- A new window **PCANView – Connect to net** will appear. If applicable enter the message filter for the CAN-controller and confirm with **OK**.

![Figure 25: Dialog box for message filter configuration in PCANView](image-url)
- The PCANView (USBCAN) main window will appear:

![PCANView (USBCAN) main window](image)

*Figure 26: PCANView (USBCAN) main window*

This screen is divided into two sections: Receive and Transmit

- **Receive**: monitors CAN messages that are received from a remote node
- **Transmit**: monitors CAN messages sent from the host-PC to the CAN network via the USB-CANmodul

- Click on the menu command **Transmit → New**. The dialog box “**New transmit message**” will appear:

![New transmit message dialog box](image)

*Figure 27: Dialog box for entering a new transmit message*

- Specify a Standard (11-bit) or Extended (29-bit) CAN identifier in field **ID**, the data length code (DLC) in field **Length** and the **Data** of the CAN message. The ID and Data has to be specified in hexadecimal format.

- The checkbox **Extended Frame** is checked automatically if the ID is greater than 0x7FF. But it can be checked manually by the user for IDs lower then 0x800 too.

- If the checkbox **Remote Request** is checked, then a CAN Remote Frame is sent instead of a Data Frame.

- Enter a 0 in the **Period** field, the CAN message will only be sent, if it has been selected in the Transmit section of the main window and the SPACE key was hit. If the period is set to a value greater then 0, then the message will be sent periodically.
Note:
The cycle period is controlled by the tool PCANView (USBCAN) which is contributed by the company Peak Systems Technik GmbH. This tool cannot be changed by the company SYS TEC electronic AG. Additionally, the Windows operation system is not a real-time OS. Therefore the real transmission period on the CAN bus is not exactly as specified in the Period field.

- Confirm all the settings by clicking to the OK button. Repeat the creation of new CAN messages if needed.

Note:
The tool PCANView (USBCAN) cannot save the list of the Transmit section.

- Transmit a selected CAN message manually by pressing the SPACE key. Received CAN messages are displayed in the Receive section of the main window. Both sections are sorted by the CAN-ID (no chronological view). All values of CAN-ID (column Message) and Data are in hexadecimal format.

![Figure 28: Transmission and reception of CAN messages in PCANView](image)

Refer to section 4.2.2 for further information about the tool PCANView (USBCAN).

Alternatively you can use the Emtas CANInterpreter with the USB-CANmodul which has additional features:

- Online monitoring of bus traffic in different views (e.g. chronological view)
- Interpretation of CAN-Data according to user specifications
- Tracing of CAN-Telegrams to a file on the hard disc and interpretation afterwards
- Flexible CAN-ID specific filtering
- Transmission of CAN messages or sequences in single or periodic

You can download an evaluation version of the tool at the SYS TEC homepage [http://www.systec-electronic.com](http://www.systec-electronic.com).
3.8 Creating a debug file from DLL

If problems with the software drivers should occur, there is a possibility to create a debug log file from USBCAN32.DLL and/or USBCAN64.DLL. You should always send this log file to our support email address so that we can find a solution for your problem.

To activate the feature please open **USB-CANmodul Control** from the control panel. At the tab-sheet **Debug** you will find the following window:

![Debug settings in USB-CANmodul Control](image)

Enable the feature by ticking the box "Enable Debug". In the list above you can activate different debug information that should be added to the debug log file. Click to "Browse" for choosing the folder in which the debug log file should be stored. The default setting is the "Documents" folder.

Apply the new settings and close USB-CANmodul Control. Start your application using an USB-CANmodul and wait until the problem will occur. After this close your application.

Afterwards, you will find a file named USBCAN_XXXXXXXX_YYYYYY.ZZZ.LOG. XXXXXXXX represents the creation date of the log file in format YYYYMMDD (year month day) and YYYYYY stands for the creation time in format HHMMSS (hour minute second). ZZZ is the name of the application executed.

**Note:**

Enabling this feature decreases the performance of the software, because API functions have to execute much more code to generate debug outputs. Limiting the debug information by changing the LOG-Level can help to increase performance again. But note that in this case important information could be missing in the log file.

Furthermore, the debug log file may increase in size. Activate the feature "Check max. LOG file size". This way, USBCAN32.DLL and/or USBCAN64.DLL will monitor the file size of the debug log file. If it is exceeded, a new debug log file will be started. Default setting of the maximum debug file size is 10240 Kbytes (means 10 Mbytes).

An application can call the function **UcanSetDebugMode()** for subsequent activation of the feature.

With the Check-Box "Show Dbg Info" you can set, whether a dialog box should be opened upon opening an application, reminding of activated debug feature. This is to avoid that debug feature remain continuously activated without being noted filling the main board with log-files.
At the bottom of the list of log functionality you find check boxes for logging kernel outputs too. If at least one of the kernel checkboxes is activated a kernel trace file USBCAN_Trace_XXXXXXX_OSYYYYYYYY_ZZZZZZZ.etl is written to the output folder. XXXXXXXX represents the hexadecimal version number of the kernel driver file and YYYYYYY stands for the detailed version of the Windows OS. ZZZ is an additional information of the Windows OS (e.g. Service Pack information).

Example:

```
USBCAN_Trace_00000A05_OS2#64#V6.1.7601_Service Pack 1.etl
```

Note:
If using the debug feature for kernel outputs always follow the following sequence:
1. Activate the Debug Feature in USB-CANmodul Control
2. Close the USB-CANmodul Control
3. Start your application until the problem occurs
4. Deactivate the Debug Feature in USB-CANmodul Control
5. Close the USB-CANmodul Control
6. Locate the ETL file using the File Explorer and send it to the SYS TEC support

The Windows OS only can complete all write operations to the ETL file after the Debug Feature is deactivated in USB-CANmodul Control. If the tool is not closed then the ETL file does not contain all the debug information. In this case it would be not helpful for any support assistance.

### 3.9 Activation of the network driver

The network driver UCANNET.SYS was developed for connecting several applications to one physical USB-CANmodul. Therefore, the kernel mode driver creates a virtual CAN network for each physical module to which several applications can connect to. All CAN messages that are sent by an application are not only sent to the physical CAN bus but also to all the other connected applications. Received CAN messages are passed on to all applications.

To activate the network driver for an USB-CANmodul, open the USB-CANmodul Control from the Control Panel. Mark the module within the hardware list that you want to use for the network driver. Push the button “Change...” to open the dialog box shown in Figure 23. Check the box “use USB-CANnetwork driver” and confirm with “OK”. After pushing the button “Apply” or “OK” in the main window of the USB-CANmodul Control, the USB-CANmodul automatically reconnects to the host PC. This results in exchanging the kernel mode driver. Now you can use several applications with this USB-CANmodul.

Each USB-CANmodul which is configured to use the network driver is marked with “net” in column “drv” in tab-sheet Hardware of the USB-CANmodul Control (refer to Figure 22).

### 3.10 Completely uninstall the driver

To completely uninstall the driver, tools and examples we recommend to use the command

Windows Start Menu → Programs → USB-CANmodul Utilities → Uninstall USB-CANmodul Utilities. Follow all steps to uninstall the driver.

Note:
The Microsoft Windows operation system keeps some system files and registry keys after the uninstallation process. The following files may be deleted manually from the folder "%windir%\ system32\drivers" or "%windir%\syswow64": ucannet.sys, usbcan.sys, usbcani3.sys, usbcani4.sys, usbcani5.sys, usbcani21.sys and usbcani22.sys. But the appropriate registry keys under “HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB!” cannot be deleted.
Under Windows 10 there is no link to the uninstallation of the driver. In this case use Programs and Features of the control panel.
4 Software Support for Windows OS

4.1 File Structure

If during the installation of the **USB-CANmodul Utilities** no other destination directory is given, then all files will be installed in the folder:

```
C:\Program Files\SYSTEC-electronic\USB-CANmodul Utility Disk
```

Using the 64-bit version of Windows OS the default destination directory is:

```
C:\Program Files (x86)\SYSTEC-electronic\USB-CANmodul Utility Disk
```

The contents of this folder are given in **Table 11**. Some folders are created depending on selected installation options during setup process.

**Table 11:**  Software file structure

<table>
<thead>
<tr>
<th>Sub-folder</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>Bin*</em></td>
<td>Executable files (e.g. PCANView)</td>
</tr>
<tr>
<td>*<em>Contrib*</em></td>
<td>Files contributed by other companies</td>
</tr>
<tr>
<td>*<em>LabView*</em></td>
<td>LabView driver with demo</td>
</tr>
<tr>
<td>**UcanTraceView**</td>
<td>Manuals</td>
</tr>
<tr>
<td>*<em>drv*</em></td>
<td>Windows Kernel drivers (current version)</td>
</tr>
<tr>
<td>*<em>drv.win7*</em></td>
<td>USBCAN32.DLL and USBCAN64.DLL version V4.18 which fixes an issue initializing an USB-CANmodul with firmware version since V5.00.</td>
</tr>
<tr>
<td>*<em>drv.win10*</em></td>
<td>USBCAN32.DLL and USBCAN64.DLL version V4.18 which fixes an issue initializing an USB-CANmodul with firmware version since V5.00.</td>
</tr>
<tr>
<td>*<em>Examples*</em></td>
<td>Demo projects for MS Visual Studio 2008 and 2010</td>
</tr>
<tr>
<td>*<em>ConvertUcanTrace*</em></td>
<td>A .NET application in C# using the Wrapper-DLL UcanDotNET.dll for conversion the binary CAN trace file created by the Tool PCANView (USBCAN) – refer to section 4.2.2.</td>
</tr>
<tr>
<td>*<em>Demo.api*</em></td>
<td>A simple MFC demo in source for one channel USB-CANmodul</td>
</tr>
<tr>
<td>**DemoCyclicMsg**</td>
<td>A simple MFC demo in source for automatically transmission of cyclic CAN messages using</td>
</tr>
<tr>
<td>*<em>DemoEnum*</em></td>
<td>A simple MFC demo in source using the API function for enumerating the connected USB-CANmodul</td>
</tr>
<tr>
<td>*<em>DemoGW006*</em></td>
<td>A simple MFC demo in source for a dual-channel USB-CANmodul</td>
</tr>
<tr>
<td>*<em>Include*</em></td>
<td>Header files in C language for the USBCAN32.DLL / USBCAN64.DLL. All demo applications for MS Visual Studio refers to these files.</td>
</tr>
<tr>
<td>*<em>Lib*</em></td>
<td>Common USBCAN32.DLL / USBCAN64.DLL and import- libraries for MS Visual Studio. The demo applications refer to these import-libraries.</td>
</tr>
<tr>
<td>*<em>UcanDotNET*</em></td>
<td>Wrapper-DLL in source code for use with Microsoft .NET projects.</td>
</tr>
<tr>
<td>**USBcanDemoVB**</td>
<td>MS Visual Basic .NET demo application in source code (using the Wrapper-DLL UcanDotNET.dll)</td>
</tr>
<tr>
<td>*<em>firmware*</em></td>
<td>Includes the firmware written to the USB-CANmodul hardware.</td>
</tr>
</tbody>
</table>

Footnotes:

1. Only available in driver version below V6.00.
2. Only available in driver version since V6.00.
3. Only available in driver version since V6.00 depending on the used Windows version.
4.2 Tools for the USB-CANmodul

4.2.1 USB-CANmodul Control for Windows

The USB-CANmodul Control tool can be started either from the Windows Control Panel or from the program group "USB-CANmodul Utilities". Figure 22 shows the tool after start up.

This tool may be used to modify the device number of the USB-CANmodul devices (also refer to section 3.5).

For the modules listed in Table 1 the tool USB-CANmodul Control can increase the performance of an USB-CANmodul by clicking to the button "Change..." since driver version V5.11. If activated the CPU frequency is increased from 96 MHz to 120 MHz (increased by 25%). This has an effect on the bit rates on the CAN bus (refer to section 4.3.4.3). Note that the checkbox is not available for older generations of the USB-CANmodul devices.

![Change parameters](image)

**Figure 30:** Activation of higher performance

In addition, this tool can also be used to manipulate the 8-bit port expansion (refer to section 2.5) and the CAN port for low-speed CAN transceivers (refer to section 2.4). To do this you have to select the corresponding USB-CANmodul from the list and then click on the "Ports..." button. Figure 31 shows the dialog box that will appear when choosing this option.

![Ports](image)

**Figure 31:** Dialog box for manipulating the port expansion and the CAN port

Initially all 8 signals are configured as inputs. With the line OE, the corresponding signal is switched to an output. This activates the box for the output value in the line OUT. If a signal is switched to a logical 1 in this line, then the corresponding signal on the port expansion will be set to high. With every modification the current state of the expansion port will be read again and shown in the line IN for the inputs. To read the current input states without having to change an output, click on the "Update Input" button.
The current state of the CAN port for the low-speed CAN transceiver is displayed in the bottom of the window. The signals EN and /STB are outputs and the signals /ERR and /TRM are an inputs. For more information refer to section 2.4.

4.2.2 PCANView (USBCAN) for Windows

The Windows utility PCANView (USBCAN) can be used to display CAN messages transmitted via the CAN bus.

Note:
The tool PCANView (USBCAN) is free of charge which was distributed by the company Peak-System Technik GmbH. Related to the drivers of the USB-CANmodul there would never be an update to a newer version of the tool.

After execution of the tool a dialog box is shown for configuring the hardware parameters (refer to Figure 24). The device number of the logical USB-CANmodul has to be filled in to the edit field “Device-Nr.”. This device number was previously programmed with the Windows Control Panel symbol USB-CANmodul Control (refer to section 3.5). Within the drop down box “Baudrate” the baud rate on CAN bus can be selected. The option “listen only” configures the CAN controller for only receiving CAN messages. This also means that no acknowledge will be sent back to the sending remote CAN device. For a logical USB-CANmodul including two CAN-channels the channel has to be selected which should be used by the tool.

After applying the settings by clicking to the “OK” button a dialog box is shown like displayed in Figure 25. The filter setting depends on the CAN message format you wish to receive: CAN identifier with 11 bits (standard frame = CAN Spec. 2.0A) or CAN identifier with 29 bits (extended frame = CAN Spec. 2.0B). Please choose one of both possibilities and enter the range of the CAN messages which has to be shown on receive section of the tool. If you do not change this range, then all CAN messages will be shown. Apply this setting by clicking to the button “OK”.

The main window of the tool appears (refer to Figure 7). This screen is divided into two sections: Receive and Transmit. Both sections are sorted by the CAN identifiers. That means no chronology is displayed.

- Receive: monitors CAN messages that are received from a remote node
- Transmit: monitors CAN messages sent from the host-PC to the CAN network via the USB-CANmodul

Receive Section

The Receive section provides the following information:

- Message: identifier of the CAN message, hexadecimal format, ranging from 0 to 7FFh for 11-bit identifiers and from 0 to 1FFFFFFFh for 29-bit identifiers
- Length: data length code of the message (ranges from 0 to 8)
- Data: values of the messages’ data bytes (up to 8) or the text Remote request, if a remote frame has been received
- Period: period of time between the reception of the last two messages with this identifier
- Count: number of messages received with this identifier (no remote frames) since last user reset
- RTR-Per.: period of time between the reception for the last two remote frames
- RTR-Cnt.: number of remote frames with this identifier
Transmit Section

The Transmit section provides the following information:

- **Message:** identifier of the CAN message, hexadecimal format, ranging from 0 to 7FFh for 11-bit identifiers and from 0 to 1FFFFFFFh from 29-bit identifiers
- **Length:** data length code of the message (ranges from 0 to 8)
- **Data:** values of the messages’ data bytes (up to 8) or the text Remote request, if a remote frame shall be sent
- **Period:** period of time between the last two message sent
- **Count:** number of messages sent with this identifier (no remote frames) since last user reset
- **Trigger:** reason for the last transmission of the message:
  - **Manual:** manual transmission by the user pressing the SPACE bar
  - **Time:** period of time has passed for periodical sending
  - **RTR:** remote frame with the same CAN-ID has been received

**Note:**

If, at creation of the message, the period was set to 0, the text *Wait* is shown in column *Period*. In this case the message can only be transmitted manually using the SPACE bar. Or the message is sent automatically after a matching remote frame has been received.

The cycle period is controlled by the tool PCANView (USBCAN) which is contributed by the company Peak Systems Technik GmbH. This tool cannot be changed by the company SYS TEC electronic AGmbH. Additionally, the Windows operation system is not a real-time OS. Therefore the real transmission period on the CAN bus is not exactly as specified in the *Period* field.

In order to edit the Transmit list, the following menu commands are available:

- **Transmit → New...:** Create a new transmit message. The editor window for the new message is displayed.
- **Transmit → Edit...:** Edit the currently selected message.
- **Transmit → Delete:** Delete the currently selected message from the transmit list.
- **Transmit → Clear all:** Delete the entire transmit list.
- **Client → Reset:** Reset the message counters and reset the connected USB-CANmodul. Also completely deletes the receive list.

**Note:**

The tool PCANView (USBCAN) cannot save the list of the Transmit section.

PCANView (USBCAN) implements additional commands within the system menu (refer to *Figure 32*). These commands are SYS TEC extensions:

- **SYSTEC cyclic CAN messages:** defines a list of up to 16 CAN messages that will be sent cyclically by the USB-CANmodul firmware
- **SYTEC Write CAN-Trace-File:** writes a binary CAN trace file for logging the CAN messages on the CAN bus
**SYSTEC cyclic CAN messages**

This feature can be used for instance when precisely timed CAN messages have to be sent to the CAN bus (e.g. SYNC messages). By clicking on this menu command a dialog box opens where all cyclic CAN messages can be configured (refer to Figure 33). Use button **Add** to add a new cyclic CAN message. With button **Edit** a previously marked cyclic CAN message can be edited. Each USB-CANmodul supports up to 16 cyclic CAN messages. Choose option **parallel** if the cycle time of each CAN message should refer to itself (refer to Figure 35). With option **sequential** the cycle time of each CAN message refers to its previous CAN message (refer to Figure 36).

**SYTEC Write CAN-Trace-File**

The tool PCANView (USBCAN) can write a CAN Trace File by clicking to this command. A “**Save as**” dialog opens for specifying the file name and the destination directory. If the feature is activated, then a checked-mark is displayed before the command within the system menu. All CAN messages are stored to the CAN Trace File, which are received and transmitted within the activated time.

The CAN Trace File is stored as binary file with the default file extension “*.ucantrc”. The company SYS TEC electronic AG provides a .NET tool for conversion of the binary CAN Trace File into a text file. Refer to section **4.2.3** for detailed information.
It is possible to run PCANView (USBCAN) by using command line parameters (e.g. using a batch file). If at least one of these command line parameters is used then the dialog box for hardware parameters is not shown (refer to Figure 24).

The following command line parameters are available in PCANView:

**Syntax:**

```
PCANView.exe [-d<devicenr>] [-c<channelnr>] -b<baudrate> [-l]
[-x<x-pos> -y<y-pos>] [-n] [-t<timeout>] [-r]
```

- **-d** `<devicenr>` defines the device number of the USB-CANmodul. The range of values is between 0 and 254. Any USB-CANmodul that is found first is allocated value 255 (default value).
- **-c** `<channelnr>` defines the CAN channel that is to be used for dual-channel USB-CANmoduls. The range of values is between 0 and 1. Default value is 0 (first CAN channel).
- **-b** `<baudrate>` defines the bit rate on CAN bus in kBit/sec. Possible values are 1000, 800, 500, 250, 125, 100, 50, 20, 10. There is no default value for this parameter. It is mandatory.
- **-l** If this parameter is set, the USB-CANmodul will be initialized in listen-only mode. In this case CAN messages cannot be sent. No acknowledge bit is set on CAN bus for received CAN messages.
- **-x, -y** `<x-pos>` and `<y-pos>` define the position of the PCANView main window. If those parameters are missing, the position of the PCANView main window is read from the registry and is the same for all instances.
- **-n** If this parameter is set, the dialog box to configure the message filter will be skipped too (refer to Figure 25). In this case all CAN messages will always be received.
- **-t** With `<timeout>` the send timeout is set, from which a multi-channel USB-CANmodul dismisses all subsequent CAN messages in order to avoid a block of another CAN-channel (also refer to function `UcanSetTxTimeout()`). The standard value is 0 (no Timeout).
- **-r** This parameter indicates, that the high resolution time stamp is to be used. The column „Period“ in main window then has a resolution of 100µs (instead 1ms).

**Example:**

```
PCANView.exe -d1 -b1000 -n
```

The following command allows for starting more than one instance of PCANView from a batch file. The batch file starts two instances of the tool without waiting until the first PCANView instance is closed:

```
start PCANView.exe -d1 -b1000 -n -x20 -y35
start PCANView.exe -d2 -b1000 -n -x600 -y460
```
4.2.3 ConvertUcanTrace for Windows

This conversion tool is provided as source code when the “USB-CANmodul Utility Disk” is completely installed. So the customer is able to adapt other formats of the output file. There are no copyrights or license rights for this tool. The source code and the project files for Microsoft Visual Studio 2008 can be found in following subfolder related to the installation path of the USB-CANmodul Utility Disk at the hard disk:

\Examples\ConvertUcanTrace

Within the subfolder „bin\Release\x86“ the tool is provided as binary executable. A CAN Trace File created by the tool PCANView (USBCAN) can be converted into a txt file by calling the following command line:

ConvertUcanTrace.exe <input-file> [<output-file>] [options]

Example:

ConvertUcanTrace.exe pcanview_20150206_122846.ucantrc

To display a detailed help of the converter tool call the following command line:

ConvertUcanTrace.exe --help

Please note that the file UcanDotNet.dll for the x86 platform has to be stored into the current working directory of the converter tool too. The tool is compiled for the Microsoft .NET Framework 3.5. But it may be adapted for each other .NET Framework version by the customer.
4.3 Description of the USBCAN32.DLL / USBCAN64.DLL

The USBCAN32.DLL / USBCAN64.DLL is a function library for application programs. It serves as an interface between the system driver layer and an application program. The library for the USB-CANmodul enables easy access to the USB-CAN system driver functions. It administers the opened USB-CANmodul and translates the USB data into CAN messages.

Add the file USBCAN32.LIB or USBCAN64.LIB to your project for static linking the DLL file to your own Microsoft Visual C/C++ project. Starting the application program automatically loads the DLL. If the LIB is not linked to the project, or you are using another environment (e.g. Borland C++ Builder), load the DLL manually with the Windows API function LoadLibrary() and add the library functions with the function GetProcAddress() (refer to the demo application "DemoGW006").

The PUBLIC calling convention of the DLL functions provides a standardized interface to the user. This standard interface ensures that users of other programming languages than C/C++ (Pascal, etc.) are able to use these functions.

The subfolders Examples\Demo.api, Examples\DemoGW006 and Examples\DemoCyclicMsg contains example programs (Demos) created by using MFC for Microsoft Visual C/C++. These example projects demonstrates the use of the DLL API functions.

4.3.1 The concept of the DLL

With the DLL, it is possible to use up to 64 USB-CANmodul devices simultaneously with one application program, as well as with several application programs. However, it is possible to use one USB-CANmodul with several application programs if it is configured to use the network driver (refer to section 3.9).

Three states within the software are generated for each USB-CANmodul when using this DLL (refer to Figure 34).

After starting the application program and loading the DLL, the software is now in the DLL_INIT state. Concurrently, all required resources for the DLL have been created.

Calling the DLL function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware() causes the software to change into the HW_INIT state. This state contains all resources required for communication with the USB-CANmodul. It is not possible to transmit or to receive CAN messages in this state.

If the application software calls the library function UcanInitCan(), UcanInitCanEx() or UcanInitCanEx2() the state changes into CAN_INIT. In this state it is possible to transmit or to receive CAN messages.

Return with the library function UcanDeinitCan() or UcanDeinitCanEx() into the state HW_INIT and with the library function UcanDeinitHardware() into the state DLL_INIT. It is possible to close the application program only after this sequence is completed.

![Software State Diagram](image)

**Figure 34:** Software State Diagram

**Note:**
Make sure to return to the state DLL_INIT before closing the application program.
The number of available functions differs in the different software states. For example, the function `UcanWriteCanMsg()` causes an error if the software state is DLL_INIT for the appropriate USB-CANmodul. Table 12 shows the availability of each function within each state.

If multiple USB-CANmodul devices are used in one application, these states have to be considered for each USB-CANmodul that is used. If the first USB-CANmodul is in the state CAN_INIT, the second one could still be in the DLL_INIT state.
### Table 12: Available API functions according the software state

<table>
<thead>
<tr>
<th>State</th>
<th>API functions</th>
<th>single-channel</th>
<th>multi-channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLL_INIT</td>
<td>UcanSetDebugMode()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetVersion()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetVersionEx()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanInitHwConnectControl()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanInitHwConnectControlEx()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanEnumerateHardware()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanInitHardware()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanInitHardwareEx()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanInitHardwareEx2()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanDeinitHwConnectControl()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HW_INIT</td>
<td>UcanGetFwVersion()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetHardwareInfo()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetHardwareInfoEx2()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanSetDeviceNr()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetModuleTime()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetStatus()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanGetStatusEx()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanResetCan()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanResetCanEx()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanInitCan()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanInitCanEx()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanInitCanEx2()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanWriteCanPort()</td>
<td>-</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanWriteCanPortEx()</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanReadCanPort()</td>
<td>-</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanReadCanPortEx()</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanConfgUserPort()</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanWriteUserPort()</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanReadUserPort()</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanReadUserPortEx()</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanDefineCyclicCanMsg()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanReadCyclicCanMsg()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanDeinitHardware()</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CAN_INIT</td>
<td>UcanSetTxTimeout()</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanSetBaudrate()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanSetBaudrateEx()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanSetAcceptance()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanSetAcceptanceEx()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanReadCanMsg()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanReadCanMsgEx()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanWriteCanMsg()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanWriteCanMsgEx()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetMsgCountInfo()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanGetMsgCountInfoEx()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanEnableCyclicCanMsg()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetMsgPending()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanGetCanErrorCounter()</td>
<td>P0</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>UcanDeinitCan()</td>
<td>X</td>
<td>C0</td>
</tr>
<tr>
<td></td>
<td>UcanDeinitCanEx()</td>
<td>P0</td>
<td>X</td>
</tr>
</tbody>
</table>

**Meaning of entries in Table 12:**

"-" Function not supported

"X" Function supported without limitations

"C0" Function supported for each module with one CAN-channel (USB-CANmodul1) and/or only for CAN-channel 0 of a logical module with two CAN-channels (e.g. USB-CANmodul2), because the function parameter for selecting the channel number is missing.

"P0" Function only supported with function parameter selecting CAN-channel 0 of a logical module, because the hardware only has one CAN-channel (USB-CANmodul1). The function parameter specifying the CAN channel always must be set to USBCAN_CHANNEL_CH0.
4.3.2 API Functions of the DLL

This section describes the various API functions provided by the USBCAN32.DLL/USBCAN64.DLL. Most of the functions return a value of the type UCANRET containing an error code (refer to section 4.3.3). The meaning of this code is the same for each function. Besides the syntax, the meaning and the parameters of each function, the possible error codes are shown.

Some of the extended functions have an additional parameter for support of multi CAN channels and enable operations on a dual-channel module (USB-CANmodul2). These extended functions are also applicable on a single CAN-channel module (USB-CANmodul1), as long as CAN channel 0 is used for the parameter specifying the channel. Otherwise the functions returns with error code USBCAN_ERR_ILLCHANNEL (refer to section 4.3.3). All standard (single-channel) functions are applicable for a dual-channel module (USB-CANmodul2) as well, but do not provide the possibility to access other CAN-channels than the first channel.

4.3.2.1 General API functions

**Function:** UcanSetDebugMode

**Syntax:**

```c
BOOL PUBLIC UcanSetDebugMode (DWORD dwDbgLevel_p,
   TCHAR* pszFilePathName_p,
   DWORD dwFlags_p);
```

**Usability:** DLL_INIT, HW_INIT, CAN_INIT

**Description:** This function enables the creation of a debug log file out of the DLL. If this feature has already been activated via the USB-CANmodul Control, the content of the “old” log file will be copied to the new file. Further debug information will be appended to the new file.

**Parameter:**

- **dwDbgLevel_p:** Bit mask which enables the activation of debug information to be written into the debug log file. This Bit mask has the same meaning as the “LOG-Level” of the USB-CANmodul Control. Refer to section 3.8 and Table 13.
- **pszFilePathName_p:** Path leading to a text-based file which is written by the DLL with debug information. This parameter may be set to NULL. In this case only the new value of parameter dwDbgLevel_p will be set.
- **dwFlags_p:** Additional flag parameter. Value 0 will create a new debug log file. If the file referring to parameter pszFilePathName_p does already exist, the old content will be deleted upon opening. Value 1 though will append all new debug information to an existing file.

**Return:** If FALSE returns, the debug log file could not be created. A possible reason could be that the directory path which is set by the parameter pszFilePathName_p does not exist.

**Example:**

```c
// set debug mode for USBCAN API
UcanSetDebugMode (0xE0C00B03L, // = default Debug-Level
   _T("C:\\MyAppPath\\MyApp.log"),
   0); // = no append mode
```
### Table 13: Constants for the debug level passed to function UcanSetDebugMode()

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCAN_DEBUG_LVL_FCT</td>
<td>0x00000001</td>
<td>Writes all called API function names into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_PARAM</td>
<td>0x00000002</td>
<td>Writes all function parameters of the called API functions into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_RESULT</td>
<td>0x00000004</td>
<td>Writes all function results of the called API functions into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_READMSG</td>
<td>0x00000100</td>
<td>Writes the call of API functions <code>UcanReadMsg()</code> and/or <code>UcanReadMsgEx()</code> into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_WRITEMSG</td>
<td>0x00000200</td>
<td>Writes the call of API functions <code>UcanWriteMsg()</code> and/or <code>UcanWriteMsgEx()</code> into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_STATUS</td>
<td>0x00000400</td>
<td>Writes the call of API functions <code>UcanGetStatus()</code> and/or <code>UcanGetStatusEx()</code> into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_CMD</td>
<td>0x00000800</td>
<td>Writes internal commands into the log file which are sent to the modules firmware.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_SYS</td>
<td>0x00001000</td>
<td>Writes all system calls into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_SERVERTHREAD</td>
<td>0x00200000</td>
<td>Using the network driver it writes information into the log file which are sent to or received from other processes.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_DEVNOTTHREAD</td>
<td>0x00400000</td>
<td>Writes debug information about the device notification (plug &amp; play) into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_DATATHREAD</td>
<td>0x00800000</td>
<td>Writes debug information of data thread into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_CPL</td>
<td>0x04000000</td>
<td>Writes debug information of USB-CANmodul Control into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_BUFFINFO</td>
<td>0x08000000</td>
<td>Writes information of receive and/or transmit buffer of CAN messages into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_TIMESTAMP</td>
<td>0x10000000</td>
<td>Adds the Windows timestamp to each received and/or sent CAN messages into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_DUMP</td>
<td>0x20000000</td>
<td>Dumps the raw data of CAN messages into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_ERROR</td>
<td>0x40000000</td>
<td>Writes all error messages into the log file.</td>
</tr>
<tr>
<td>UCAN_DEBUG_LVL_ALWAYS</td>
<td>0x80000000</td>
<td>Writes all general information into the log file.</td>
</tr>
</tbody>
</table>
Function: UcanGetVersion

Syntax: DWORD PUBLIC UcanGetVersion (void);

Usability: DLL_INIT, HW_INIT, CAN_INIT

Description: This function returns the software version number of the USBCAN-library. It is overdue and should not be used in current projects. Use the function UcanGetVersionEx() instead of.

Parameter: none

Return: Software version number as DWORD with the following format:
- Bit 0 to 7: least significant digits of the version number in binary format
- Bit 8 to 15: most significant digits of the version number in binary format
- Bit 16 to 30: reserved
- Bit 31: 1 = customer specific version

Function: UcanGetVersionEx

Syntax: DWORD PUBLIC UcanGetVersionEx (tUcanVersionType VerType_p);

Usability: DLL_INIT, HW_INIT, CAN_INIT

Description: This function returns the version numbers of the individual software modules.

Parameter:
- VerType_p: Type of version information shows from which software module the version is to be returned. Table 14 lists all possible values for this parameter. The format of the version information differs from that of the UcanGetVersion() function.

Return: Software version number as DWORD using the following format:
- Bit 0-7: Version (use macro USBCAN_MAJOR_VER)
- Bit 8-15: Revision (use macro USBCAN_MINOR_VER)
- Bit 16-31: Release (use macro USBCAN_RELEASE_VER)

Example:

DWORD dwVersion;
_TCHAR szVersion[16];
...
// get the DLL version number
dwVersion = UcanGetVersionEx (kVerTypeUserDll);

// convert into a string
_stprintf (szVersion, _T("V%d.%02d.%d"),
  USBCAN_MAJOR_VER(dwVersion),
  USBCAN_MINOR_VER(dwVersion),
  USBCAN_RELEASE_VER(dwVersion));
...
### Table 14: Constants for the type of version information for function UcanGetVersionEx()

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Available for version</th>
</tr>
</thead>
<tbody>
<tr>
<td>kVerTypeUserDll</td>
<td>0x0001</td>
<td>Returns the version of the DLL.</td>
<td>Yes, Yes, Yes</td>
</tr>
<tr>
<td>kVerTypeUserLib</td>
<td>0x0002</td>
<td>Returns the version of the file USBCAN.SYS (device driver).</td>
<td>Yes, Yes, No</td>
</tr>
<tr>
<td>kVerTypeSysDrv</td>
<td>0x0003</td>
<td>Returns the version of the file UCANNET.SYS (network driver).</td>
<td>Yes, Yes, Yes</td>
</tr>
<tr>
<td>kVerTypeSysLd</td>
<td>0x0004</td>
<td>Obsolete - returns the version of loader USBCANLD.SYS for USB-CANmodul of first generation (GW-001).</td>
<td>Yes, No, No</td>
</tr>
<tr>
<td>kVerTypeSysL2</td>
<td>0x0005</td>
<td>Obsolete - returns the version of loader USBCANL2.SYS for USB-CANmodul of second generation (GW-002).</td>
<td>Yes, No, No</td>
</tr>
<tr>
<td>kVerTypeSysL3</td>
<td>0x0006</td>
<td>Obsolete - returns the version of loader USBCANL3.SYS for USB-CANmodul8/16 of third generation.</td>
<td>Yes, Yes, No</td>
</tr>
<tr>
<td>kVerTypeSysL4</td>
<td>0x0007</td>
<td>Returns the version of the file USBCANL4.CPL (USB-CANmodul Control from Windows Control Panel).</td>
<td>Yes, Yes, Yes</td>
</tr>
<tr>
<td>kVerTypeSysL5</td>
<td>0x0008</td>
<td>Returns the version of loader USBCANL5.SYS for USB-CANmodul2 of third generation.</td>
<td>Yes, Yes, No</td>
</tr>
<tr>
<td>kVerTypeCpl</td>
<td>0x0009</td>
<td>Returns the version of loader USBCANL21.SYS for USB-CANmodul2.</td>
<td>No, Yes, No</td>
</tr>
<tr>
<td>kVerTypeL21</td>
<td>0x000A</td>
<td>Returns the version of loader USBCANL22.SYS for USB-CANmodul1.</td>
<td>No, Yes, No</td>
</tr>
<tr>
<td>kVerTypeL22</td>
<td>0x000B</td>
<td>Returns the version of loader USBCANL23.SYS for USB-CANmodul8 and/or USB-CANmodul16.</td>
<td>No, Yes, No</td>
</tr>
<tr>
<td>kVerTypeL23</td>
<td>0x000C</td>
<td>Returns the version of the Extended Loader USBCANLEX.SYS for all USB-CANmodul types of 4th generation.</td>
<td>No, No, Yes</td>
</tr>
</tbody>
</table>

**Note:**
The function *UcanGetVersionEx()* returns the value 0x00000000 if the appropriate software module is not available or if an unknown type is used for the parameter *VerType_p*. 
## Function: UcanGetFwVersion

**Syntax:**

```
DWORD PUBLIC UcanGetFwVersion (tUcanHandle UcanHandle_p);
```

**Usability:**

`HW_INIT, CAN_INIT`

**Description:**

This function returns the version number of the firmware in the USB-CANmodul.

**Parameter:**

- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()`, or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.

**Return:**

Firmware version number as DWORD using the following format:

- Bit 0-7: Version (use macro `USBCAN_MAJOR_VER`)
- Bit 8-15: Revision (use macro `USBCAN_MINOR_VER`)
- Bit 16-31: Release (use macro `USBCAN_RELEASE_VER`)

The version number format is the same format as in the function `UcanGetVersionEx()`.

## Function: UcanInitHwConnectControl

**Syntax:**

```
UCANRET PUBLIC UcanInitHwConnectControl (tConnectControlFkt pfnConnectControl_p);
```

**Usability:**

`DLL_INIT, HW_INIT, CAN_INIT`

**Description:**

Initializes the supervision for recently connected USB-CANmodul devices. If a new module is connected to the PC, the callback function that is indicated in the parameter will be called. This callback function is also called if a module is disconnected from the PC.

Alternatively the function `UcanInitHwConnectControlEx()` can be used to pass a user-defined parameter to the callback handler.

**Parameter:**

- **pfnConnectControl_p:** Pointer to the connect control callback function that has to be called if a new USB-CANmodul is connected or disconnected. This pointer must not be NULL!

**Return:**

Error code of the function – refer to **Table 25**

The callback function must have the following format (refer to section 4.3.7.1):

```
void PUBLIC AppConnectControlCallback (BYTE bEvent_p, DWORD dwParam_p);
```
### Function: UcanInitHwConnectControlEx

#### Syntax:
```
UCANRET PUBLIC UcanInitHwConnectControlEx (tConnectControlFktEx pfConnectControlEx_p, void* pCallbackArg_p);
```

#### Usability:
- DLL_INIT, HW_INIT, CAN_INIT

#### Description:
Initializes the supervision for recently connected USB-CANmoduls as function `UcanInitHwConnectControl()` does. Unlike function `UcanInitHwConnectControl()`, this function has an additional parameter, which is also passed to the callback function. This parameter can be used to handle user-specific information, such as the used CAN instance for example.

#### Parameter:
- `pfConnectControlEx_p`: Pointer to the connect control callback function that has to be called if a new USB-CANmodul is connected or disconnected. This pointer must not be NULL!
- `pCallbackArg_p`: User-specific parameter that is passed to the callback function as well. This parameter may be NULL.

#### Return:
Error code of the function – refer to Table 25

The callback function must have the following format (refer to section 4.3.7.1):
```
void PUBLIC AppConnectControlCallbackEx (DWORD dwEvent_p, DWORD dwParam_p, void* pArg_p);
```

**Attention:**
This function must not be used simultaneously with function `UcanInitHwConnectControl()` within the same application!

### Function: UcanDeinitHwConnectControl

#### Syntax:
```
UCANRET PUBLIC UcanDeinitHwConnectControl (void);
```

#### Usability:
- DLL_INIT, HW_INIT, CAN_INIT

#### Description:
This function finishes the monitoring of the recently connected or disconnected USB-CANmodul devices. This function must be called before closing the application but only if the function `UcanInitHwConnectControl()` or `UcanInitHwConnectControlEx()` was called before.

#### Parameter:
- none

#### Return:
Error code of the function – refer to Table 25
**Function:** UcanEnumerateHardware

**Syntax:**
```c
DWORD PUBLIC UcanEnumerateHardware ( 
    tUcanEnumCallback pfEnumCallback_p, 
    void* pCallbackArg_p, 
    BOOL fEnumUsedDevs_p, 
    BYTE bDeviceNrLow_p, 
    BYTE bDeviceNrHigh_p, 
    DWORD dwSerialNrLow_p, 
    DWORD dwSerialNrHigh_p, 
    DWORD dwProductCodeLow_p, 
    DWORD dwProductCodeHigh_p);
```

**Usability:** DLL_INIT

**Description:** This function scans all USB-CANmodul devices connected at the host and calls a callback function for each found module. The amount of the USB-CANmodul devices to be found can be limited by filter parameters. Within the callback function the user can decide whether the found USB-CANmodul should be automatically initialized by the DLL. In this case the module changes to the state HW_INIT.

**Parameter:**
- `pfEnumCallback_p`: Pointer to the Enumeration Callback Function which is called for each found USB-CANmodul. This callback function is not called if the filter parameters does not match. This parameter must not be NULL.
- `pCallbackArg_p`: User-specific parameter that is passed to the callback function as well.
- `fEnumUsedDevs_p`: Set to TRUE if USB-CANmodul devices shall be found too which are currently exclusively used by another application. These modules cannot be used by the own application instance.
- `bDeviceNrLow_p, bDeviceNrHigh_p`: Filter parameters for the device number. The value `bDeviceNrLow_p` specifies the lower limit and the value `bDeviceNrHigh_p` specifies the upper limit of the device number which have to be found.
- `dwSerialNrLow_p, dwSerialNrHigh_p`: Filter parameters for the serial number. The values specifies the lower and upper limit of the serial number area which have to be found. Possible values are shown in Table 18.
- `dwProductCodeLow_p, dwProductCodeHigh_p`: Filter parameters for the Product-Code. The values specifies the lower and upper limit of the Product-Code area which have to be found.

**Return:** This function returns the number of found USB-CANmodul devices (logical modules). The value includes the modules which are exclusively used by other applications if parameter `fEnumUsedDevs_p` is set to TRUE.

**Example 1:**
```c
DWORD dwFoundModules;
...
// find all USB-CANmoduls, which are NOT used by other applications
dwFoundModules = UcanEnumerateHardware (AppEnumCallback, NULL, 
    FALSE, 
    0, ~0, // no limitation of device number  
    0, ~0, // no limitation of serial number  
    0, ~0); // no limitation of Product-Code  
...```

© SYSTEC electronic AG 2019  L-487e_02_05  Page 61
Example 2:

```c
DWORD dwFoundModules;
...
// find all USB-CANmoduls of typ USB-CANmodul1 (G4)
dwFoundModules = UcanEnumerateHardware(AppEnumCallback, NULL, FALSE,
0, ~0, // no limitation of device number
0, ~0, // no limitation of serial number
USBCAN_PRODCODE_PID_BASIC_G4, USBCAN_PRODCODE_PID_BASIC_G4);
...
```

Example 3:

```c
DWORD dwFoundModules;
DWORD dwSerialNr;
...
// find all logical modules of an USB-CANmodul16 (G3)
dwSerialNr = 123456; // <-- serial number at the sticker at the device case
dwFoundModules = UcanEnumerateHardware(AppEnumCallback, NULL, FALSE,
0, ~0, // no limitation of device number
(dwSerialNr * 1000) + 1, (dwSerialNr * 1000) + 8,
USBCAN_PRODCODE_PID_USBCAN16_G4, USBCAN_PRODCODE_PID_USBCAN16_G4);
...
```

Also refer to example on page 132.
Function: UcanInitHardware

Syntax:

```
UCANRET PUBLIC UcanInitHardware ( 
    tUcanHandle* pUcanHandle_p,
    BYTE     bDeviceNr_p,
    tCallbackFkt pfnEventCallback_p);
```

Usability: DLL_INIT

Description: Initializes an USB-CANmodul. The software changes into the state HW_INIT. From this point, the functions in section HW_INIT can be called (refer to Table 12). If the function was executed successfully, the function transfers an USBCAN handle to the variable addressed by the pointer `pUcabHandle_p`. Other functions communicating with device have to be called with this handle.

Alternatively the function `UcanInitHardwareEx()` can be used to pass a user-defined parameter to the callback handler. This function must be used if a multi-channel USB-CANmodul is used and the event callback handler is not NULL.

Parameter:

- **pUcanHandle_p**: Pointer to the variable for the USBCAN Handle. This pointer must not be NULL!
- **bDeviceNr_p**: Device number of the USB-CANmodul (0 – 254). The value USBCAN_ANY_MODULE (= 255) let initialize the first allocated USB-CANmodul.
- **pfnEventCallback_p**: Pointer to an event callback function related to this USB-CANmodul. This value can be NULL. In this case no callback function will be called if corresponding events appear. This address can be same as one that is already registered for other USB-CANmoduls, because the callback function passes the associated USBCAN handle.

Return: Error code of the function – refer to Table 25

The event callback function must have the following format (refer to section 4.3.7.2):

```
void PUBLIC AppEventCallback (tUcanHandle UcanHandle_p, BYTE bEvent_p);
```

Example:

```
UCANRET bRet;
    tUcanHandle UcanHandle;
...
// initializes an USB-CANmodul without callback function
    bRet = UcanInitHardware (&UcanHandle, USBCAN_ANY_MODULE, NULL);
    ...
```
**Function:** UcanInitHardwareEx

| Syntax: | UCANRET PUBLIC UcanInitHardwareEx ( tUcanHandle* pUcanHandle_p,  
| | BYTE bDeviceNr_p,  
| | tCallbackFkt pfnEventCallbackEx_p,  
| | void* pCallbackArg_p); |

**Usability:** DLL_INIT

**Description:** Initializes an USB-CANmodul. The software changes into the state HW_INIT. From this point, the functions in section HW_INIT can be called (refer to Table 12). If the function was executed successfully, the function transfers an USBCAN handle to the variable addressed by the pointer pUcabHandle_p. Other functions communicating with device have to be called with this handle.

Unlike function UcanInitHardware(), this function has an additional parameter, which is also passed to the event callback function. Additionally the CAN channel is passed to the event callback function.

**Parameter:**

- **pUcanHandle_p:** Pointer to the variable for the USBCAN Handle. This pointer must not be NULL!
- **bDeviceNr_p:** Device number of the USB-CANmodul (0 – 254). The value USBCAN_ANY_MODULE (= 255) let initialize the first allocated USB-CANmodul.
- **pfnEventCallbackEx_p:** Pointer to an event callback function related to this USB-CANmodul. This value can be NULL. In this case no callback function will be called if corresponding events appear. This address can be same as one that is already registered for other USB-CANmodul devices, because the callback function passes the associated USBCAN handle.
- **pCallbackArg_p:** User-specific parameter that is passed to the event callback function as well. This value can be NULL.

**Return:** Error code of the function – refer to Table 25

The event callback function must have the following format (refer to section 4.3.7.2):

```
void PUBLIC AppEventCallbackEx (tUcanHandle UcanHandle_p, BYTE bEvent_p,  
| BYTE bChannel_p, void* pArg_p); 
```

**Example:**

```
UCANRET bRet;  
tUcanHandle UcanHandle;  
...  
// initializes an USB-CANmodul without callback function  
bRet = UcanInitHardwareEx (&UcanHandle, USBCAN_ANY_MODULE, NULL, NULL);  
... 
```
**Function:** UcanInitHardwareEx2

**Syntax:**
```
UCANRET PUBLIC UcanInitHardwareEx2 ( 
   tUcanHandle* pUcanHandle_p, 
   DWORD dwSerialNr_p, 
   tCallbackFkt pfEventCallbackEx_p, 
   void* pCallbackArg_p);
```

**Usability:** DLL_INIT

**Description:** Initializes an USB-CANmodul as alternative to the functions `UcanInitHardware()` and `UcanInitHardwareEx()`. Instead of passing the device number the serial number is passed to identify the USB-CANmodul.

**Parameter:**

- **pUcanHandle_p:** Pointer to the variable for the USB CAN Handle. This pointer must not be NULL!
- **dwSerialNr_p:** Serial number of the USB-CANmodul (at the bar code sticker at the device’s case). For the logical modules of an 8 or 16 channel device the serial number must be calculated with the following formula:
  
  \[
  dwSerialNr = BarCodeNr \times 1000 + n;
  \]

  where \( n \) is the number of the logical module beginning with 1.
- **pfEventCallbackEx_p:** Pointer to an event callback function related to this USB-CANmodul. This value can be NULL. In this case no callback function will be called if corresponding events appear. This address can be same as one that is already registered for other USB-CANmodul devices, because the callback function passes the associated USBCan handle.
- **pCallbackArg_p:** User-specific parameter that is passed to the event callback function as well. This value can be NULL.

**Return:** Error code of the function – refer to Table 25

The event callback function must have the following format (refer to section 4.3.7.2):
```
void PUBLIC AppEventCallbackHandlerEx (tUcanHandle UcanHandle_p, BYTE bEvent_p, BYTE bChannel_p, void* pArg_p);
```

**Example:**
```
#define APP_BARCODE_NR 123456

UCANRET bRet;
tUcanHandle aUcanHandle[4];
DWORD dwSerialNr;
DWORD dwLogDevice;
...
for (dwLogDevice = 0; dwLogDevice <= 4; dwLogDevice++)
{
   dwSerialNr = APP_BARCODE_NR * 1000 + (dwLogDevice + 1);

   // initializes an USB-CANmodul without callback function
   bRet = UcanInitHardwareEx2 (&aUcanHandle[dwLogDevice],
                              dwSerialNr, NULL, NULL);
}
...
**Function:** UcanDeinitHardware

**Syntax:**
```c
UCANRET PUBLIC UcanDeinitHardware (tUcanHandle UcanHandle_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:** Shuts down an initialized USB-CANmodul that was initialized with UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware(). The software returns to the state DLL_INIT. After the function call, the USB CAN handle is not valid. That means, execution of the valid functions (see Table 4) for HW_INIT and CAN_INIT is no longer possible.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

**Return:** Error code of the function – refer to Table 25

**Note:**
This function has to be called before closing the application, otherwise other applications are no longer able to access this specific USB-CANmodul.

---

**Function:** UcanGetModuleTime

**Syntax:**
```c
UCANRET PUBLIC UcanGetModuleTime (tUcanHandle UcanHandle_p, DWORD* pdwTime_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:** This function reads the current time stamp from the device.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **pdwTime_p:** Pointer to a variable where the time stamp is to be stored to. This parameter must not be NULL!

**Return:** Error code of the function – refer to Table 25

**Note:**
The execution of this function as well as the transfer of the time stamp needs run-time. In other words, after this function has returned successfully, the time stamp will be out-dated. The accuracy of this time stamp depends on many factors and is unpredictable on non-real-time operating systems.

The base time of the time stamp is 1 millisecond as long as the flag kUcanModeHighResTimer is not set to the parameter m_bMode of structure tUcanInitCanParam passed to the function UcanInitCanEx() or UcanInitCanEx2(). If the flag kUcanModeHighResTimer is set in parameter m_bMode of structure tUcanInitCanParam then the time stamp returns in multiple of 100 microseconds.
**Function:** UcanSetDeviceNr  

**Syntax:**
```
UCANRET PUBLIC UcanSetDeviceNr (tUcanHandle UcanHandle_p,  
    BYTE bDeviceNr_p);
```

**Usability:** HW_INIT, CAN_INIT  

**Description:** This function writes a new device number to the USB-CANmodul.

**Parameter:**  
- **UcanHandle_p:** USBCAN handle that was received with the function *UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2()* as well as *UcanEnumerateHardware().*  
- **bDeviceNr_p:** New device number. Valid values are 0 to 254.

**Return:** Error code of the function – refer to *Table 25*

---

**Function:** UcanInitCan  

**Syntax:**
```
UCANRET PUBLIC UcanInitCan (tUcanHandle UcanHandle_p,  
    BYTE bBTR0_p, BYTE bBTR1_p,  
    DWORD dwAMR_p, DWORD dwACR_p);
```

**Usability:** HW_INIT, only single CAN-channel devices  

**Description:** Initializes the CAN interface of an USB-CANmodul. The software changes into the state CAN_INIT. From this point, the functions in section CAN_INIT can be called (refer to *Table 12*). This API function is obsolete. We recommend to use the function *UcanInitCanEx() and/or UcanInitCanEx2().*

**Parameter:**  
- **UcanHandle_p:** USBCAN handle that was received with the function *UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2()* as well as *UcanEnumerateHardware().*  
- **bBTR0_p, bBTR1_p:** Baud rate register 0 and 1 to select the CAN baud rate for a SJA1000 CAN controller (refer to *section 4.3.4*).  
- **dwAMR_p, dwACR_p:** Acceptance Mask and Code Register to configure the hardware filter for receiving CAN messages (refer to *section 4.3.5*).

**Return:** Error code of the function – refer to *Table 25*
Function: UcanInitCanEx

Syntax: UCANRET PUBLIC UcanInitCanEx (tUcanHandle UcanHandle_p, tUcanInitCanParam* pInitCanParam_p);

Usability: HW_INIT, only single CAN-channel devices

Description: Initializes the CAN interface of an USB-CANmodul. The software changes into the state CAN_INIT. From this point, the functions in section CAN_INIT can be called (refer to Table 12).

This API function is an alternative function for UcanInitCan() and/or UcanInitCanEx2().

Parameter:

UcanHandle_p: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

pInitCanParam_p: Pointer to a variable of type tUcanInitCanParam containing all CAN initialization parameters.

Return: Error code of the function – refer to Table 25

typedef struct
{
    DWORD m_dwSize;
    BYTE m_bMode;
    BYTE m_bBTR0;
    BYTE m_bBTR1;
    BYTE m_bOCR;
    DWORD m_dwAMR;
    DWORD m_dwACR;
    DWORD m_dwBaudrate;
    WORD m_wNrOfRxBufferEntries;
    WORD m_wNrOfTxBufferEntries;
} tUcanInitCanParam;

Parameter:

m_dwSize: Size of this structure in bytes. Always set it to the value returned by sizeof(tUcanInitCanParam) before calling the function UcanInitCanEx() or UcanInitCanEx2().

m_bMode: The CAN mode containing flags affecting the behavior of the transmission and reception of CAN messages (refer to Table 15). These flags can be combined.

m_bBTR0, m_bBTR1: Baud rate register 0 and 1 to select the CAN baud rate for a SJA1000 CAN controller (refer to section 4.3.4). These two parameters are obsolete (refer to following note).

m_bOCR: This parameter is obsolete. Always set it to the pre-defined value USBCAN_OCR_DEFAULT.

m_dwAMR, m_dwACR: Acceptance Mask and Code Register to configure the hardware filter for receiving CAN messages (refer to section 4.3.5).

m_dwBaudrate: Baud rate register to select the CAN baud rate for an USB-CANmodul of third or fourth generation (refer to section 4.3.4).

m_wNrOfRxBufferEntries, m_wNrOfTxBufferEntries: Number of entries in receive and transmit buffer within the DLL. Set these parameters to zero if the DLL shall use the default buffer size of 1024 entries.
Note:
The configuration of the baud rate differs significantly between the older USB-CANmodul versions and the all USB-CANmodul devices of third and fourth generation. If you need to support older hardware versions as they are described in this manual, the standardized baud rate values for BTR0 and BTR1 have to be used to specify the CAN baud rate (refer to section 4.3.4). Therefore set dwBaudrate to the pre-defined value USBCAN_BAUDEX_USE_BTR01. Otherwise set both BTR0 and BTR1 to zero and set the appropriate register value to dwBaudrate.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kUcanModeNormal</td>
<td>0x00</td>
<td>normal transmit- and receive mode</td>
</tr>
<tr>
<td>kUcanModeListenOnly</td>
<td>0x01</td>
<td>listen-only mode; transmitted CAN messages are not sent out via CAN-bus. Received CAN-messages of remote nodes are not acknowledged.</td>
</tr>
<tr>
<td>kUcanModeTxEcho</td>
<td>0x02</td>
<td>UcanReadCanMsg() and/or UcanReadCanMsgEx() also returns transmitted messages as so-called “transmit echo”.</td>
</tr>
<tr>
<td>kUcanModeHighResTimer</td>
<td>0x08</td>
<td>The time stamp of CAN-message structure tCanMsgStruct is high-resolution for received CAN messages. This means the value in the member-variable m_dwTime has 100µs resolution (instead 1ms). An overrun of the 32-Bit value is reached every 4d:23h:18min:16.7296sec (instead 49d:17h:2min:47.295sec). The resolution of returned value of API function UcanGetModuleTime() is also changed with this configuration.</td>
</tr>
</tbody>
</table>

Function: UcanInitCanEx2

Syntax:
UCANRET PUBLIC UcanInitCanEx2 (tUcanHandle UcanHandle_p, BYTE bChannel_p, tUcanInitCanParam* pInitCanParam_p);

Usability: HW_INIT

Description: Initializes the CAN interface of an USB-CANmodul. The software changes into the state CAN_INIT. From this point, the functions in section CAN_INIT can be called (refer to Table 12).

This API function is an alternative function for UcanInitCan() and/or UcanInitCanEx().

Parameter:

UcanHandle_p:
USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

bChannel_p:
CAN channel, which is to be used. USBCAN_CHANNEL_CHAN_0 for CAN channel 0 USBCAN_CHANNEL_CHAN_1 for CAN channel 1

pInitCanParam_p:
Pointer to a variable of type tUcanInitCanParam containing all CAN initialization parameters.

Return:
Error code of the function – refer to Table 25

Example: refer to example on page 119.
### Function: UcanSetTxTimeout

**Syntax:**

```c
UCANRET PUBLIC UcanSetTxTimeout (tUcanHandle UcanHandle_p,
    BYTE bChannel_p,
    DWORD dwTxTimeout_p);
```

**Usability:**

CAN_INIT, only multi CAN-channel devices

**Description:**

If this function is called with a timeout value greater than 0 milliseconds then firmware controls all transmit CAN messages by this timeout value. If a CAN message cannot be sent during this timeout (notified by reception of the acknowledge bit on CAN bus) then firmware changes to a special state whereas all further transmit CAN messages for the specified channel will be deleted automatically. At each deleted transmit CAN message firmware sets the new CAN driver state USBCAN_CANERR_TXMSGLOST. When the CAN message could be sent later then firmware leaves this special state.

This feature is to prevent that transmit CAN messages of a channel blocks transmit CAN messages of the other channel caused by not connected remote CAN device or any physical problems on CAN bus.

**Parameter:**

- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **bChannel_p:** CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1
- **dwTxTimeout_p:** Transmission Timeout in milliseconds. The value 0 switches off the timeout control.

**Return:**

Error code of the function – refer to Table 25

---

### Function: UcanResetCan

**Syntax:**

```c
UCANRET PUBLIC UcanResetCan (tUcanHandle UcanHandle_p);
```

**Usability:**

HW_INIT, CAN_INIT, only for single CAN-channel devices

**Description:**

Resets the CAN controller in the USB-CANmodul and erases the CAN message buffer. This function needs to be called if a BUSOFF event occurred. The CAN status error (readable via `UcanGetStatus()` or `UcanGetStatusEx()`) is also cleared.

**Parameter:**

- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.

**Return:**

Error code of the function – refer to Table 25
**Function:** UcanResetCanEx

**Syntax:**

```c
UCANRET PUBLIC UcanResetCanEx (tUcanHandle UcanHandle_p,
BYTE  bChannel_p,
DWORD dwResetFlags_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:** Resets several features of a separate CAN channel of an USB-CANmodul. This API function is an extended version of function UcanResetCan().

**Parameter:**

- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **bChannel_p:** CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1
- **dwResetFlags_p:** The flags of this parameter specify which components are to be reset (refer to Table 16 and Table 17). The logical combination of different flags is possible.

**Return:** Error code of the function – refer to Table 25

### Table 16: Constants for Reset Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_RESET_ALL</td>
<td>0x00000000</td>
<td>Reset all components. However, the firmware is not reset completely.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_STATUS</td>
<td>0x00000001</td>
<td>Skip reset of the CAN error status.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_CANCTRL</td>
<td>0x00000002</td>
<td>Skip reset of the CAN controller.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_TXCOUNTER</td>
<td>0x00000004</td>
<td>Skip reset of the transmit message counter.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_RXCOUNTER</td>
<td>0x00000008</td>
<td>Skip reset of the receive message counter.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_TXBUFFER_CH</td>
<td>0x00000010</td>
<td>Skip reset of the transmit buffers of a specific CAN-channel (CAN-channel is specified by parameter <code>bChannel_p</code>).</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_TXBUFFER_DLL</td>
<td>0x00000020</td>
<td>Skip reset of the transmit buffer for both CAN-channels within the DLL.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_TXBUFFER_FW</td>
<td>0x00000080</td>
<td>Skip reset of the transmit buffers of both CAN-channels within the device’s firmware.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_RXBUFFER_CH</td>
<td>0x00000100</td>
<td>Skip reset of the receive buffers of a specific CAN-channel (CAN-channel is specified by parameter <code>bChannel_p</code>).</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_RXBUFFER_DLL</td>
<td>0x00000200</td>
<td>Skip reset of both receive message counters within the DLL.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_RXBUFFER_SYS</td>
<td>0x00000400</td>
<td>Skip reset of the receive buffer within the kernel mode driver.</td>
</tr>
<tr>
<td>USBCAN_RESET_NO_RXBUFFER_FW</td>
<td>0x00000800</td>
<td>Skip reset of receive buffer within the device’s firmware.</td>
</tr>
<tr>
<td>USBCAN_RESET_FIRMWARE</td>
<td>0xFFFFFFFF</td>
<td>Complete reset of the device firmware. The device will be automatically disconnected from the USB interface and reconnected again.</td>
</tr>
</tbody>
</table>
Table 17: Constants as pre-defined combinations for Reset Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_RESET_ONLY_STATUS</td>
<td>0x0000FFFE</td>
<td>Reset of the CAN error status only.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_CANCTRL</td>
<td>0x0000FFFD</td>
<td>Only resets the CAN controller of the USB-CANmodul.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_RXBUFFER_FW</td>
<td>0x0000F7FF</td>
<td>Only resets the receive buffer within the firmware of the USB-CANmodul.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_TXBUFFER_FW</td>
<td>0x0000FF7F</td>
<td>Only resets the transmit buffer within the firmware of the USB-CANmodul.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_RXCHANNEL_BUFF</td>
<td>0x0000FEFF</td>
<td>Reset of the receive buffer of only one CAN-channel.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_TXCHANNEL_BUFF</td>
<td>0x0000FFEF</td>
<td>Reset of the transmit buffer of only one CAN-channel.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_RX_BUFF</td>
<td>0x0000F0F7</td>
<td>Reset of the receive buffers in all software parts and reset of the receive message counter.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_TX_BUFF</td>
<td>0x0000FF0B</td>
<td>Reset of the transmit buffers in all software parts and reset of the transmit message counter.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_ALL_BUFF</td>
<td>0x0000F003</td>
<td>Reset of all message buffers (receive and transmit buffers) in all software parts and reset of the reception and transmit message counters.</td>
</tr>
<tr>
<td>USBCAN_RESET_ONLY_ALL_COUNTER</td>
<td>0x0000FFF3</td>
<td>Reset of all reception and transmit counters.</td>
</tr>
</tbody>
</table>

Important:

If the constants USBCAN_RESET_NO... must be combined, a logical OR has to be used.

Example:

```c
dwFalgs = USBCAN_RESET_NO_COUNTER_ALL | USBCAN_RESET_NO_BUFFER_ALL;
```

If the constants USBCAN_RESET_ONLY... must be combined, a logical AND has to be used.

Example:

```c
dwFalgs = USBCAN_RESET_ONLY_RX_BUFF & USBCAN_RESET_ONLY_STATUS;
```
### Function: **UcanDeinitCan**

<table>
<thead>
<tr>
<th><strong>Syntax:</strong></th>
<th>UCANRET PUBLIC UcanDeinitCan (tUcanHandle UcanHandle_p);</th>
</tr>
</thead>
</table>

**Usability:**
CAN_INIT, only single CAN-channel devices

**Description:**
Shuts down the CAN interface of an USB-CANmodul. The software changes back to the state HW_INIT.
After calling this function, all CAN messages left in receive buffer of the firmware are ignored and not transferred to the PC. These CAN messages are lost.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function *UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2()* as well as *UcanEnumerateHardware().*

**Return:**
Error code of the function – refer to *Table 25*

### Function: **UcanDeinitCanEx**

<table>
<thead>
<tr>
<th><strong>Syntax:</strong></th>
<th>UCANRET PUBLIC UcanDeinitCanEx (tUcanHandle UcanHandle_p, BYTE bChannel_p);</th>
</tr>
</thead>
</table>

**Usability:**
CAN_INIT

**Description:**
Shuts down the CAN interface of an USB-CANmodul. The software changes back to the state HW_INIT.
After calling this function, all CAN messages left in receive buffer of the firmware are ignored and not transferred to the PC. These CAN messages are lost.

This API function is an extended function to be used for multi-channel devices.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function *UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2()* as well as *UcanEnumerateHardware().*
- **bChannel_p:** CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1

**Return:**
Error code of the function – refer to *Table 25*
**Function:** UcanGetHardwareInfo

**Syntax:**
```
UCANRET PUBLIC UcanGetHardwareInfo (tUcanHandle UcanHandle_p,
   tUcanHardwareInfo* pHwInfo_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:** This function returns the hardware information of an USB-CANmodul. This function is especially useful if an USB-CANmodul has been initialized with the device number USBCAN_ANY_MODULE. Afterwards, the hardware information contains the device number of the initialized USB-CANmodul.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **pHwInfo_p:** Pointer to the structure `tUcanHardwareInfo` containing the hardware information (see description below). This pointer must not be NULL.

**Return:** Error code of the function – refer to Table 25

```c
typedef struct
{
  BYTE          m_bDeviceNr;
  tUcanHandle   m_UcanHandle;
  DWORD         m_dwReserved;
  BYTE          m_bBTR0;
  BYTE          m_bBTR1;
  BYTE          m_bOCR;
  DWORD         m_dwAMR;
  DWORD         m_dwACR;
  BYTE          m_bMode;
  DWORD         m_dwSerialNr;
} tUcanHardwareInfo;
```

**Parameter:**
- **m_bDeviceNr:** Device number of the USB-CANmodul device.
- **m_UcanHandle:** USBCAN handle returned by `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **m_dwReserved:** Reserved
- **m_bBTR0, m_bBTR1:** Baud rate register 0 and 1 to select the CAN baud rate for a SJA1000 CAN controller (refer to section 4.3.4).
- **m_bOCR:** This parameter is obsolete.
- **m_dwAMR, m_dwACR:** Acceptance Mask and Code Register to configure the hardware filter for receiving CAN messages (refer to section 4.3.5).
- **m_bMode:** The CAN mode containing flags affecting the behavior of the transmission and reception of CAN messages (refer to Table 15). These flags can be combined.
- **m_dwSerialNr:** Serial number of the USB-CANmodul (at the bar code sticker at the device’s case).
Example:

UCANRET bRet;
tUcanHandle UcanHandle;
tUcanHardwareInfo HwInfo;
_TCHAR szDeviceNr[24];
...
// initialize USB-CANmodul
bRet = UcanInitHardware (&UcanHandle, USBCAN_ANY_MODULE, NULL);

// no error?
if (bRet == USBCAN_SUCCESSFUL)
{
    // get hardware information
    UcanGetHardwareInfo (UcanHandle, &HwInfo);

    // change the device number into a string
    _stprintf (szDeviceNr, _T("device number = %d"),
               HwInfo.m_bDeviceNr);
    ...
}
...

Function: UcanGetHardwareInfoEx2

Syntax:  

UCANRET PUBLIC UcanGetHardwareInfoEx2 (tUcanHandle UcanHandle_p,
tUcanHardwareInfoEx* pHwInfoEx_p,
tUcanChannelInfo* pCanInfoCh0_p,
tUcanChannelInfo* pCanInfoCh1_p);

Usability: HW_INIT, CAN_INIT

Description: This function returns the extended hardware information of an USB-CANmodul. The hardware information of each CAN-channel is returned separately.

Parameter:

UcanHandle_p: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

pHwInfoEx_p: Pointer to the structure tUcanHardwareInfoEx containing the hardware information (see description below). This pointer must not be NULL.

pCanInfoCh0_p, pCanInfoCh1_p: Pointers to information structure used for CAN channel 0 and 1. This parameters may be set to NULL.

Return: Error code of the function – refer to Table 25

typedef struct
{
    DWORD m_dwSize;
    tUcanHandle m_UcanHandle;
    BYTE m_bDeviceNr;
    DWORD m_dwSerialNr;
    DWORD m_dwFwVersionEx;
    DWORD m_dwReserved;
    DWORD m_dwProductCode;
}
tUcanHardwareInfoEx;
**Parameter:**

- **m_dwSize:** Size of this structure in bytes. Always set it to the value returned by `sizeof(tUcanHardwareInfoEx)` before calling the function `UcanGetHardwareInfoEx2()`.

- **m_UcanHandle:** USBCAN handle returned by `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.

- **m_bDeviceNr:** Device number of the USB-CANmodul device.

- **m_dwSerialNr:** Serial number of the USB-CANmodul (at the bar code sticker at the device's case).

- **m_dwFwVersionEx:** Version of the firmware within the USB-CANmodul (refer to `UcanGetFwVersion()` for the format).

- **m_dwReserved:** Reserved

- **m_dwProductCode:** Type of the hardware (refer to **Table 18**)

**typedef struct**

```c
{
    DWORD    m_dwSize;
    BYTE     m_bMode;
    BYTE     m_bBTR0;
    BYTE     m_bBTR1;
    BYTE     m_bOCR;
    DWORD    m_dwAMR;
    DWORD    m_dwACR;
    DWORD    m_dwBaudrate;
    BOOL     m_fCanIsInit;
    WORD     m_wCanStatus;
}
```

**Parameter:**

- **m_dwSize:** Size of this structure in bytes. Always set it to the value returned by `sizeof(tUcanChannelInfo)` before calling the function `UcanGetHardwareInfoEx2()`.

- **m_bMode:** The CAN mode containing flags affecting the behavior of the transmission and reception of CAN messages (refer to **Table 15**). These flags can be combined.

- **m_bBTR0, m_bBTR1:** Baud rate register 0 and 1 to select the CAN baud rate for a SJA1000 CAN controller (refer to **section 4.3.4**).

- **m_bOCR:** This parameter is obsolete.

- **m_dwAMR, m_dwACR:** Acceptance Mask and Code Register to configure the hardware filter for receiving CAN messages (refer to **section 4.3.5**).

- **m_dwBaudrate:** Baud rate register to select the CAN baud rate for an USB-CANmodul of third or fourth generation (refer to **section 4.3.4**).

- **m_fCanIsInit:** If set to non-zero then the CAN interface of the USB-CANmodul is initialized by using the function `UcanInitCan()`, `UcanInitCanEx()` or `UcanInitCanEx2()`.

- **m_wCanStatus:** Last received CAN state (refer to `UcanGetStatus()` or `UcanGetStatusEx()`).
The 32-Bit value in \textit{m\_dwProductCode} of structure \textit{tUcanHardwareInfoEx} specifies the Hardware-Type of the USB-CANmodul with the lower 16 bits. Table 18 lists all possible values:

**Table 18: Constants for Product-Code / Hardware-Type**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_PRODCODE_PID_GW001</td>
<td>0x1100</td>
<td>USB-CANmodul (^1) of first generation (G1)</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_GW002</td>
<td>0x1102</td>
<td>USB-CANmodul (^1) of second generation (G2)</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_MULTIPORT</td>
<td>0x1103</td>
<td>Multiport CAN-to USB (^1) of third generation (G3) including 16 CAN channels</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_BASIC</td>
<td>0x1104</td>
<td>USB-CANmodul1 (^1) of third generation (G3) including one CAN channel</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_ADVANCED</td>
<td>0x1105</td>
<td>USB-CANmodul2 (^1) of third generation (G3) including 2 CAN channels</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_USBCAN8</td>
<td>0x1107</td>
<td>USB-CANmodul8 (^1) of third generation (G3) including 8 CAN channels</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_USBCAN16</td>
<td>0x1109</td>
<td>USB-CANmodul16 (^1) of third generation (G3) including 16 CAN channels</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_RESERVED3</td>
<td>0x1110</td>
<td>Reserved (^1)</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_ADVANCED_G4</td>
<td>0x1121</td>
<td>USB-CANmodul2 of fourth generation (G4) including 2 CAN channels</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_BASIC_G4</td>
<td>0x1122</td>
<td>USB-CANmodul1 of fourth generation (G4) including one CAN channel</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_RESERVED1</td>
<td>0x1144</td>
<td>Reserved (^1)</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_RESERVED2</td>
<td>0x1145</td>
<td>Reserved (^1)</td>
</tr>
<tr>
<td>USBCAN_PRODCODE_PID_RESERVED4</td>
<td>0x1162</td>
<td>Reserved (^1)</td>
</tr>
</tbody>
</table>

\(^1\) Not documented within the scope of this document.

Use the following macros are for getting information about the support of several new features:

**Macro:** USBCAN\_CHECK\_SUPPORT\_CYCLIC\_MSG

**Syntax:**

\[
\text{USBCAN\_CHECK\_SUPPORT\_CYCLIC\_MSG(pHwInfoEx\_p)}
\]

**Description:**

This Macro checks whether the logical USB-CANmodul supports the automatic transmission of cyclic CAN messages.

**Parameter:**

\(pHwInfoEx\_p\):

Pointer to the structure \textit{tUcanHardwareInfoEx} containing the hardware information returned by function \textit{UcanGetHardwareInfoEx2()}. This pointer must not be NULL.

**Return:**

FALSE (zero) or TRUE (non-zero)
### Macro: `USBCAN_CHECK_SUPPORT_TWO_CHANNEL`

**Syntax:**
```
USBCAN_CHECK_SUPPORT_TWO_CHANNEL(pHWInfoEx_p)
```

**Description:**
This Macro checks whether the logical USB-CANmodul supports two CAN-channels.

**Parameter:**
- `pHWInfoEx_p`: Pointer to the structure `tUcanHardwareInfoEx` containing the hardware information returned by function `UcanGetHardwareInfoEx2()`. This pointer must not be NULL.

**Return:**
FALSE (zero) or TRUE (non-zero)

### Macro: `USBCAN_CHECK_SUPPORT_TERM_RESISTOR`

**Syntax:**
```
USBCAN_CHECK_SUPPORT_TERM_RESISTOR(pHWInfoEx_p)
```

**Description:**
This Macro checks whether the logical USB-CANmodul supports to read back the state of the termination resistor (refer to section 2.3).

**Parameter:**
- `pHWInfoEx_p`: Pointer to the structure `tUcanHardwareInfoEx` containing the hardware information returned by function `UcanGetHardwareInfoEx2()`. This pointer must not be NULL.

**Return:**
FALSE (zero) or TRUE (non-zero)

### Macro: `USBCAN_CHECK_SUPPORT_USER_PORT`

**Syntax:**
```
USBCAN_CHECK_SUPPORT_USER_PORT(pHWInfoEx_p)
```

**Description:**
This Macro checks whether the logical USB CANmodul supports a programmable expansion port (refer to section 2.5).

**Parameter:**
- `pHWInfoEx_p`: Pointer to the structure `tUcanHardwareInfoEx` containing the hardware information returned by function `UcanGetHardwareInfoEx2()`. This pointer must not be NULL.

**Return:**
FALSE (zero) or TRUE (non-zero)
Macro: \texttt{USBCAN\_CHECK\_SUPPORT\_RBUSER\_PORT}

Syntax: \texttt{USBCAN\_CHECK\_SUPPORT\_RBUSER\_PORT(pHWInfoEx\_p)}

Description: This Macro checks whether the logical USB-CANmodul supports a programmable expansion port (refer to section 2.5) including the storing of the last output configuration to a non-volatile memory. After next power-on this configuration will be automatically set to the expansion port.

Parameter:
\begin{itemize}
\item \texttt{pHWInfoEx\_p}: Pointer to the structure \texttt{tUcanHardwareInfoEx} containing the hardware information returned by function \texttt{UcanGetHardwareInfoEx2()}. This pointer must not be NULL.
\end{itemize}

Return: FALSE (zero) or TRUE (non-zero)

Macro: \texttt{USBCAN\_CHECK\_SUPPORT\_RBCAN\_PORT}

Syntax: \texttt{USBCAN\_CHECK\_SUPPORT\_RBCAN\_PORT(pHWInfoEx\_p)}

Description: This Macro checks whether the logical USB-CANmodul supports a programmable CAN port (for low-speed CAN transceivers – refer to section 2.4) including the storing of the last output configuration to a non-volatile memory. After next power-on this configuration will be automatically set to the CAN port.

Parameter:
\begin{itemize}
\item \texttt{pHWInfoEx\_p}: Pointer to the structure \texttt{tUcanHardwareInfoEx} containing the hardware information returned by function \texttt{UcanGetHardwareInfoEx2()}. This pointer must not be NULL.
\end{itemize}

Return: FALSE (zero) or TRUE (non-zero)

Macro: \texttt{USBCAN\_CHECK\_SUPPORT\_UCANNET}

Syntax: \texttt{USBCAN\_CHECK\_SUPPORT\_UCANNET(pHWInfoEx\_p)}

Description: This Macro checks whether the logical USB-CANmodul supports the usage of the network driver (refer to section 3.9).

Parameter:
\begin{itemize}
\item \texttt{pHWInfoEx\_p}: Pointer to the structure \texttt{tUcanHardwareInfoEx} containing the hardware information returned by function \texttt{UcanGetHardwareInfoEx2()}. This pointer must not be NULL.
\end{itemize}

Return: FALSE (zero) or TRUE (non-zero)
**Macro:**  
USBCAN_CHECK_IS_SYSWORXX

**Syntax:**  
USBCAN_CHECK_IS_SYSWORXX(pHwInfoEx_p)

**Description:**  
This Macro checks whether the logical USB-CANmodul belongs to the sysWORXX series of USB-CANmodul (at least third generation - G3).

**Parameter:**  
- **pHwInfoEx_p:** Pointer to the structure tUcanHardwareInfoEx containing the hardware information returned by function UcanGetHardwareInfoEx2(). This pointer must not be NULL.

**Return:**  
FALSE (zero) or TRUE (non-zero)

---

**Macro:**  
USBCAN_CHECK_IS_G1  
USBCAN_CHECK_IS_G2  
USBCAN_CHECK_IS_G3  
USBCAN_CHECK_IS_G4

**Syntax:**  
USBCAN_CHECK_IS_G1(pHwInfoEx_p)  
USBCAN_CHECK_IS_G2(pHwInfoEx_p)  
USBCAN_CHECK_IS_G3(pHwInfoEx_p)  
USBCAN_CHECK_IS_G4(pHwInfoEx_p)

**Description:**  
This Macro checks whether the logical USB-CANmodul belongs to the first, second, third or fourth generation of USB-CANmodul (refer to Table 18).

**Parameter:**  
- **pHwInfoEx_p:** Pointer to the structure tUcanHardwareInfoEx containing the hardware information returned by function UcanGetHardwareInfoEx2(). This pointer must not be NULL.

**Return:**  
FALSE (zero) or TRUE (non-zero)
Example:

```c
UCANRET bRet;
tUcanHandle UcanHandle;
tUcanHardwareInfoEx HwInfoEx;
...
// init USB-CANmodul
bRet = UcanInitHardware (&UcanHandle, USBCAN_ANY_MODULE, NULL);
if (bRet == USBCAN_SUCCESSFUL)
{
    // prepare the hardware info structure
    memset (&HwInfoEx, 0, sizeof (HwInfoEx));
    HwInfoEx.m_dwSize = sizeof (HwInfoEx);

    // get the extended hardware information
    bRet = UcanGetHardwareInfoEx2 (UcanHandle, &HwInfoEx, NULL, NULL);
    if (bRet == USBCAN_SUCCESSFUL)
    {
        TRACE1 ("product code = 0x%04X\n",
                HwInfoEx->m_dwProductCode & USBCAN_PRODCODE_MASK_PID);

        // check whether two CAN-channels are supported
        if (USBCAN_CHECK_SUPPORT_TWO_CHANNEL (&HwInfoEx))
        {
            ...
        }
    }
}
...
```
### Function: UcanGetMsgCountInfo

**Syntax:**

```c
UCANRET PUBLIC UcanGetMsgCountInfo (tUcanHandle UcanHandle_p, tUcanMsgCountInfo* pMsgCountInfo_p);
```

**Usability:**

CAN.Init, only single-channel devices

**Description:**

Reads the counters for transmitted and received CAN messages from the device.

**Parameter:**

- **UcanHandle_p:** USBCAN handle that was received with the function  
  `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.

- **pMsgCountInfo_p:** Pointer to a structure of type `tUcanMsgCountInfo` where the counters are to be stored to (see below). This pointer must not be NULL.

**Return:**

Error code of the function – refer to Table 25

```c
typedef struct
{
    WORD m_wSentMsgCount;
    WORD m_wRecvdMsgCount;
} tUcanMsgCountInfo;
```

**Parameter:**

- **m_wSentMsgCount:** Number of transmitted CAN messages.

- **m_wRecvdMsgCount:** Number of received CAN messages.

---

### Function: UcanGetMsgCountInfoEx

**Syntax:**

```c
UCANRET PUBLIC UcanGetMsgCountInfoEx (tUcanHandle UcanHandle_p, 
                                          BYTE bChannel_p, 
                                          tUcanMsgCountInfo* pMsgCountInfo_p);
```

**Usability:**

CAN.Init

**Description:**

Reads the counters for transmitted and received CAN messages from the device. This API function is an extended version of function `UcanGetMsgCountInfo()`.

**Parameter:**

- **UcanHandle_p:** USBCAN handle that was received with the function  
  `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.

- **bChannel_p:** CAN channel, which is to be used.  
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1

- **pMsgCountInfo_p:** Pointer to a structure of type `tUcanMsgCountInfo` where the counters are to be stored to (refer to function `UcanGetMsgCountInfoEx()`). This pointer must not be NULL.

**Return:**

Error code of the function – refer to Table 25
**Function:** UcanGetStatus

**Syntax:**
```
UCANRET PUBLIC UcanGetStatus (tUcanHandle UcanHandle_p, tStatusStruct* pStatus_p);
```

**Usability:** HW_INIT, CAN_INIT, only single-channel devices

**Description:** This function returns the current error status from the USB-CANmodul. The error status must be cleared by calling the function UcanResetCan() or UcanResetCanEx().

**Parameter:**

- **UcanHandle_p:** USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **pStatus_p:** Pointer to a structure of type tStatusStruct where the error status is to be stored to (see below). This pointer must not be NULL.

**Return:** Error code of the function – refer to Table 25

If an error occurred on the USB-CANmodul, the red status LED starts blinking and a status notification is sent to the PC. If an event callback function has been passed to the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2(), this event callback function is called passing the event USBCAN_EVENT_STATUS. Indirectly call the function UcanGetStatus() or UcanGetStatusEx() to receive the error status.

```c
typedef struct
{
  WORD m_wCanStatus;
  WORD m_wUsbStatus;
} tStatusStruct;
```

**Parameter:**

- **m_wCanStatus:** CAN error status (refer to Table 19). More than one error status bit may be set.
- **m_wUsbStatus:** General device error status (refer to Table 20). More than one error status bit may be set.
### Table 19: Constants for CAN error status

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_CANERR_OK</td>
<td>0x0000</td>
<td>No error occurred.</td>
</tr>
<tr>
<td>USBCAN_CANERR_XMTFULL</td>
<td>0x0001</td>
<td>Transmit buffer in CAN controller is overrun.</td>
</tr>
<tr>
<td>USBCAN_CANERR_OVERRUN</td>
<td>0x0002</td>
<td>Receive buffer in CAN controller is overrun.</td>
</tr>
<tr>
<td>USBCAN_CANERR_BUSLIGHT</td>
<td>0x0004</td>
<td>Error limit 1 in CAN controller exceeded. The CAN controller is in state “Warning limit”.</td>
</tr>
<tr>
<td>USBCAN_CANERR_BUSHEAVY</td>
<td>0x0008</td>
<td>Error limit 2 in CAN controller exceeded. The CAN controller is in state “Error Passive”.</td>
</tr>
<tr>
<td>USBCAN_CANERR_BUSOFF</td>
<td>0x0010</td>
<td>CAN controller is in BUSOFF state.</td>
</tr>
<tr>
<td>USBCAN_CANERR_OVERRUN</td>
<td>0x0040</td>
<td>Receive buffer in module’s firmware is overrun.</td>
</tr>
<tr>
<td>USBCAN_CANERR_QXMTFULL</td>
<td>0x0080</td>
<td>Transmit buffer in module’s firmware is overrun.</td>
</tr>
<tr>
<td>USBCAN_CANERR_REGTEST</td>
<td>0x0100</td>
<td>Obsolete</td>
</tr>
<tr>
<td>USBCAN_CANERR_MEMTEST</td>
<td>0x0200</td>
<td>Obsolete</td>
</tr>
<tr>
<td>USBCAN_CANERR_TXMSGLOST</td>
<td>0x0400</td>
<td>A transmit CAN message was deleted automatically by the firmware because transmission timeout run over (refer to function <code>UcanSetTxTimeout()</code>).</td>
</tr>
</tbody>
</table>

### Table 20: Constants for general error status

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_USBERR_STATUS_TIMEOUT</td>
<td>0x2000</td>
<td>The USB-CANmodul has been reset because the status channel was not polled each second.</td>
</tr>
<tr>
<td>USBCAN_USBERR_WATCHDOG_TIMEOUT</td>
<td>0x4000</td>
<td>The USB-CANmodul has been reset because the internal watchdog was not triggered by the firmware.</td>
</tr>
</tbody>
</table>
### Function: UcanGetStatusEx

**Syntax:**

```
UCANRET PUBLIC UcanGetStatusEx (tUcanHandle UcanHandle_p, 
  BYTE bChannel_p, 
  tStatusStruct* pStatus_p);
```

**Usability:**

HW_INIT, CAN_INIT

**Description:**

This function returns the current error status of a specific CAN-channel from the USB-CANmodul. It is an extended version of function *UcanGetStatus()*.

**Parameter:**

- **UcanHandle_p**: USBCAN handle that was received with the function *UcanInitHardware()*,*UcanInitHardwareEx()* or *UcanInitHardwareEx2()* as well as *UcanEnumerateHardware()*.

- **bChannel_p**: CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1

- **pStatus_p**: Pointer to a structure of type *tStatusStruct* where the error status is to be stored to (refer to *UcanGetStatus()*). This pointer must not be NULL.

**Return:**

Error code of the function – refer to *Table 25*

### Function: UcanSetBaudrate

**Syntax:**

```
UCANRET PUBLIC UcanSetBaudrate (tUcanHandle UcanHandle_p, 
  BYTE bBTR0_p, BYTE bBTR1_p);
```

**Usability:**

CAN_INIT, only single-channel devices

**Description:**

Changes the baud rate configuration of the USB-CANmodul. This API function is obsolete. We recommend to use the function *UcanSetBaudrateEx()*.

**Parameter:**

- **UcanHandle_p**: USBCAN handle that was received with the function *UcanInitHardware()*,*UcanInitHardwareEx()* or *UcanInitHardwareEx2()* as well as *UcanEnumerateHardware()*.

- **bBTR0_p, bBTR1_p**: Baud rate register 0 and 1 to select the CAN baud rate for a SJA1000 CAN controller (refer to *section 4.3.4*).

**Return:**

Error code of the function – refer to *Table 25*
Function: UcanSetBaudrateEx

Syntax: UCANRET PUBLIC UcanSetBaudrateEx (tUcanHandle UcanHandle_p, BYTE bChannel_p, BYTE bBTR0_p, BYTE bBTR1_p, DWORD dwBaudrate_p);

Usability: CAN_INIT

Description: Changes the baud rate configuration of a specific CAN-channel of the USB-CANmodul.

This API function is an extended version of function UcanSetBaudrate().

Parameter:

UcanHandle_p: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

bChannel_p: CAN channel, which is to be used.

USBCAN_CHANNEL_CH0 for CAN channel 0
USBCAN_CHANNEL_CH1 for CAN channel 1

bBTR0_p, bBTR1_p: Baud rate register 0 and 1 to select the CAN baud rate for a SJA1000 CAN controller (refer to section 4.3.4).

dwBaudrate_p: Baud rate register for all USB-CANmodul devices of third or fourth generation (refer to section 4.3.4).

Return: Error code of the function – refer to Table 25

Note: The configuration of the baud rate differs significantly between the older USB-CANmodul versions and the all USB-CANmodul devices of third and fourth generation. If you need to support older hardware versions as they are described in this manual, the standardized baud rate values for bBTR0 and bBTR1 have to be used to specify the CAN baud rate (refer to section 4.3.4). Therefore set dwBaudrate to the pre-defined value USBCAN_BAUDEX_USE_BTR01. Otherwise set both bBTR0 and bBTR1 to zero and set the appropriate register value to dwBaudrate.

Function: UcanSetAcceptance

Syntax: UCANRET PUBLIC UcanSetAcceptance (tUcanHandle UcanHandle_p, DWORD dwAMR_p, DWORD dwACR_p);

Usability: CAN_INIT, only single-channel devices

Description: Changes the acceptance filter registers of the USB-CANmodul for receiving CAN messages.

Parameter:

UcanHandle_p: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

dwAMR_p, dwACR_p: Acceptance Mask and Code Register to configure the hardware filter for receiving CAN messages (refer to section 4.3.5).

Return: Error code of the function – refer to Table 25
Function: UcanSetAcceptanceEx

Syntax:  
UCANRET PUBLIC UcanSetAcceptanceEx (tUcanHandle UcanHandle_p,  
BYTE bChannel_p,  
DWORD dwAMR_p, DWORD dwACR_p);

Usability: CAN_INIT

Description: Changes the acceptance filter registers of a specific CAN-channel of the USB-CANmodul for receiving CAN messages. This API function is an extended version of function UcanSetAcceptance().

Parameter:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UcanHandle_p</td>
<td>USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().</td>
</tr>
<tr>
<td>bChannel_p</td>
<td>CAN channel, which is to be used. USBCAN_CHANNEL_CH0 for CAN channel 0 USBCAN_CHANNEL_CH1 for CAN channel 1</td>
</tr>
<tr>
<td>dwAMR_p, dwACR_p</td>
<td>Acceptance Mask and Code Register to configure the hardware filter for receiving CAN messages (refer to section 4.3.5).</td>
</tr>
</tbody>
</table>

Return: Error code of the function – refer to Table 25.
### Function: UcanReadCanMsg

**Syntax:**
```c
UCANRET PUBLIC UcanReadCanMsg (tUcanHandle UcanHandle_p,
     tCanMsgStruct* pCanMsg_p);
```

**Usability:** CAN_INIT, only single-channel devices

**Description:** Reads a CAN message from the receive buffer. This function also reads back transmitted CAN messages as long as the CAN mode flag `kUcanModeTxEcho` was enabled at initialization time (refer to `UcanInitCanEx()` or `UcanInitCanEx2()`).

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **pCanMsg_p:** Pointer to the CAN message structure (see below). This pointer must not be NULL.

**Return:** Error code of the function

If the buffer contains no CAN messages, this function returns the warning USBCAN_WARN_NODATA. If a buffer overrun occurred, this function returns a valid CAN message and one of the warnings USBCAN_WARN_DLL_RXOVERRUN, USBCAN_WARN_SYS_RXOVERRUN or USBCAN_WARN_FW_RXOVERRUN.

Refer to Table 25 for detailed information.

```c
typedef struct
{
    DWORD m_dwID;
    BYTE m_bFF;
    BYTE m_bDLC;
    BYTE m_bData[8];
    DWORD m_dwTime;
} tCanMsgStruct;
```

**Parameter:**
- **m_dwID:** CAN identifier (CAN-ID)
- **m_bFF:** CAN frame format (refer to Table 21)
- **m_bDLC:** CAN data length code (DLC)
- **m_bData[8]:** CAN data (up to 8 bytes)
- **m_dwTime:** Time stamp of reception (or transmission for echo).
The CAN frame format is a bit mask that specifies the format of the CAN message. The following table lists all valid values:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_MSG_FF_STD</td>
<td>0x00</td>
<td>CAN2.0A message with 11-bit CAN-ID</td>
</tr>
<tr>
<td>USBCAN_MSG_FF_ECHO</td>
<td>0x20</td>
<td>Transmit echo; is only received if mode kUcanModeTxEcho was enabled at initialization time (refer to UcanInitCanEx() or UcanInitCanEx2()).</td>
</tr>
<tr>
<td>USBCAN_MSG_FF_RTR</td>
<td>0x40</td>
<td>CAN Remote Frame (all bytes are ignored)</td>
</tr>
<tr>
<td>USBCAN_MSG_FF_EXT</td>
<td>0x80</td>
<td>CAN2.0B message with 29-bit CAN-ID</td>
</tr>
</tbody>
</table>

**Note:**

In order to avoid receive buffer overflows it is recommended to call function UcanReadCanMsg() or UcanReadCanMsgEx() cyclically (e.g. in a loop) as long as a valid CAN message was received.

A valid CAN message was read, even if a warning was returned (except USBCAN_WARN_NODATA). You can use the macro USBCAN_CHECK_VALID_RXCANMSG() for checking whether a valid CAN message was stored to the CAN message structure (like shown in upper example).

**Example:**

```c
  tUcanHandle UcanHandle;
  tCabMsgStruct CanMsg;
  UCANRET bRet;

  ...  

  while (1) {
    // read CAN-message
    bRet = UcanReadCanMsg (UcanHandle, &CanMsg);

    // valid CAN message? print CAN-message
    if (USBCAN_CHECK_VALID_RXCANMSG (bRet)) {
      AppPrintCanMsg (&CanMsg);
      if (USBCAN_CHECK_WARNING (bRet)) {
        AppPrintWarning (bRet);
      }
    } // error occurred? print error
    else if (USBCAN_CHECK_ERROR (bRet)) {
      AppPrintError (bRet);
      break;
    } else {
      break;
    }
  } 

  ...  
```
Function: UcanReadCanMsgEx

Syntax:
```
UCANRET PUBLIC UcanReadCanMsgEx
(tUcanHandle UcanHandle_p,
 BYTE* pbChannel_p,
 tCanMsgStruct* pCanMsg_p,
 DWORD* pdwCount_p);
```

Usability: CAN_INIT

Description: Reads a CAN message from the receive buffer of a specific CAN-channel. This function is an extended version of function UcanReadCanMsg() and also reads back transmitted CAN messages as long as the CAN mode flag kUcanModeTxEcho was enabled at initialization time (refer to UcanInitCanEx() or UcanInitCanEx2()).

Parameter:

- **UcanHandle_p**: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **pbChannel_p**: Pointer to a variable of type BYTE.
  - Input: CAN channel, which is to be used.
    - USBCAN_CHANNEL_CH0 for CAN channel 0
    - USBCAN_CHANNEL_CH1 for CAN channel 1
    - USBCAN_CHANNEL_ANY for CAN channel 0 or 1
  - Output: CAN channel, the CAN message was read from
- **pCanMsg_p**: Pointer to the structure tCanMsgStruct. This pointer must not be NULL.
- **pdwCount_p**: Pointer to a variable of type DWORD
  - Input: Maximum number of CAN messages to be read.
  - Output: Number of CAN messages that were read from the receive buffer.
  - If this parameter is set to NULL, only one CAN message is read from the receive buffer.

Return: Error code of the function

- If the buffer contains no CAN messages, this function returns the warning USBCAN_WARN_NODATA. If a buffer overrun occurred, this function returns a valid CAN message and one of the warnings USBCAN_WARN_DLL_RXOVERRUN, USBCAN_WARN_SYS_RXOVERRUN or USBCAN_WARN_FW_RXOVERRUN.
- Refer to Table 25 for detailed information.

Note:
In order to avoid receive buffer overflows it is recommended to call function UcanReadCanMsg() or UcanReadCanMsgEx() cyclically (e.g. in a loop) as long as a valid CAN message was received.

A valid CAN message was read, even if a warning was returned (except USBCAN_WARN_NODATA). You can use the macro USBCAN_CHECK_VALID_RXCANMSG() for checking whether a valid CAN message was stored to the CAN message structure (like shown in upper example).
Example:

tUcanHandle UcanHandle;
tCabMsgStruct aRxCanMsg[16];
UCANRET bRet;
BYTE  bChannel;
DWORD dwCount;

while (1) {
    // read up to 16 CAN messages
    bChannel = USBCAN_CHANNEL_ANY;
    dwCount = sizeof(aRxCanMsg) / sizeof(aRxCanMsg[0]);
    bRet = UcanReadCanMsgEx (UcanHandle, &bChannel, &aRxCanMsg, &dwCount);

    // valid CAN message? print CAN messages
    if (USBCAN_CHECK_VALID_RXMSG (bRet)) {
        AppPrintCanMessages (&aRxCanMsg[0], dwCount);
        if (USBCAN_CHECK_WARNING (bRet))
            AppPrintWarning (bRet);
    }
    // error occurred? print error
    else if (USBCAN_CHECK_ERROR (bRet)) {
        AppPrintError (bRet);
        break;
    } else {
        break;
    }
}

Function: UcanWriteCanMsg

Syntax: 

UCANRET PUBLIC UcanWriteCanMsg (tUcanHandle UcanHandle_p, tCanMsgStruct* pCanMsg_p);

Usability: CAN_INIT, only single-channel devices

Description: Transmits a CAN message through the USB-CANmodul.

Parameter:

UcanHandle_p: USBCAN handle that was received with the function 
UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

pCanMsg_p: Pointer to the CAN message structure (refer to UcanReadCanMsg()). This pointer must not be NULL.

The meaning of CAN frame format is given in Table 21. For transmission of CAN messages, the bit USBCAN_MSG_FF_ECHO has no meaning.

For transmission of CAN messages, the parameter m_dwTime of structure tCanMsgStruct has no meaning.

Return: Error code of the function - refer to Table 25
Function: UcanWriteCanMsgEx

Syntax:

UCANRET PUBLIC UcanWriteCanMsgEx (tUcanHandle UcanHandle_p, BYTE bChannel_p, tCanMsgStruct* pCanMsg_p, DWORD* pdwCount_p);

Usability: CAN_INIT

Description: Transmits a CAN message through a specific CAN-channel of the USB-CANmodul.

Parameter:

UcanHandle_p: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

bChannel_p: CAN channel, which is to be used.
USBCAN_CHANNEL_CH0 for CAN channel 0
USBCAN_CHANNEL_CH1 for CAN channel 1

pCanMsg_p: Pointer to the CAN message structure (refer to UcanReadCanMsg()). This pointer must not be NULL.
The meaning of CAN frame format is given in Table 21. For transmission of CAN messages, the bit USBCAN_MSG_FF_ECHO has no meaning.
For transmission of CAN messages, the parameter m_dwTime of structure tCanMsgStruct has no meaning.

pdwCount_p: Pointer to a variable of type DWORD
Input: Number of CAN messages to be written to the transmitted buffer.
Output: Number of CAN messages that were successfully written to the transmit buffer.
If this parameter is set to NULL, only one CAN message is written to the transmit buffer.

Return: Error code of the function - refer to Table 25

Note:

If this function is called for transmitting more than one CAN messages, then the return code has also to be checked for the warning USBCAN_WARN_TXLIMIT. Receiving this return value only a part of the CAN messages was stored to the transmit buffer in USBCAN32.DLL. The variable which is referenced by the parameter pdwCount_p gets the number of successfully stored CAN messages. The part which was not stored to the transmit buffer has to be tried to be transmitted again by the application. Otherwise they will be lost.

You can use the macro USBCAN_CHECK_TX_NOTALL() for checking the return value whether some CAN messages could not be copied to the transmit buffer (see following example). The macro USBCAN_CHECK_TX_SUCCESS() checks whether all CAN messages could be stored to the transmit buffer while the macro USBCAN_CHECK_TX_OK() checks whether one CAN message at least was stored to the transmit buffer.
Example:

tUcanHandle UcanHandle;
tCabMsgStruct TxCanMsg[10];
UCANRET bRet;
DWORD dwCount;
...

// transmit up to 10 CAN messages

  // Check whether no error occurred
  if (USBCAN_CHECK_TX_OK (bRet))
    {
      // check whether a part of the array was not sent
      if (USBCAN_CHECK_TX_NOTALL (bRet))
        {
          // e.g. release the number of CAN messages from application
          AppReleaseTxMessages (dwCount);
          ...
        }
      else if (USBCAN_CHECK_WARNING (bRet))
        {
          AppPrintWarning (bRet);
        }
      // check whether an error occurred
    else if (USBCAN_CHECK_ERROR (bRet))
      {
        AppPrintError (bRet);
      }
    }
Function: UcanGetMsgPending

Syntax:  

UCANRET PUBLIC UcanGetMsgPending (tUcanHandle UcanHandle_p,  
BYTE  bChannel_p,  
DWORD dwFlags_p,  
DWORD* pdwCount_p);

Usability: CAN_INIT

Description: This function returns the number of the CAN messages which are currently stored to the buffers within the several software parts. The parameter dwFlags_p specifies which buffers should be checked. Should the function check more than one buffer, then the number of CAN messages will be added before writing to the variable which is referenced by the parameter pdwCount_p.

Parameter:

UcanHandle_p:  USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().

bChannel_p: CAN channel, which is to be used.

USBCAN_CHANNEL_CH0 for CAN channel 0
USBCAN_CHANNEL_CH1 for CAN channel 1
USBCAN_CHANNEL_ANY for both CAN channels

dwFlags_p: Specifies which buffers should be checked (refer to Table 22). The several flags can be combined using the OR-operation. In that case the number of CAN messages will be added.

pdwCount_p: Pointer to a variable of type DWORD receiving the number of pending CAN messages. This parameter must not be NULL.

Return: Error code of the function - refer to Table 25

Table 22: Constants for the flags parameter in function UcanGetMsgPending()

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_PENDING_FLAG_RX_DLL</td>
<td>0x00000001</td>
<td>Checks the number of messages of receive buffer within the DLL.</td>
</tr>
<tr>
<td>USBCAN_PENDING_FLAG_RX_FW</td>
<td>0x00000004</td>
<td>Checks the number of messages of receive buffer within module’s firmware.</td>
</tr>
<tr>
<td>USBCAN_PENDING_FLAG_TX_DLL</td>
<td>0x00000010</td>
<td>Checks the number of messages of transmit buffer within the DLL.</td>
</tr>
<tr>
<td>USBCAN_PENDING_FLAG_TX_FW</td>
<td>0x00000040</td>
<td>Checks the number of messages of transmit buffer within module’s firmware.</td>
</tr>
</tbody>
</table>

Note:

After function UcanGetMsgPending() returned to the application, the number of the CAN messages can already be changed within the several software parts. When the application calls this function too often, the performance can be decreased.
### Function: UcanGetCanErrorCounter

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>UCANRET PUBLIC UcanGetCanErrorCounter (tUcanHandle UcanHandle_p, BYTE bChannel_p, DWORD* pdwTxCount_p, DWORD* pdwRxCount_p);</th>
</tr>
</thead>
</table>

### Usability: CAN_INIT

### Description:
Returns the current error counters from CAN controller. These values are directly read from the hardware.

### Parameter:
- **UcanHandle_p**: USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **bChannel_p**: CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1
- **pdwTxCount_p, pdwRxCount_p**: Pointer to a variable of type DWORD to receive the current state of transmit or receive error counter. These parameters must not be NULL.

### Return:
Error code of the function - refer to [Table 25](#).
4.3.2.2 API Functions for automatic transmission

The following API functions are used to automatic transmission of cyclic CAN messages by the module’s firmware. This results a better cycle time as a Windows PC application could realize.

**Note:**
The accuracy of the cycle time also depends on the configured CAN baud rate. E.g. a jitter of approx. 10 milliseconds is a result of using a CAN baud rate of 10 kbps.

There is a maximum of 16 CAN messages which can be defined for the automatic transmission of cyclic CAN messages. Two modes are available for the automatic transmission. The first mode is called "parallel mode" the second one is called "sequential mode".

In parallel mode the cycle times of all defined CAN messages are checked within a process cycle. When a cycle time of a defined CAN message is over it will be sent to the CAN bus. The cycle time of a defined CAN message relates to the previous transmission of the same CAN message (refer to **Figure 35**).

In sequential mode the defined CAN messages are considered as a list of CAN messages which should be sent sequentially to the CAN bus. The cycle time of a defined CAN message relates to the transmission of the previously defined CAN message (refer to **Figure 36**). You can define a CAN message including the same CAN identifier but different data bytes more than once in sequential mode.

**Important:**
The transmission of CAN messages by calling the API function `UcanWriteCanMsg()` or `UcanWriteCanMsgEx()` can be influenced by the automatic transmission of cyclic CAN messages. When the CAN bus load is high (e.g. 50% or more) the CAN messages sent by the application are processed more rarely. The result could be that these API functions returns the error indicating a full transmit buffer.
**Function:** UcanDefineCyclicCanMsg  

**Syntax:**
```
UCANRET PUBLIC UcanDefineCyclicCanMsg (tUcanHandle UcanHandle_p,
BYTE bChannel_p,
tCanMsgStruct* pCanMsgList_p,
DWORD dwCount_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:** The function defines a set of up to 16 CAN messages within firmware of an USB-CANmodul for the automatic transmission of cyclic CAN messages. Call function UcanEnableCyclicCanMsg() for enabling the automatic transmission. Please note that UcanDefineCyclicCanMsg() completely exchanges a previously defined set of CAN messages.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **bChannel_p:** CAN channel, which is to be used.  
  - USBCAN_CHANNEL_CH0 for CAN channel 0  
  - USBCAN_CHANNEL_CH1 for CAN channel 1
- **pCanMsgList_p:** Pointer to an array of type tCanMsgStruct containing a set of CAN messages for automatic transmission. The member m_dwTime of the structure tCanMsgStruct specifies the cycle time. This parameter may only be NULL when dwCount_p is zero too. In this case a previously defined set of CAN messages will be deleted.
- **dwCount_p:** Specifies the number of CAN messages included within the array. The value range is 0 to 16. A previously defined set of CAN messages will be deleted by specifying the number of 0 CAN messages.

**Return:** Error code of the function - refer to Table 25

**Example:**

```
tUcanHandle UcanHandle;
tCabMsgStruct aTxCanMsg[] =
{  
  {0x080, USBCAN_MSG_FF_STD, 0, {0,0,0,0, 0,0,0,0}, 100}, // message 1
  {0x100, USBCAN_MSG_FF_STD, 4, {1,2,3,4, 0,0,0,0}, 150}}; // message 2
UCANRET bRet;
DWORD dwCount;
...
// define 2 CAN messages for automatic transmission by the USB-CANmodul
dwCount = sizeof (aTxCanMsg) / sizeof (aTxCanMsg);
bRet = UcanDefineCyclicCanMsg (UcanHandle, USBCAN_CHANNEL_CH0,
aTxCanMsg[0], dwCount);
if (bRet == USBCAN_SUCCESSFUL)
{
    // start the transmission
    bRet = UcanEnableCyclicCanMsg(UcanHandle, USBCAN_CHANNEL_CH0,
    USBCAN_CYCLIC_FLAG_START | USBCAN_CYCLIC_FLAG_NOECHO);
    if (bRet == USBCAN_SUCCESSFUL)
    {
        ...
    }
}
...```
## Function: UcanReadCyclicCanMsg

### Syntax:

```
UCANRET PUBLIC UcanReadCyclicCanMsg (tUcanHandle UcanHandle_p,
    BYTE bChannel_p,
    tCanMsgStruct* pCanMsgList_p,
    DWORD* pdwCount_p);
```

### Usability:

HW_INIT, CAN_INIT

### Description:

The function reads back the set of CAN messages which was previously defined for automatic transmission of cyclic CAN messages (refer to function UcanDefineCyclicCanMsg()).

### Parameter:

- **UcanHandle_p**: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **bChannel_p**: CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1
- **pCanMsgList_p**: Pointer to an array of type tCanMsgStruct receiving the set of up to 16 CAN messages for automatic transmission. This parameter must not be NULL.
- **pdwCount_p**: Pointer to a variable of type DWORD for receiving the number of defined CAN messages within the set.

### Return:

Error code of the function - refer to Table 25

### Example:

```c
// read the CAN messages for automatic transmission by the USB-CANmodul
UCANRET bRet;
DWORD dwCount, i;
...

bRet = UcanReadCyclicCanMsg(UcanHandle, USBCAN_CHANNEL_CH0, &aTxCanMsg[0], &pdwCount);
if (bRet == USBCAN_SUCCESSFUL)
{
    // print all CAN messages
    for (i = 0; i < dwCount; i++)
    {
        AppPrintMsg (&aTxCanMsg[i]);
    }
}
...```
**Function:** UcanEnableCyclicCanMsg

**Syntax:**
```c
UCANRET PUBLIC UcanEnableCyclicCanMsg (tUcanHandle UcanHandle_p,
       BYTE bChannel_p,
       DWORD dwFlags_p);
```

**Usability:** CAN_INIT

**Description:** This function specifies the mode of the automatic transmission and specifies whether the automatic transmission of a set of defined CAN messages should be enabled or disabled. Additionally separate CAN messages of the set can be locked or unlocked.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **bChannel_p:** CAN channel, which is to be used. USBCAN_CHANNEL_CH0 for CAN channel 0 USBCAN_CHANNEL_CH1 for CAN channel 1
- **dwFlags_p:** Specifies flags containing the mode, the enable state and the locking state (refer to Table 23). These flags can be combined.

**Return:** Error code of the function - refer to Table 25

**Table 23:** Constants for the flags parameter in function UcanEnableCyclicCanMsg()

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_CYCLIC_FLAG_START</td>
<td>0x80000000</td>
<td>If this flag is set, the automatic transmission will be started, otherwise it will be stopped.</td>
</tr>
<tr>
<td>USBCAN_CYCLIC_FLAG_SEQUMODE</td>
<td>0x40000000</td>
<td>If this flag is set, the “sequential mode” is processed, otherwise the “parallel mode” is processed (refer to Figure 35 and Figure 36).</td>
</tr>
<tr>
<td>USBCAN_CYCLIC_FLAG_NOECHO</td>
<td>0x00010000</td>
<td>If this flag is set, the sent cyclic CAN messages are not received back using transmit echo.</td>
</tr>
<tr>
<td>USBCAN_CYCLIC_FLAG_LOCK_0</td>
<td>0x00000001</td>
<td>If same of these flags are set, the appropriate CAN message from the set is not sent to the CAN bus (locked state).</td>
</tr>
<tr>
<td>USBCAN_CYCLIC_FLAG_LOCK_15</td>
<td>0x00008000</td>
<td></td>
</tr>
</tbody>
</table>

**Example:** Refer to the example on page 97.

### 4.3.2.3 API Functions for the CAN port

The following API functions can only be used with the USB-CANmodul2. They are an expansion for using the USB-CANmodul with a low-speed or single-wire CAN transceiver (e.g. TJA1054 or AU5790). Using these API functions with the USB-CANmodul2 with a high-speed CAN transceiver or USB-CANmodul1 has no effect. However no error message will be returned either. In order to use these API functions, the header file USBCANLS.H must be included in addition to the USBCAN32.H header file.
**Function:** UcanWriteCanPort

**Syntax:**
UCANRET PUBLIC UcanWriteCanPort (tUcanHandle UcanHandle_p, BYTE bValue_p);

**Usability:** HW_INIT, CAN_INIT, only single-channel devices

**Description:** Writes a value to the CAN port interface. Thus additional signals such as Standby (STB) and Enable (EN) on a low-speed or single-wire CAN transceiver can be controlled.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **bValue_p:** New output value for the CAN port interface (refer to Table 24 and Table 6).

**Return:** Error code of the function - refer to Table 25

**Note:**
With the call to this API function the output value is additionally stored to the non-volatile memory of the USB-CANmodul. The last stored output value is restored to the CAN port after power-on on the USB-CANmodul.

**Table 24:** Constants for low-speed CAN port

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCAN_CANPORT_TR</td>
<td>0x10</td>
<td>Input</td>
<td>Termination resistor</td>
</tr>
<tr>
<td>UCAN_CANPORT_ERR</td>
<td>0x20</td>
<td>Input</td>
<td>Error signal of low-speed CAN transceiver</td>
</tr>
<tr>
<td>UCAN_CANPORT_STB</td>
<td>0x40</td>
<td>Output</td>
<td>Stand-by (STB) signal of low-speed CAN transceiver</td>
</tr>
<tr>
<td>UCAN_CANPORT_EN</td>
<td>0x80</td>
<td>Output</td>
<td>Enable signal (EN) of low-speed CAN transceiver</td>
</tr>
</tbody>
</table>
### Function: UcanWriteCanPortEx

**Syntax:**
```
UCANRET PUBLIC UcanWriteCanPortEx (tUcanHandle UcanHandle_p,
    BYTE bChannel_p,
    BYTE bValue_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:**
Writes a value to the CAN port interface of a specific CAN-channel. This function is an extended version of function `UcanWriteCanPort()`.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **bChannel_p:** CAN channel, which is to be used. USBCAN_CHANNEL_CH0 for CAN channel 0, USBCAN_CHANNEL_CH1 for CAN channel 1
- **bValue_p:** New output value for the CAN port interface (refer to Table 24 and Table 6).

**Return:** Error code of the function - refer to Table 25

### Function: UcanReadCanPort

**Syntax:**
```
UCANRET PUBLIC UcanReadCanPort (tUcanHandle UcanHandle_p,
    BYTE* pbValue_p);
```

**Usability:** HW_INIT, CAN_INIT, only single-channel devices

**Description:**
Reads the current input value from the CAN port interface. Thus the additional error signal (ERR) can be read on a low-speed CAN transceiver. It is also possible to read the state/constant for the terminating resistor on devices with high-speed transceivers (currently only supported for USB-CANmodul2 – refer to section 2.3).

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **pbValue_p:** Pointer to a variable that receives the read input value (refer to Table 24 and Table 6). This parameter must not be NULL.

**Return:** Error code of the function - refer to Table 25
Function: UcanReadCanPortEx

Syntax: 
```c
UCANRET PUBLIC UcanReadCanPortEx (tUcanHandle UcanHandle_p,
    BYTE   bChannel_p,
    BYTE*  pbInValue_p,
    BYTE*  pbLastOutValue_p);
```

Usability: HW_INIT, CAN_INIT

Description: Reads the current input value from the specified CAN-channel. This function is an extended version of function UcanReadCanPort().

Parameter:
- **UcanHandle_p**: USBCAN handle that was received with the function UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2() as well as UcanEnumerateHardware().
- **bChannel_p**: CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1
- **pbInValue_p**: Pointer to a variable that receives the read input value (refer to Table 24 and Table 6). This parameter must not be NULL.
- **pbLastOutValue_p**: Pointer to a variable that receives the last written output value (using UcanWriteCanPort() or UcanWriteCanPortEx() - refer to Table 24 and Table 6). This parameter may be NULL.

Return: Error code of the function - refer to Table 25
4.3.2.4 API Functions for the expansion port

The following API functions can only be used with the USB-CANmodul2. They are an expansion for the use of the USB-CANmodul with the expansion port. Using these API functions with other variants of USB-CANmodul devices has no effect. In order to use these API functions, the file USBCANUP.H must be included in addition to the USBCAN32.H header file.

**Function:** UcanConfigUserPort  
**Syntax:** UCANRET PUBLIC UcanConfigUserPort (tUcanHandle UcanHandle_p, BYTE bOutputEnable_p);

**Usability:** HW_INIT, CAN_INIT

**Description:** Configures the expansion port (refer to section 2.5). Each individual pin of the 8-bit port can be used as an input or an output. The logical value 0 of a bit in the parameter bOutputEnable_p defines the corresponding pin on the expansion port an input and a logical 1 defines it as an output.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function _UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2_ as well as _UcanEnumerateHardware_.
- **bOutputEnable_p:** Configuring the 8-bit port as input or output: 
  - Bit X = 0: Pin X = input
  - Bit Y = 1: Pin Y = output

**Return:** Error code of the function - refer to _Table 25_

**Note:**
With the call to this API function the configuration value is additionally stored to the non-volatile memory of the USB-CANmodul. The last stored configuration is restored after power-on on the USB-CANmodul.

**Function:** UcanWriteUserPort  
**Syntax:** UCANRET PUBLIC UcanWriteUserPort (tUcanHandle UcanHandle_p, BYTE bOutputValue_p);

**Usability:** HW_INIT, CAN_INIT

**Description:** Writes a value to the expansion port. In order to write to output lines, the corresponding bits resp. port pins must be configured as outputs using the _UcanConfigUserPort()_ function.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function _UcanInitHardware(), UcanInitHardwareEx() or UcanInitHardwareEx2_ as well as _UcanEnumerateHardware_.
- **bOutputValue_p:** New output value for the expansion port outputs. Each bit in this parameter corresponds to matching pin on the expansion port.

**Return:** Error code of the function - refer to _Table 25_
Note:
No time critical switching procedures can be performed with this function using the expansion port, since the reaction time is influenced by multiple factors.

With the call to this API function the output value is additionally stored to the non-volatile memory of the USB-CANmodul. The last stored configuration is restored after power-on on the USB-CANmodul.

**Function:** UcanReadUserPort

**Syntax:**

```c
UCANRET PUBLIC UcanReadUserPort (tUcanHandle UcanHandle_p,
       BYTE* pbInputValue_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:**
Reads the current input value from the expansion port. This function can also be used to read back the states of ports configured as outputs.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.  
- **pbInputValue_p:** Pointer to a variable that receives the read input value. This variable then contains the state of the 8-bit expansion port. Each bit in this parameter corresponds to matching pin on the expansion port. This parameter must not be NULL.

**Return:** Error code of the function - refer to Table 25

**Function:** UcanReadUserPortEx

**Syntax:**

```c
UCANRET PUBLIC UcanReadUserPortEx (tUcanHandle UcanHandle_p,
       BYTE* pbInputValue_p,
       BYTE* pbLastOutputEnable_p,
       BYTE* pbLastOutputValue_p);
```

**Usability:** HW_INIT, CAN_INIT

**Description:**
Reads the current input value from the expansion port. This function is an extended version of function `UcanReadUserPort()`.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`, `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.  
- **pbInputValue_p:** Pointer to a variable that receives the read input value. This variable then contains the state of the 8-bit expansion port. Each bit in this parameter corresponds to matching pin on the expansion port. This parameter must not be NULL.
- **pbLastOutputEnable_p:** Pointer to a variable that receives the output configuration (configuration that was previously done with `UcanConfigUserPort()`). This parameter may be NULL.
- **pbLastOutputValue_p:** Pointing to a variable that receives the last output value (value that was written with `UcanWriteUserPort()`). This parameter may be NULL.

**Return:** Error code of the function - refer to Table 25
4.3.3 Error codes of the API functions

The API functions of the DLL return an error code with the type of UCANRET. Each return value represents an error. The only two exception are the functions UcanReadCanMsg() and UcanReadCanMsgEx() which can also return warnings. The warning USBCAN_WARN_NODATA indicates that no CAN messages are in the buffer. Other warnings indicate that an event has occurred but a valid CAN message is transferred.

Table 25: Error codes of the API functions

<table>
<thead>
<tr>
<th>Error code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_SUCCESSFUL</td>
<td>0x00</td>
<td>This value returns if the function is executed successfully.</td>
</tr>
<tr>
<td>USBCAN_ERR_RESOURCE</td>
<td>0x01</td>
<td>This error code returns if one resource could not be generated. In this case the term resource means memory and handles provided by the Windows OS.</td>
</tr>
<tr>
<td>USBCAN_ERR_MAXMODULES</td>
<td>0x02</td>
<td>An application has tried to open more than 64 USB-CANmodul devices. The standard version of the DLL only supports up to 64 USB-CANmodul devices at the same time. This error also appears if several applications try to access more than 64 USB-CANmodul devices. For example, application 1 has opened 60 modules, application 2 has opened 4 modules and application 3 wants to open a module. Application 3 receives this error code.</td>
</tr>
<tr>
<td>USBCAN_ERR_HWINUSE</td>
<td>0x03</td>
<td>An application tries to initialize an USB-CANmodul with the given device number. If this module has already been initialized by its own or by another application, this error code is returned.</td>
</tr>
<tr>
<td>USBCAN_ERR_ILLVERSION</td>
<td>0x04</td>
<td>This error code returns if the firmware version of the USB-CANmodul is not compatible to the software version of the DLL. In this case, install the latest driver for the USB-CANmodul. Furthermore make sure that the latest firmware version is programmed to the USB-CANmodul.</td>
</tr>
<tr>
<td>USBCAN_ERR_ILLHW</td>
<td>0x05</td>
<td>This error code returns if an USB-CANmodul with the given device number is not found. If the function UcanInitHardware() or UcanInitHardwareEx() has been called with the device number USBCAN_ANY_MODULE, and the error code appears, it indicates that no module is connected to the PC or all connected modules are already in use.</td>
</tr>
<tr>
<td>USBCAN_ERR_ILLHANDLE</td>
<td>0x06</td>
<td>This error code returns if a function received an incorrect USBCAN handle. The function first checks which USB-CANmodul is related to this handle. This error occurs if no device belongs this handle.</td>
</tr>
<tr>
<td>USBCAN_ERR_ILLPARAM</td>
<td>0x07</td>
<td>This error code returns if a wrong parameter is passed to the function. For example, the value NULL has been passed to a pointer variable instead of a valid address.</td>
</tr>
<tr>
<td>USBCAN_ERR_BUSY</td>
<td>0x08</td>
<td>This error code occurs if several threads are accessing an USB-CANmodul within a single application. After the other threads have finished their tasks, the function may be called again.</td>
</tr>
<tr>
<td>USBCAN_ERR_TIMEOUT</td>
<td>0x09</td>
<td>This error code occurs if the function transmits a command to the USB-CANmodul but no reply is returned. To solve this problem, close the application, disconnect the USB-CANmodul, and connect it again.</td>
</tr>
<tr>
<td>USBCAN_ERR_IOFAILED</td>
<td>0x0A</td>
<td>This error code occurs if the communication to the kernel driver was interrupted. This happens, for example, if the USB-CANmodul is disconnected during transferring data or commands to the USB-CANmodul.</td>
</tr>
<tr>
<td>USBCAN_ERR_DLL_TXFULL</td>
<td>0x0B</td>
<td>The function UcanWriteCanMsg() or UcanWriteCanMsgEx() first checks if the transmit buffer within the DLL has enough capacity to store new CAN messages. If the buffer is full, this error code returns. The CAN message passed to these functions will not be written into the transmit buffer in order to protect other CAN messages against overwriting. The size of the transmit buffer is configurable (refer to function UcanInitCanEx() and structure UcanInitCanParam).</td>
</tr>
<tr>
<td>USBCAN_ERR_MAXINSTANCES</td>
<td>0x0C</td>
<td>A maximum amount of 64 applications are able to have access to the DLL. If more applications attempting to access to the DLL, this error code is returned. In this case, it is not possible to use an USB-CANmodul by this application.</td>
</tr>
<tr>
<td>Error code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>USBCAN_ERR_CANNOTINIT</td>
<td>0x0D</td>
<td>This error code returns if an application tries to call an API function which only can be called in software state CAN_INIT but the current software is still in state HW_INIT. Refer to section 4.3.1 and Table 12 for detailed information.</td>
</tr>
<tr>
<td>USBCAN_ERR_DISCONNECT</td>
<td>0x0E</td>
<td>This error code occurs if an API function was called for an USB-CANmodul that was plugged-off from the computer recently.</td>
</tr>
<tr>
<td>USBCAN_ERR_ILLCHANNEL</td>
<td>0x10</td>
<td>This error code is returned if an extended function of the DLL is called with parameter bChannel_p = USBCAN_CHANNEL_CH1, but a single-channel USB-CANmodul was used.</td>
</tr>
<tr>
<td>USBCAN_ERR_ILLHWTYPE</td>
<td>0x12</td>
<td>This error code occurs if an extended function of the DLL was called for a hardware which does not support the feature.</td>
</tr>
<tr>
<td>USBCAN_ERRRCMD_NOTEQU</td>
<td>0x40</td>
<td>The software tests the CAN controller on the USB-CANmodul when the CAN interface is initialized. Several registers of the CAN controller are checked. This error code returns if an error appears during this register test.</td>
</tr>
<tr>
<td>USBCAN_ERRRCMD_REGTST</td>
<td>0x41</td>
<td>This error code returns if the USB-CANmodul receives a non-defined command. This error represents a version conflict between the firmware in the USB-CANmodul and the DLL.</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_ILLCMD</td>
<td>0x42</td>
<td>The USB-CANmodul has a built-in EEPROM. This EEPROM contains several configurations, e.g. the device number and the serial number. If an error occurs while reading these values, this error code is returned.</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_EEPROM</td>
<td>0x43</td>
<td>The USB-CANmodul has been initialized with an invalid baud rate (refer to section 4.3.4).</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_ILLBDR</td>
<td>0x47</td>
<td>The USB-CANmodul was already initialized.</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_NOTINIT</td>
<td>0x48</td>
<td>It was tried to access a CAN-channel of a multi-channel USB-CANmodul that was not initialized.</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_ALREADYINIT</td>
<td>0x49</td>
<td>The accessed CAN-channel of a multi-channel USB-CANmodul was already initialized.</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_ILLSUBCMD</td>
<td>0x4A</td>
<td>An internal error occurred within the DLL. In this case an unknown subcommand was called instead of a main command (e.g. for the cyclic CAN message-feature).</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_ILLIDX</td>
<td>0x4B</td>
<td>An internal error occurred within the DLL. In this case an invalid index for a list was delivered to the firmware (e.g. for the cyclic CAN message-feature).</td>
</tr>
<tr>
<td>USBCAN_ERRCMD_RUNNING</td>
<td>0x4C</td>
<td>The caller tries to define a new list of cyclic CAN messages but this feature was already started. For defining a new list, it is necessary to stop the feature beforehand.</td>
</tr>
<tr>
<td>USBCAN_WARN_NODATA</td>
<td>0x80</td>
<td>If the function UcanReadCanMsg() or UcanReadCanMsgEx() returns with this warning, it is an indication that the receive buffer contains no CAN messages.</td>
</tr>
<tr>
<td>USBCAN_WARN_SYS_RXOVERRUN</td>
<td>0x81</td>
<td>This is returned by UcanReadCanMsg() or UcanReadCanMsgEx() if the receive buffer within the kernel driver runs over. The function nevertheless returns a valid CAN message. It also indicates that at least one CAN message are lost. However, it does not indicate the position of the lost CAN messages.</td>
</tr>
<tr>
<td>USBCAN_WARN_DLL_RXOVERRUN</td>
<td>0x82</td>
<td>The DLL automatically requests CAN messages from the USB-CANmodul and stores the messages into a buffer of the DLL. If more CAN messages are received than the DLL buffer size allows, this error code returns and CAN messages are lost. However, it does not indicate the position of the lost CAN messages. The size of the receive buffer is configurable (refer to function UcanInitCanEx() and structure tUcanInitCanParam).</td>
</tr>
</tbody>
</table>
### Error Codes for USBCAN

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_WARN_FW_TXOVERRUN</td>
<td>0x85</td>
<td>This warning is returned by function <code>UcanWriteCanMsg()</code> or <code>UcanWriteCanMsgEx()</code> if flag USBCAN_CANERR_QXMTFULL is set in the CAN driver status. However, the transmit CAN message could be stored to the DLL transmit buffer. This warning indicates that at least one transmit CAN message got lost in the device firmware layer. This warning does not indicate the position of the lost CAN message.</td>
</tr>
<tr>
<td>USBCAN_WARN_FW_RXOVERRUN</td>
<td>0x86</td>
<td>This warning is returned by function <code>UcanWriteCanMsg()</code> or <code>UcanWriteCanMsgEx()</code> if flag USBCAN_CANERR_QOVERRUN or flag USBCAN_CANERR_OVERRUN are set in the CAN driver status. The function has returned with a valid CAN message. This warning indicates that at least one received CAN message got lost in the firmware layer. This warning does not indicate the position of the lost CAN message.</td>
</tr>
<tr>
<td>USBCAN_WARN_NULL_PTR</td>
<td>0x90</td>
<td>This warning is returned by functions <code>UcanInitHwConnectControl()</code> or <code>UcanInitHwConnectControlEx()</code> if a NULL pointer was passed as callback function address.</td>
</tr>
<tr>
<td>USBCAN_WARN_TXLIMIT</td>
<td>0x91</td>
<td>This warning is returned by the function <code>UcanWriteCanMsgEx()</code> if it was called to transmit more than one CAN message, but a part of them could not be stored to the transmit buffer within the DLL (because the buffer is full). The returned variable addressed by the parameter <code>pdwCount_p</code> indicates the number of CAN messages which are stored successfully to the transmit buffer.</td>
</tr>
</tbody>
</table>

Use the following macros are for checking the return value of several functions:

**Macro:** USBCAN_CHECK_VALID_RXCANMSG

**Syntax:**

```
USBCAN_CHECK_VALID_RXCANMSG(bRet_p)
```

**Description:**
This Macro checks whether the function `UcanReadCanMsg()` or `UcanReadCanMsgEx()` returns a valid CAN message.

**Parameter:**
- `bRet_p`: Return value of type UCANRET as defined in Table 25.

**Return:**
- FALSE (zero) or TRUE (non-zero)

**Macro:** USBCAN_CHECK_TX_OK

**Syntax:**

```
USBCAN_CHECK_TX_OK(bRet_p)
```

**Description:**
This Macro checks whether the function `UcanWriteCanMsg()` or `UcanWriteCanMsgEx()` successfully wrote CAN message(s) to the transmit buffer. While using `UcanWriteCanMsgEx()` the number of written CAN messages may be less than the number of CAN messages passed to this function (refer to error code USBCAN_WARN_TXLIMIT).

**Parameter:**
- `bRet_p`: Return value of type UCANRET as defined in Table 25.

**Return:**
- FALSE (zero) or TRUE (non-zero)
### Macro: **USBCAN_CHECK_TX_SUCCESS**

**Syntax:**

```c
USBCAN_CHECK_TX_SUCCESS(bRet_p)
```

**Description:**

This Macro checks whether the function `UcanWriteCanMsg()` or `UcanWriteCanMsgEx()` successfully wrote all CAN message(s) to the transmit buffer.

**Parameter:**

- `bRet_p`: Return value of type UCANRET as defined in Table 25.

**Return:**

FALSE (zero) or TRUE (non-zero)

### Macro: **USBCAN_CHECK_TX_NOTALL**

**Syntax:**

```c
USBCAN_CHECK_TX_NOTALL(bRet_p)
```

**Description:**

This Macro checks whether the function `UcanWriteCanMsgEx()` could not write at least one CAN message to the transmit buffer.

**Parameter:**

- `bRet_p`: Return value of type UCANRET as defined in Table 25.

**Return:**

FALSE (zero) or TRUE (non-zero)

### Macro: **USBCAN_CHECK_WARNING**

**Syntax:**

```c
USBCAN_CHECK_WARNING(bRet_p)
```

**Description:**

This Macro checks whether any function returned a warning.

**Parameter:**

- `bRet_p`: Return value of type UCANRET as defined in Table 25.

**Return:**

FALSE (zero) or TRUE (non-zero)

### Macro: **USBCAN_CHECK_ERROR**

**Syntax:**

```c
USBCAN_CHECK_ERROR(bRet_p)
```

**Description:**

This Macro checks whether any function returned an error.

**Parameter:**

- `bRet_p`: Return value of type UCANRET as defined in Table 25.

**Return:**

FALSE (zero) or TRUE (non-zero)
### Macro: USBCAN_CHECK_ERROR_CMD

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>USBCAN_CHECK_ERROR(bRet_p)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>This Macro checks whether any function returned an error which occurred in firmware of the USB-CANmodul.</td>
</tr>
<tr>
<td><strong>Parameter:</strong></td>
<td></td>
</tr>
<tr>
<td>bRet_p:</td>
<td>Return value of type UCANRET as defined in <a href="#">Table 25</a>.</td>
</tr>
<tr>
<td><strong>Return:</strong></td>
<td>FALSE (zero) or TRUE (non-zero)</td>
</tr>
</tbody>
</table>
4.3.4 Baud Rate Configuration

Sections 4.3.4.1 and 4.3.4.2 describes the baud rate configuration of obsolete types of USB-CANmodul devices. The baud rate configuration of these devices are described here only for compatibility reason. In section 4.3.4.3 the baud rate configuration of the USB-CANmodul devices of fourth generation is described. Only these devices are documented within the scope of this manual.

4.3.4.1 Baud Rate Configuration for first and second generation USB-CANmodul

The baud rate configuration for obsolete USB-CANmodul devices of first and second generation is passed to the API function `UcanInitCan()` as parameter `bBTR0_p` and `bBTR1_p`. Using the API function `UcanInitCanEx()` and/or `UcanInitCanEx2()` this configuration is passed to the parameters `m_bBTR0` and `m_bBTR1` located in structure `tUcanInitCanParam`. The configuration may also be changed later by calling the function `UcanSetBaudrate()` or `UcanSetBaudrateEx()`.

The following values are recommended if obsolete USB-CANmodul devices of first and/or second generation shall be supported by your application too:

**Table 26: Constants for CAN baud rates for first and second generation**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Sample-point</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USBCAN_BAUD_10kBit</strong></td>
<td>0x672F</td>
<td>CAN baud rate 10 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_20kBit</strong></td>
<td>0x532F</td>
<td>CAN baud rate 20 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_50kBit</strong></td>
<td>0x472F</td>
<td>CAN baud rate 50 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_100kBit</strong></td>
<td>0x432F</td>
<td>CAN baud rate 100 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_125kBit</strong></td>
<td>0x031C</td>
<td>CAN baud rate 125 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_250kBit</strong></td>
<td>0x011C</td>
<td>CAN baud rate 250 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_500kBit</strong></td>
<td>0x001C</td>
<td>CAN baud rate 500 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_800kBit</strong></td>
<td>0x0016</td>
<td>CAN baud rate 800 kbps</td>
<td>80.00%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_1MBit</strong></td>
<td>0x0014</td>
<td>CAN baud rate 1000 kbps</td>
<td>75.00%</td>
</tr>
<tr>
<td><strong>USBCAN_BAUD_USE_BTREX</strong></td>
<td>0x0000</td>
<td>Parameter <code>dwBaudrate</code> is used – refer to Table 27, Table 28 or Table 29</td>
<td></td>
</tr>
</tbody>
</table>

Use the macros `HIGBYTE()` to pass the constant defined in Table 26 to the parameter BTR0 and use the macro `LOBYTE()` to pass the constant to the parameter BTR1.
Example:

```c
//tUcanHandle UcanHandle;
UCANRET bRet;
...

// initializes the hardware
bRet = UcanInitHardware (&UcanHandle, 0, NULL);
...

// initializes the CAN interface
bRet = UcanInitCan (UcanHandle,
HIBYTE (USBCAN_BAUD_1MBit), // BTR0 for 1MBit/s
LOBYTE (USBCAN_BAUD_1MBit), // BTR1 for 1MBit/s
0xFFFFFFFF, // AMR for all messages received
0x00000000); // ACR for all messages received

// Error? print error
if (bRet != USBCAN_SUCCESSFUL)
    PrintError (bRet);
...
```

Configuration of user-defined baud rates is also possible. The structure of the BTR0 and BTR1 registers is described below. Refer to the NXP SJA1000 manual for detailed description.

**Figure 37:** Structure of baud rate register BTR0

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SJW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPR[5:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 38:** Structure of baud rate register BTR1

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAM</td>
<td></td>
<td></td>
<td></td>
<td>TSEG2[2:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSEG1[3:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Parameter:**

- **BPR:** *Baudrate Prescaler* specifies the ratio between system clock of the SJA1000 and the bus clock on the CAN-bus.

- **SJW:** *Synchronization Jump Width* specifies the compensation of the phase-shift between the system clock and the different CAN-controllers connected to the CAN-bus.

- **SAM:** *Sampling* specifies the number of sample points used for reading the bits on the CAN-bus. If SAM=1 three sample points are used, otherwise only one sample point is used.

- **TSEG1, TSEG2:** *Time Segment* specifies the number of clock cycles of one bit on the CAN-bus as well as the position of the sample points.
System Manual USB-CANmodul

The following mathematical correlations apply:

\[
\begin{align*}
  t_{CLK} &= \frac{1}{16 \text{MHz}} = 62.5\text{ns} \quad \text{(system clock)} \\
  t_{SCL} &= 2 \cdot t_{CLK} \cdot (BPR + 1) \quad \text{(bus clock)} \\
  t_{SYNCSEG} &= 1 \cdot t_{SCL} \\
  t_{TSEG1} &= t_{SCL} \cdot (TSEG1 + 1) \\
  t_{TSEG2} &= t_{SCL} \cdot (TSEG2 + 1) \\
  t_{Bit} &= t_{SYNCSEG} + t_{TSEG1} + t_{TSEG2} \quad \text{(time of one bit on the CAN bus)} \\
  p_{Sample} &= \frac{t_{SYNCSEG} + t_{TSEG1} + t_{TSEG2}}{t_{SYNCSEG} + t_{TSEG1} + t_{TSEG2}} \cdot 100\% \quad \text{(sample-point)}
\end{align*}
\]

Example for 125 kbps (TSEG1 = 12, TSEG2 = 1, BPR = 3):

\[
\begin{align*}
  t_{SCL} &= 2 \cdot 62.5\text{ns} \cdot (3 + 1) = 500\text{ns} \\
  t_{SYNCSEG} &= 1 \cdot 500\text{ns} = 500\text{ns} \\
  t_{TSEG1} &= 500\text{ns} \cdot (12 + 1) = 6500\text{ns} \\
  t_{TSEG2} &= 500\text{ns} \cdot (1 + 1) = 1000\text{ns} \\
  t_{Bit} &= 500\text{ns} + 6500\text{ns} + 1000\text{ns} = 8000\text{ns} \\
  \frac{1}{t_{Bit}} &= \frac{1}{8000\text{ns}} = 125\text{kbps} \\
  p_{Sample} &= \frac{500\text{ns} + 6500\text{ns} + 1000\text{ns}}{500\text{ns} + 6500\text{ns} + 1000\text{ns}} \cdot 100\% = 87.5\%
\end{align*}
\]

4.3.4.2 Baud Rate Configuration for third generation USB-CANmodul

The baud rate configuration for obsolete USB-CANmodul devices of third generation is passed to the API function `UcanInitCanEx()` and/or `UcanInitCanEx2()` using the parameters `m_dwBaudrate` located in structure `tUcanInitCanParam`. The configuration may also be changed later by calling the function `UcanSetBaudrateEx()`.

Figure 39: General structure of one bit on the CAN-bus (source: NXP SJA1000 manual)
The following values are recommended if obsolete USB-CANmodul devices of third generation shall be supported by your application:

Table 27: Constants for CAN baud rates for third generation

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Sample-point</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_BADEX_SP2_10kBit</td>
<td>0x80771772</td>
<td>CAN baud rate 10 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_20kBit</td>
<td>0x00771772</td>
<td>CAN baud rate 20 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_50kBit</td>
<td>0x003B1741</td>
<td>CAN baud rate 50 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_100kBit</td>
<td>0x001D1741</td>
<td>CAN baud rate 100 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_125kBit</td>
<td>0x00170741</td>
<td>CAN baud rate 125 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_250kBit</td>
<td>0x000B0741</td>
<td>CAN baud rate 250 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_500kBit</td>
<td>0x00050741</td>
<td>CAN baud rate 500 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_800kBit</td>
<td>0x00030731</td>
<td>CAN baud rate 800 kbps</td>
<td>86.67%</td>
</tr>
<tr>
<td>USBCAN_BADEX_SP2_1MBit</td>
<td>0x00020741</td>
<td>CAN baud rate 1000 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BADEX_USE_BTR01</td>
<td>0x00000000</td>
<td>Parameters BTR0/BTR1 are used – refer to Table 26</td>
<td></td>
</tr>
</tbody>
</table>

Configuration of user-defined baud rates is possible. The register structure for extended baud rate configuration is described below.

![Structure of baud rate register dwBaudrate of third generation modules](image)

Figure 40: Structure of baud rate register dwBaudrate of third generation modules
**Parameter:**

**CLK:**  
*Clock* specifies the frequency of the microcontroller. If set to 0, then the microcontroller runs with 48 MHz clock cycle internally, otherwise it runs with 24 MHz. This influences the CAN-bus baud rate (refer to system clock \( t_{MCK} \) in the following example).

**SMP**  
*Sampling* specifies the number of sample points used for reading the bits on the CAN-bus. If \( SAM=1 \) three sample points are used, otherwise only one sample point is used.

**BRP:**  
*Baudrate Prescaler* specifies the ratio between system clock of the microcontroller and the bus clock on the CAN-bus.

**SJW:**  
*Synchronization Jump Width* specifies the compensation of the phase-shift between the system clock and the different CAN-controllers connected to the CAN-bus.

**PHASE_SEG1, PHASE_SEG2:**  
*Time Segment* specifies the number of clock cycles of one bit on the CAN-bus as well as the position of the sample points.

![Diagram of CAN-bus bit structure](source: Atmel AT91SAM7A3 manual)

**The following mathematical correlations apply:**

\[
\begin{align*}
 t_{MCK0} &= \frac{1}{48 \text{ MHz}} = 20.833 \text{ ns} & \text{(system clock if } CLK = 0) \\
 t_{MCK1} &= \frac{1}{24 \text{ MHz}} = 41.667 \text{ ns} & \text{(system clock if } CLK = 1) \\
 t_{CSC} &= t_{MCKx} \cdot (BRP + 1) & \text{(bus clock)} \\
 t_{SYNCSEG} &= 1 \cdot t_{CSC} \\
 t_{PRS} &= t_{CSC} \cdot (PROP\_SEG + 1) \\
 t_{PHS1} &= t_{CSC} \cdot (PHASE\_SEG1 + 1) \\
 t_{PHS2} &= t_{CSC} \cdot (PHASE\_SEG2 + 1) \\
 t_{Bit} &= t_{SYNCSEG} + t_{PRS} + t_{PHS1} + t_{PHS2} & \text{(time of one bit on the CAN bus)} \\
 p_{Sample} &= \frac{t_{SYNCSEG} + t_{PRS} + t_{PHS1} + t_{PHS2}}{t_{SYNCSEG} + t_{PRS} + t_{PHS1} + t_{PHS2}} \cdot 100\% & \text{(sample-point)}
\end{align*}
\]
Example for 125 kbps \((PROP\_SEG = 7, PHASE\_SEG1 = 4, PHASE\_SEG2 = 1, BRP = 23, CLK=0)\):

\[
\begin{align*}
t_{SC} &= 20.833\,\text{ns} \cdot (23 + 1) = 500\,\text{ns} \\
t_{SYNCSEG} &= 1 \cdot 500\,\text{ns} = 500\,\text{ns} \\
t_{RS} &= 500\,\text{ns} \cdot (7 + 1) = 4000\,\text{ns} \\
t_{PHS1} &= 500\,\text{ns} \cdot (4 + 1) = 2500\,\text{ns} \\
t_{PHS2} &= 500\,\text{ns} \cdot (1 + 1) = 1000\,\text{ns} \\
t_{Bit} &= 500\,\text{ns} + 4000\,\text{ns} + 2500\,\text{ns} + 1000\,\text{ns} = 8000\,\text{ns} \\
\frac{1}{t_{Bit}} &= \frac{1}{8000\,\text{ns}} = 125\,\text{kbps} \\
p_{\text{Sample}} &= \frac{500\,\text{ns} + 4000\,\text{ns} + 2500\,\text{ns}}{500\,\text{ns} + 4000\,\text{ns} + 2500\,\text{ns} + 1000\,\text{ns}} \cdot 100\% = 87.5\%
\end{align*}
\]

Note:
For compatibility reasons, constant `USBCAN_BAUDEX_USE_BTR01` was defined. If this constant is used for baud rate configuration in parameter `m DwBaudrate` of structure `tUcanInitCanParam`, the parameters `m_bBTR0` and `m_bBTR1` registers become available for configuration. In this case, only the baud rates in Table 26 are available. Configuration of user-specific baud rates is not possible (error code `USBCAN_ERRCMD_ILLBDR` will be returned).

**Example 1 (compatible to first and second generation):**

```c
... // preset all init parameters
memset (&InitParam, 0, sizeof (InitParam));
InitParam.m_dwSize = sizeof (InitParam);
InitParam.m_bMode = kUcanModeNormal;
InitParam.m_bBTR0 = HIBYTE (USBCAN_BAUD_125kBit);
InitParam.m_bBTR1 = LOBYTE (USBCAN_BAUD_125kBit);
InitParam.m_bOCR = USBCAN_OCR_DEFAULT;
InitParam.m_dwAMR = USBCAN_AMR_ALL;
InitParam.m_dwACR = USBCAN_ACR_ALL;
InitParam.m_dwBaudrate = USBCAN_BAUDEX_USE_BTR01;
InitParam.m_wNrOfRxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;
InitParam.m_wNrOfTxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;

// initialize CAN-channel
bRet = UcanInitCanEx2 (UcanHandle, USBCAN_CHANNEL_CH0, &InitParam);
...```

Example 2: (not compatible to first and second generation):

tUcanHandle UcanHandle;
UCANRET bRet;
tUcanInitCanParam InitParam;

...  
// preset init parameters
memset (&InitParam, 0, sizeof (InitParam));
InitParam.m_dwSize = sizeof (InitParam);
InitParam.m_bMode = kUcanModeNormal;
InitParam.m_bBTR0 = HIBYTE (USBCAN_BAUD_USE_BTREX);
InitParam.m_bBTR1 = LOBYTE (USBCAN_BAUD_USE_BTREX);
InitParam.m_bOCR = USBCAN_OCR_DEFAULT;
InitParam.m_dwAMR = USBCAN_AMR_ALL;
InitParam.m_dwACR = USBCAN_ACR_ALL;
InitParam.m_dwBaudrate = USBCAN_BAUDSP2_125kBit;
InitParam.m_wNrOfRxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;
InitParam.m_wNrOfTxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;

// initialize CAN-channel
bRet = UcanInitCanEx2 (UcanHandle, USBCAN_CHANNEL_CH0, &InitParam);
...
4.3.4.3 Baud Rate Configuration for fourth generation USB-CANmodul

Since driver-version V5.00 a new device-revision is supported „Fourth Generation – USB-CANmodul“ (abbr. G4). Due to discontinue of components changes had to be done for the configuration of baud rates. However the software was changed in order the baud rate constants of Table 27 can still be used for the new device revision. Should other baud rate settings become necessary these settings must be done as followed:

Due to compatibility reasons the pre-defined values BTR0 and BTR1 from Table 26 can still be used for USB-CANmodul devices of fourth generation. If the value USBCAN_BAUD_USE_BTREX is used for BTR0 and BTR1, the pre-defined values of Table 27 can be used for the m_dwBaudrate as well. However the correct pre-defined values for the fourth generation are as follows:

Table 28: Constants for CAN baud rates for fourth generation (CPU frequency = 96 MHz)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Sample-point</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_BAUD_G4_10kBit</td>
<td>0x412F0077</td>
<td>CAN baud rate 10 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_20kBit</td>
<td>0x412F003B</td>
<td>CAN baud rate 20 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_50kBit</td>
<td>0x412F0017</td>
<td>CAN baud rate 50 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_100kBit</td>
<td>0x412F000B</td>
<td>CAN baud rate 100 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_125kBit</td>
<td>0x401C000B</td>
<td>CAN baud rate 125 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_250kBit</td>
<td>0x401C0005</td>
<td>CAN baud rate 250 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_500kBit</td>
<td>0x401C0002</td>
<td>CAN baud rate 500 kbps</td>
<td>87.50%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_800kBit</td>
<td>0x401B0001</td>
<td>CAN baud rate 800 kbps</td>
<td>86.67%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4_1MBit</td>
<td>0x40180001</td>
<td>CAN baud rate 1000 kbps</td>
<td>83.33%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_USE_BTR01</td>
<td>0x00000000</td>
<td>Parameters BTR0/BTR1 are used – refer to Table 26</td>
<td></td>
</tr>
</tbody>
</table>

Table 29: Constants for CAN baud rates for fourth generation (CPU frequency = 120 MHz)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Sample-point</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_BAUD_G4X_10kBit</td>
<td>0xC12F0095</td>
<td>CAN baud rate 10 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_20kBit</td>
<td>0xC12F004A</td>
<td>CAN baud rate 20 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_50kBit</td>
<td>0xC12F001D</td>
<td>CAN baud rate 50 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_100kBit</td>
<td>0xC12F000E</td>
<td>CAN baud rate 100 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_125kBit</td>
<td>0xC02F000B</td>
<td>CAN baud rate 125 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_250kBit</td>
<td>0xC02F0005</td>
<td>CAN baud rate 250 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_500kBit</td>
<td>0xC02F0002</td>
<td>CAN baud rate 500 kbps</td>
<td>85.00%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_800kBit</td>
<td>-</td>
<td>CAN baud rate 800 kbps – not supported</td>
<td>-</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_1MBit</td>
<td>0xC01B0001</td>
<td>CAN baud rate 1000 kbps</td>
<td>86.67%</td>
</tr>
<tr>
<td>USBCAN_BAUD_G4X_USE_BTR01</td>
<td>0x00000000</td>
<td>Parameters BTR0/BTR1 are used – refer to Table 26</td>
<td></td>
</tr>
</tbody>
</table>
User-defined values can be set by the user. Following the format of the baud rate register is explained.

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bit 23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>-</td>
<td>TS2[2:0]</td>
<td>TS1[3:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>BRP[9:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| BRP[7:0] |

Figure 42: Structure of baud rate register dwBaudrate for fourth generation modules

**Parameter:**

**CLK:** Clock specifies the speed of the microcontroller (since firmware version V5.11 available). If this bit is set to 0, the microcontroller internally runs with 96 MHz (standard CPU speed) and the CAN controller periphery is clocked with 24 MHz. But if this bit is set to 1, the microcontroller internally runs with 120 MHz (25% higher performance) and the CAN controller periphery is clocked with 30 MHz. This has an effect on the bit rate settings.

**SJW:** Synchronization Jump Width specifies the compensation of the phase-shift between the system clock and the different CAN-controllers connected to the CAN-bus.

**TS1, TS2:** Time Segment specifies the number of clock cycles of one bit on the CAN-bus as well as the position of the sample points.

**BRP:** Baudrate Prescaler specifies the ratio between internal clock of the microcontroller and the bus clock on the CAN-bus.

![Diagram of a CAN bus bit structure](source: STM32F205xx manual)

**Figure 43:** General structure of one bit on the CAN-bus
The following mathematical correlations apply:

\[ t_{\text{PCLK0}} = \frac{1}{24 \text{ MHz}} = 41.667\text{ns} \quad \text{(system clock if CLK = 0)} \]
\[ t_{\text{PCLK1}} = \frac{1}{30 \text{ MHz}} = 33.333\text{ns} \quad \text{(system clock if CLK = 1)} \]
\[ t_q = t_{\text{PCLKX}} \cdot (\text{BRP} + 1) \quad \text{(bus clock)} \]
\[ t_{\text{SYNCSEG}} = 1 \cdot t_q \]
\[ t_{\text{BS1}} = t_q \cdot (\text{TS1} + 1) \]
\[ t_{\text{BS2}} = t_q \cdot (\text{TS2} + 1) \]
\[ t_{\text{Bit}} = t_{\text{SYNCSEG}} + t_{\text{BS1}} + t_{\text{BS2}} \quad \text{(time of one bit on the CAN bus)} \]
\[ p_{\text{Sample}} = \frac{t_{\text{SYNCSEG}} + t_{\text{BS1}} + t_{\text{BS2}}}{t_{\text{SYNCSEG}} + t_{\text{BS1}} + t_{\text{BS2}}} \cdot 100\% \quad \text{(sample-point)} \]

Example for 125 kbps (TS1 = 12, TS2 = 1, BPR = 11, standard CPU speed = 96 MHz):

\[ t_q = 41.667\text{ns} \cdot (11 + 1) = 500\text{ns} \]
\[ t_{\text{SYNCSEG}} = 1 \cdot 500\text{ns} = 500\text{ns} \]
\[ t_{\text{BS1}} = 500\text{ns} \cdot (12 + 1) = 6500\text{ns} \]
\[ t_{\text{BS2}} = 500\text{ns} \cdot (1 + 1) = 1000\text{ns} \]
\[ t_{\text{Bit}} = 500\text{ns} + 6500\text{ns} + 1000\text{ns} = 8000\text{ns} \]
\[ \frac{1}{t_{\text{Bit}}} = \frac{1}{8000\text{ns}} = 125\text{kbps} \]
\[ p_{\text{Sample}} = \frac{500\text{ns} + 6500\text{ns} + 1000\text{ns}}{500\text{ns} + 6500\text{ns} + 1000\text{ns}} \cdot 100\% = 87.5\% \]

Example (not compatible to first, second and third generation):

```c
UCANRET bRet;

tUcanHandle UcanHandle;
UCANRET bRet;
tUcanInitCanParam InitParam;
...

// preset init parameters
memset (&InitParam, 0, sizeof (InitParam));
InitParam.m_dwSize = sizeof (InitParam);
InitParam.m_bMode = kUcanModeNormal;
InitParam.m_bBTR0 = HIBYTE (USBCAN_BAUD_USE_BTREX);
InitParam.m_bBTR1 = LOBYTE (USBCAN_BAUD_USE_BTREX);
InitParam.m_bOCR = USBCAN_OCR_DEFAULT;
InitParam.m_dwAMR = USBCAN_AMR_ALL;
InitParam.m_dwBaudrate = USBCAN_BAUD_G4_125kBit;
InitParam.m_wNrOfRxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;
InitParam.m_wNrOfTxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;

// initialize CAN-channel
bRet = UcanInitCanEx2 (UcanHandle, USBCAN_CHANNEL_CH0, &InitParam);
...
```
Note:
The higher performance of the USB-CANmodul devices only can be activated since firmware version V5.11. The activation is done by the tool USB-CANmodul Control within the tab-sheet “Hardware” using the button “Change” (refer to section 4.2.1). If the higher performance is activated, the bit rate constants USBCAN_BAUDEX_G4X... has to be used (refer to Table 29) instead of the constants USBCAN_BAUDEX_G4... (refer to Table 28). Otherwise the error code USBCAN_ERRCMD_ILLBDR (0x47) is returned by the API functions. In addition not all CAN baud rates are supported if the higher performance is activated (e.g. 800 kbps).

4.3.4.4 Use of user-defined CAN baud rates

Because the configuration of the CAN baud rate is done via register values, also other CAN baud rates can be set not listed with the previous sub-sections. For defining these CAN baud rates the mathematical correlations must be used given in the previous sub-sections. In Table 30 a selection of CAN baud rates only for USB-CANmodul devices of fourth generations are listed which have been frequently asked for.

Table 30: Examples for user-defined CAN baud rates

<table>
<thead>
<tr>
<th>CAN baud rate</th>
<th>Value / sample-point (normal CPU speed)</th>
<th>Value / sample-point (high CPU speed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.333 kbps</td>
<td>0x412F0023 / 85.00%</td>
<td>0xC02F002C / 85.00%</td>
</tr>
<tr>
<td>83.333 kbps</td>
<td>0x411E000F / 88.89%</td>
<td>0xC02F0011 / 85.00%</td>
</tr>
<tr>
<td>307.692 kbps</td>
<td>0x40190005 / 84.62%</td>
<td>Not supported</td>
</tr>
<tr>
<td>333.333 kbps</td>
<td>0x401E0003 / 88.89%</td>
<td>0xC01E0004 / 88.89%</td>
</tr>
<tr>
<td>615.384 kbps</td>
<td>0x40190002 / 84.62%</td>
<td>0xC01B0002 / 86.67%</td>
</tr>
<tr>
<td>666.667 kbps</td>
<td>0x401E0001 / 88.89%</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
4.3.5 CAN Messages Filter Function

It is possible to filter the received CAN messages by hardware. The configurations of the filter are passed to the API function `UcanInitCan()` using the parameters `dwAMR_p` and `dwACR_p`. Using the API function `UcanInitCanEx()` or `UcanInitCanEx2()` the parameters `m_dwAMR` and `m_dwACR_p` of structure `tUcanInitCanParam` are used to configure the filter. It is also possible to change these values later using the function `UcanSetAcceptance()` or `UcanSetAcceptanceEx()`.

The following mechanism is used for filtration:

![CAN message filter mechanism](image)

**Figure 44:** CAN message filter mechanism used within the USB-CANmodul

**Table 31:** CAN message filter mechanism for only accepted CAN messages

<table>
<thead>
<tr>
<th>Acceptance mask bit (AMR)</th>
<th>Acceptance code bit (ACR)</th>
<th>Bit of received message for being accepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Don’t care</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Don’t care</td>
</tr>
</tbody>
</table>

The bits of AMR / ACR corresponds to the message bits for 11-bit CAN-Identifier as follows:

![CAN message filter corresponding bits](image)

**Figure 45:** CAN message filter corresponding bits for 11-bit CAN-ID
The bits of AMR / ACR corresponds to the message bits for 29-bit CAN-Identifier as follows:

```
    Bit 31 30 ... 3 2 1 0  
    ACR  
    ↓  
    AMR  
    ↓  
    Bit 28 27 ... 0  
    29-Bit CAN-ID
```

Figure 46: CAN message filter corresponding bits for 29-bit CAN-ID

The macros `USBCAN_CALCULATE_AMR()` and `USBCAN_CALCULATE_ACR()` may be used to calculate the filter values. The data bytes cannot be passed to these macros.

**Macro:**

- `USBCAN_CALCULATE_AMR`
- `USBCAN_CALCULATE_ACR`

**Syntax:**

```
USBCAN_CALCULATE_AMR(extended,from_id,to_id,rtr_only,rtr_too)
USBCAN_CALCULATE_ACR(extended,from_id,to_id,rtr_only,rtr_too)
```

**Description:**

The macro `USBCAN_CALCULATE_AMR()` calculates the for acceptance mask register (AMR) macro `USBCAN_CALCULATE_ACR()` calculates the for acceptance code register (ACR) to be used for receiving CAN messages with the given parameters.

Note: Always pass the same values of all parameters of macro `USBCAN_CALCULATE_AMR()` as for macro `USBCAN_CALCULATE_ACR()`.

**Parameter:**

- `extended`: If non-zero the parameters `from_id` and `to_id` are specifying 29-bit CAN-Identifier. Otherwise they are specifying 11-bit CAN-Identifier.
- `from_id`: Specifies the start of the range of CAN-Identifier to be received.
- `to_id`: Specifies the end of the range of CAN-Identifier to be received (including this identifier).
- `rtr_only`: If non-zero then only RTR frames are received and the parameter `rtr_too` is ignored.
- `rtr_too`: If non-zero then data frames and RTR frames are received. Otherwise only data frames are received.

**Return:**

- `USBCAN_CALCULATE_AMR()`: the value for acceptance mask register (AMR)
- `USBCAN_CALCULATE_ACR()`: the value for acceptance code register (ACR)
**Example:**

tUcanHandle UcanHandle;
UCANRET bRet;
...

// initialize the hardware
bRet = UcanInitHardware (&UcanHandle, 0, NULL);
...

// preset init parameters
// filters 11-bit CAN messages with ID 0x600 to 0x67F,
// RTR frames not important
memcpy (&InitParam, 0, sizeof (InitParam));
InitParam.m_dwSize = sizeof (InitParam);
InitParam.m_bMode = kUcanModeNormal;
InitParam.m_bBTR0 = HIBYTE (USBCAN_BAUD_USE_BTREX);
InitParam.m_bBTR1 = LOBYTE (USBCAN_BAUD_USE_BTREX);
InitParam.m_bOCR = USBCAN_OCR_DEFAULT;
InitParam.m_dwAMR = USBCAN_CALCULATE_AMR (0,0x600,0x67F,0,0);
InitParam.m_dwACR = USBCAN_CALCULATE_ACR (0,0x600,0x67F,0,0);
InitParam.m_dwBaudrate = USBCAN_BAUDEX_G4_125kBit;
InitParam.m_wNrOfRxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;
InitParam.m_wNrOfTxBufferEntries = USBCAN_DEFAULT_BUFFER_ENTRIES;

// initialize CAN-channel
bRet = UcanInitCanEx2 (UcanHandle, USBCAN_CHANNEL_CH0, &InitParam);
...

Use the following two constants for receiving all CAN messages transferred over the CAN bus:

**Table 32:** Constants for acceptance filter for receiving all CAN messages

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_AMR_ALL</td>
<td>0xFFFFFFFF</td>
<td>Value for AMR for receiving all CAN messages.</td>
</tr>
<tr>
<td>USBCAN_ACR_ALL</td>
<td>0x00000000</td>
<td>Value for ACR for receiving all CAN messages.</td>
</tr>
</tbody>
</table>

**Example:** refer to example on page 119.
4.3.6 Using multiple CAN-channels

The **USB-CANmodul2** has two CAN-channels. This device is called “logical device” in this sub-section. However the **USB-CANmodul16** has 16 CAN-channels which are divided into 8 logical devices with 2 channels each. In other words, each logical device provides 2 CAN-channels, which need to get initialized. Refer to section 4.3.8 for more information.

An **USB-CANmodul8** behaves like an USB-CANmodul16 but includes only 4 logical devices and 8 CAN-channels. USB CANmodul2 has only one logical device and 2 CAN-channels.

Both CAN-channels of each logical device have to be initialized by using the API function **UcanInitCanEx2()**.

There are five constants to select a CAN-channel:

*Table 33: Constants for CAN-channel selection*

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_CHANNEL_CH0</td>
<td>0</td>
<td>first CAN channel</td>
</tr>
<tr>
<td>USBCAN_CHANNEL_CH1</td>
<td>1</td>
<td>second CAN channel</td>
</tr>
<tr>
<td>USBCAN_CHANNEL_ANY</td>
<td>255</td>
<td>any CAN channel</td>
</tr>
<tr>
<td>USBCAN_CHANNEL_CAN1</td>
<td>0</td>
<td>first CAN channel</td>
</tr>
<tr>
<td>USBCAN_CHANNEL_CAN2</td>
<td>1</td>
<td>second CAN channel</td>
</tr>
</tbody>
</table>

Constant USBCAN_CHANNEL_ANY can only be used with functions **UcanReadCanMsgEx()** and **UcanGetMsgPending()**.

In function **UcanReadCanMsgEx()** it indicates that the function shall examine, from which CAN-channel the next CAN message is received from. If this function returns at least one valid CAN message (refer to macro **USBCAN_CHECK_VALID_RXCANMSG()**), then it also passes the respective CAN-channel to the calling function: USBCAN_CHANNEL_CH0 or USBCAN_CHANNEL_CH1 (refer to function **UcanReadCanMsgEx()** for detailed information).

In function **UcanGetMsgPending()** it indicates that the function shall return the pending CAN messages of both CAN channels of a logical device (first and the second one).

The constants USBCAN_CHANNEL_CAN1 and USBCAN_CHANNEL_CAN2 have the same values as USBCAN_CHANNEL_CH0 and USBCAN_CHANNEL_CH1. They were defined because on top of the housing of **USB-CANmodul2**, the first channel was named CAN1 but in the software the first channel is named CH0.
4.3.7 Using the Callback Functions

The DLL library provides three types of callback functions. The **Connect Control Callback Function** informs about Plug & Play events for the USB-CANmodul (e.g.: new USB-CANmodul connected to the PC; or disconnected from the PC; ...). The second type announces events, which occur during the work with the USB-CANmodul (e.g.: CAN message receive; error status changed; ... – refer to section 4.3.7.2).

An extended format (support of multiple CAN channels, support of a user-defined argument) exists for both types of the callback function named above.

**Note:**

The "Connect Control callback" function has a different format than callback functions for the other events. Make sure to use the correct format in your application. It is not possible to use the very same implementation for both types of callback function!

Also the format of the extended callback functions differs from the format of the standard functions. Make sure to use the extended callback functions if the extended API functions are used. Access violations will occur during runtime otherwise!

Also note that the callback functions are declared as PUBLIC, which is defined as “__stdcall” in Microsoft Visual Studio.

The third type of callback functions is the **Enumeration Callback Function**. It informs about found USB-CANmodul devices calling the API function `UcanEnumerateHardware()`.

### 4.3.7.1 Connect Control Callback Function

**Function:** AppConnectControlCallback

**Syntax:**

```c
void PUBLIC AppConnectControl ( 
    BYTE  bEvent_p, 
    DWORD dwParam_p);
```

**Description:**

This callback function informs the application if a new USB-CANmodul is connected to the PC, or a connected USB-CANmodul has been disconnected. This callback function is registered with the API function `UcanInitHwConnectControl()` and may have another name within the application as named above.

**Parameter:**

- `bEvent_p`: Event which occurred (refer to Table 34).
- `dwParam_p`: Additional parameter depending on the occurred event (refer to Table 34).

**Table 34:** Constants for the event informed with the connect control callback functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Value for bParam_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_EVENT_CONNECT</td>
<td>0x06</td>
<td>A new logical USB-CANmodul is connected.</td>
<td>Don't care</td>
</tr>
<tr>
<td>USBCAN_EVENT_DISCONNECT</td>
<td>0x07</td>
<td>An USB-CANmodul is disconnected which was not used by the application.</td>
<td>Don't care</td>
</tr>
<tr>
<td>USBCAN_EVENT_FATALDISCON</td>
<td>0x08</td>
<td>An USB-CANmodul in either HW_INIT or CAN_INIT state is disconnected from the computer. Data loss is possible.</td>
<td>The USBCAN handle of the disconnected module. This handle can no longer be used.</td>
</tr>
</tbody>
</table>
Function: AppConnectControlCallbackEx

Syntax:

```c
void PUBLIC AppConnectControlEx(
    DWORD dwEvent_p,
    DWORD dwParam_p,
    void* pArg_p);
```

Description:
This callback function informs the application if a new USB-CANmodul is connected to the PC, or a connected USB-CANmodul has been disconnected. This callback function is registered with the API function `UcanInitHwConnectControlEx()` and may have another name within the application as named above.

Parameter:

- `dwEvent_p`: Event which occurred (refer to `Table 34`).
- `dwParam_p`: Additional parameter depending on the occurred event (refer to `Table 34`).
- `pArg_p`: Additional user-parameter, which was passed to function `UcanInitHwConnectControlEx()` as parameter `pCallbackArg_p`.

Example: refer to example on page 129.

Note:
If it is necessary that an application shall reconnect to an previous disconnected logical module, then the application firstly has to call the API function `UcanDeinitHardware()` after the received event `USBCAN_EVENT_FATALDISCON`. With the following event `USBCAN_EVENT_CONNECT` the logical module has to be initialized again using one of the API functions `UcanInitHardware()`, `UcanInitHardwareEx()`, `UcanInitHardwareEx2()` or `UcanEnumerateHardware()` followed by `UcanInitCanEx2()`.

Before the application can recognize the fatal disconnect any CAN messages may be lost. If so the transmissions shall be repeated after the reconnection (depending on the used CAN protocol stack and/or requirements of the application).
4.3.7.2 Event Callback Function

**Function:** AppEventCallback

**Syntax:**
```c
void PUBLIC AppEventCallback (tUcanHandle UcanHandle_p,
BYTE bEvent_p);
```

**Description:** This callback function informs the application if an event occurred on an initialized USB-CANmodul. This callback function is registered with the API function `UcanInitHardware()` and may have another name within the application as named above.

**Parameter:**
- **UcanHandle_p:** USBCAN handle that was received with the function `UcanInitHardware()`.
- **bEvent_p:** Event which occurred (refer to Table 35).

**Table 35:** Constants for the event informed with the event callback functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Value for bChannel_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBCAN_EVENT_INITHW</td>
<td>0x00</td>
<td>The USB-CANmodul is initialized successfully.</td>
<td>Don't care</td>
</tr>
<tr>
<td>USBCAN_EVENT_INITCAN</td>
<td>0x01</td>
<td>The CAN interface is initialized successfully.</td>
<td>The CAN-channel that was initialized.</td>
</tr>
<tr>
<td>USBCAN_EVENT_RECEIVE</td>
<td>0x02</td>
<td>At least one CAN message is received. May also be more than one CAN message.</td>
<td>The CAN-channel that received the CAN message(s).</td>
</tr>
<tr>
<td>USBCAN_EVENT_STATUS</td>
<td>0x03</td>
<td>The error status at the USB-CANmodul has changed.</td>
<td>The CAN-channel, which CAN error state has been changed.</td>
</tr>
<tr>
<td>USBCAN_EVENT_DEINITCAN</td>
<td>0x04</td>
<td>The CAN interface is shut down.</td>
<td>The CAN-channel that is being shut down.</td>
</tr>
<tr>
<td>USBCAN_EVENT_DEINITHW</td>
<td>0x05</td>
<td>The USB-CANmodul is completely shut down.</td>
<td>Don't care</td>
</tr>
</tbody>
</table>
Function: AppEventCallbackEx

Syntax:

```c
void PUBLIC AppEventCallbackEx (tUcanHandle UcanHandle_p,
    DWORD dwEvent_p,
    BYTE bChannel_p,
    void* pArg_p);
```

Description: This callback function informs the application if an event occurred on an initialized USB-CANmodul. This callback function is registered with the API function `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()` and may have another name within the application as named above.

Parameter:

- **UcanHandle_p**: USBCAN handle that was received with the function `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` as well as `UcanEnumerateHardware()`.
- **dwEvent_p**: Event which occurred (refer to Table 35).
- **bChannel_p**: CAN channel, which is to be used.
  - USBCAN_CHANNEL_CH0 for CAN channel 0
  - USBCAN_CHANNEL_CH1 for CAN channel 1
  - USBCAN_CHANNEL_ANY for don't care (refer to Table 35)
- **pArg_p**: Additional user-parameter, which was passed to function `UcanInitHardwareEx()` or `UcanInitHardwareEx2()` with parameter `pCallbackArg_p` as well as `UcanEnumerateHardware()` with parameter `m_pCallbackArg` of structure `tUcanHardwareInitInfo`.

Note: The callback functions should not call the API functions of the DLL directly. This can lead to undesired results. The best method for using the callback functions is to wait for an event in the main program (e.g. with the Win32 function `WaitForMultipleObjects()`) and then to call the API functions from there after the event has occurred. The callback functions only shall signal the corresponding event (e.g. with the Win32 function `SetEvent()`).
Example:

```c
// CONNECT, RECV, STATUS, ...
// CONNECT
// initialize USB-CANmodul with USBCAN_ANY_MODULE and
// register second callback function
bRet = UcanInitHardwareEx (&UcanHandle_g, USBCAN_ANY_MODULE,
    AppEventCallbackEx, NULL);
...  
// initialize CAN interface
bRet = UcanInitCanEx2 (UcanHandle_g, USBCAN_CHANNEL_CH0,
        &InitParam_g);
...  
break;

// RECV
// read CAN message
bChannel = USBCAN_CHANNEL_CH0;
dwRxCount = 100;
bRet = UcanReadCanMsgEx (UcanHandle_g, &bChannel,
        &aCanRxMsg_g, &dwRxCount);
...  
break;
```

```c
void PUBLIC AppConnectControlCallbackEx (DWORD dwEvent_p, DWORD dwParam_p,
    void* pArg_p)
{
    UCANRET bRet;

    // which event did occur?
    switch (dwEvent_p)
    {
    case USBCAN_EVENT_CONNECT: // new USB-CANmodul connected
        // Send signal to main function, so that the USB-CANmodul
        SetEvent(ahWaitHandles_g[CONNECT]);
        ...  
        break;

    case USBCAN_EVENT_DISCONNECT: // USB-CANmodul disconnected
        ...  
        break;
    }
}
```
```c
#include "canmodul.h"

void PUBLIC AppEventCallbackEx (tUcanHandle UcanHandle_p,
                                DWORD dwEvent_p, BYTE bChannel_p, void* pArg_p)
{
    // what event appeared?
    switch (dwEvent_p)
    {
        case USBCAN_EVENT_RECEIVE:    // CAN message received
            SetEvent(ahWaitHandles_g[RECV]);
            break;

        case USBCAN_EVENT_STATUS:     // changes error status
            SetEvent(ahWaitHandles_g[STATUS]);
            break;
        ... };
}
```

### 4.3.7.3 Enumeration Callback Function

<table>
<thead>
<tr>
<th>Function:</th>
<th>AppEnumCallback</th>
</tr>
</thead>
</table>
| Syntax:                 | void PUBLIC AppEnumCallback (DWORD dwIndex_p, 
|                         |   BOOL fIsUsed_p, 
|                         |   tUcanHardwareInfoEx* pHwInfoEx_p, 
|                         |   tUcanHardwareInitInfo* pInitInfo_p 
|                         |   void* pArg_p); |
| Description:            | This callback function is called from the context of the function UcanEnumerateHardware() when a connected USB-CANmodul is found which matches to the filter parameters passed to UcanEnumerateHardware(). It is registered using the function UcanEnumerateHardware() and may have a different name within the application. |
| Parameter:              |                 |
| dwIndex_p:              | Ongoing index which is incremented by the value 1 for each found USB-CANmodul. The value is 0 for the first call of this callback function. |
| fIsUsed_p:              | This flag is TRUE when the found USB-CANmodul is currently exclusively used by another application. This parameter only can be TRUE if the function UcanEnumerateHardware() was called with the parameter fEnumUsedDevs_p = TRUE. An USB-CANmodul cannot be used by the own application when it is exclusively used by another application (means the network driver is not used). |
| pHwInfoEx_p:            | Pointer to a variable of the structure of type tUcanHardwareInfoEx holding the hardware information of the found USB-CANmodul. |
| pInitInfo_p:            | Pointer to a variable of the structure of type tUcanHardwareInitInfo. This structure controls the further process of the function UcanEnumerateHardware(). This structure is detailed explained below. The user has to fill out this structure before returning from the callback function. |
| pArg_p:                 | Additional user-parameter, which was passed to function UcanEnumerateHardware() with parameter pCallbackArg_p. |
typedef struct _tUcanHardwareInitInfo
{
    DWORD           m_dwSize;
    BOOL            m_fDoInitialize;
    tUcanHandle*    m_pUcanHandle;
    tCallbackFktEx  m_fpCallbackFktEx;
    void*           m_pCallbackArg;
    BOOL            m_fTryNext;
} tUcanHardwareInitInfo;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>[Direction]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_dwSize</td>
<td>[OUT]</td>
<td>Size of this structure in bytes. This parameter is set to the value <code>sizeof(tUcanHardwareInitInfo)</code> by the DLL before calling the Enumeration Callback Function.</td>
</tr>
<tr>
<td>m_fDoInitialize</td>
<td>[IN]</td>
<td>Set to TRUE if the DLL shall automatically initialize the found USB-CANmodul. In this case the parameters <code>m_pUcanHandle</code>, <code>m_fpCallbackFktEx</code> and <code>m_pCallbackArg</code> must be filled out.</td>
</tr>
<tr>
<td>m_pUcanHandle</td>
<td>[IN]</td>
<td>Pointer to a variable of type <code>tUcanHandle</code> to receive the USBCAN handle of the found and automatically initialized USB-CANmodul. This parameter must not be NULL if <code>m_fDoInitialize</code> is set to TRUE.</td>
</tr>
<tr>
<td>m_fpCallbackFktEx</td>
<td>[IN]</td>
<td>Pointer to an event callback function (refer to <code>AppEventCallbackEx()</code>) used for the found and automatically initialized USB-CANmodul. This parameter may be NULL.</td>
</tr>
<tr>
<td>m_pCallbackArg</td>
<td>[IN]</td>
<td>User-specific parameter that is passed to the event callback function as well. This parameter may be NULL.</td>
</tr>
<tr>
<td>m_fTryNext</td>
<td>[IN]</td>
<td>Set to TRUE if the function <code>UcanEnumerateHardware()</code> shall try to find further USB-CANmodul devices. Otherwise it stops the enumeration process.</td>
</tr>
</tbody>
</table>
Example:

```c
#define APP_MAX_DEVICES 4 // <-- for example only enumerate up to 4 modules
tUcanHandle aUcanHandles_g[APP_MAX_DEVICES];
DWORD dwFoundModules_g;

int main (void)
{
    UCANRET bRet;
tUcanInitCanParam InitParam;

    ... // enumerate connected USB-CANmodul devices
dwFoundModules_g = UcanEnumerateHardware (AppEnumCallback, (void*) &InitParam,
    TRUE, // also find modules, which are currently used by other apps
    0, ~0, // no limitations for the device number
    0, ~0, // no limitations for the serial number
    0, ~0); // no limitations for the Product-Code

    // beginning from here all auto-initialized modules can be used
    if (dwFoundModules_g > 0)
    {
        // preset init parameters
        memset (&InitParam, 0, sizeof (InitParam));
        InitParam.m_dwSize               = sizeof (InitParam);
        InitParam.m_bMode                = kUcanModeNormal;
        InitParam.m_bBTR0                = HIBYTE (USBCAN_BAUD_USE_BTREX);
        InitParam.m_bBTR1                = LOBYTE (USBCAN_BAUD_USE_BTREX);
        InitParam.m_bOCR                 = USBCAN_OCR_DEFAULT;
        InitParam.m_dwAMR                = USBCAN_AMR_ALL;
        InitParam.m_dwACR                = USBCAN_ACR_ALL;
        InitParam.m_dwBaudrate           = USBCAN_BAUDEX_G4_125kBit;

        // initialize the first channel of found CAN-channel
        bRet = UcanInitCanEx2 (UcanHandle, USBCAN_CHANNEL_CH0, &InitParam);
        ...
    }
    ...
}

void PUBLIC AppEnumCallback (DWORD dwIndex_p, BOOL fIsUsed_p,
tUcanHardwareInfoEx* pHwInfoEx_p, tUcanHardwareInitInfo* pInitInfo_p,
    void* pArg_p)
{
    if (fIsUsed_p != FALSE)
    {
        printf ("module %d is already used\n", pHwInfoEx_p->m_dwSerialNr);
    }
    else if (dwIndex_p < APP_MAX_DEVICES)
    {
        printf ("initialize module %d...\n", pHwInfoEx_p->m_dwSerialNr);

        // fill out the parameters for auto-initializing
        pInitInfo_p->m_fDoInitialize = TRUE;
        pInitInfo_p->m_pUcanHandle    = &aUcanHandles_g[dwIndex_p];
        pInitInfo_p->m_fpCallbackFktEx = AppEventCallbackEx;
        pInitInfo_p->m_pCallbackArg    = (void*) &aUcanHandles_g[dwIndex_p];

        // enumerate further modules
        pInitInfo_p->m_fTryNext = TRUE;
    }
    else
    {
        // do not enumerate further modules
        pInitInfo_p->m_fTryNext = FALSE;
    }
}
```
4.3.8 Assignment of CAN-channels of Multiport devices

All Multiport devices are divided into "logical devices" which each have two CAN-channels. The USB-CANmodul8 has 4 logical devices and USB-CANmodul16 has 8 logical devices. Each logical device is pre-configured with its own device number. We recommend to keep this pre-configuration.

Additionally the serial number of each logical device (stored to the internal EEPROM) is calculated by the following formula:

\[
\text{SerialNumber}_{\text{EEPROM}} = (\text{SerialNumber}_{\text{Barcode}} \times 1000) + \text{LogicalDeviceNumber}
\]

The serial number stored to the internal EEPROM of a logical device is unchangeable. Find the Barcode Serial Number at the sticker at the backend of the table case or 19" rack-mounted case (refer to Figure 8 or Figure 12). Table 36 lists all device and serial numbers of each logical device of an USB-CANmodul8 or USB-CANmodul16.

Table 36: Assignment of CAN-channels of Multiport devices

<table>
<thead>
<tr>
<th>CAN channel on the front panel</th>
<th>Logical device number</th>
<th>Pre-defined device number</th>
<th>CAN channel of logical device</th>
<th>Parameters</th>
<th>Serial number in EEPROM</th>
<th>Example for serial number in EEPROM using Barcode serial number = 123456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 1</td>
<td>123456001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 1</td>
<td>123456001</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 2</td>
<td>123456002</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 2</td>
<td>123456002</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 3</td>
<td>123456003</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 3</td>
<td>123456003</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 4</td>
<td>123456004</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 4</td>
<td>123456004</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 5</td>
<td>123456005</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 5</td>
<td>123456005</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 6</td>
<td>123456006</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 6</td>
<td>123456006</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 7</td>
<td>123456007</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 7</td>
<td>123456007</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 8</td>
<td>123456008</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>(SerialNumber\text{Barcode} \times 1000) + 8</td>
<td>123456008</td>
<td></td>
</tr>
</tbody>
</table>

To initialize an logical device use the API function \texttt{UcanEnumerateHardware()} or \texttt{UcanInitHardware()} or \texttt{UcanInitHardwareEx()} or \texttt{UcanInitHardwareEx2()}. Refer to the Example 3 on page 62 or to the Example on page 65. These examples are using the serial number to initialize logical devices of a Multiport device. Call \texttt{UcanInitHardwareEx()} or \texttt{UcanInitHardwareEx2()} in a loop to initialize all logical devices of the Multiport device (if needed).
For each logical device the API function `UcanInitCanEx2()` needs to be called to initialize the CAN-channel 0 and/or 1 of the logical device. Use the parameters listed in Table 36 for initializing the correct CAN-channel of the device.
5 Software support for Linux OS

For the Linux operating system a Socket-CAN driver is being offered. Please ask at the help desk support for the respective article number or refer to the download-page of the SYS TEC homepage: www.systec-electronic.com.
6 Known issues

- Using VMware under Windows OS as host there can cause problems when connecting the USB-CANmodul to the guest OS. Until now we have detected problems on Renesas USB 3.0 ports. Furthermore the establishing of the connect can fail when using the USB-CANnetwork driver on the host OS.
## Index

### A
- API Functions ........................................... 55

### B
- Blinking cycles ........................................ 26
- Borland C++ Builder ................................. 52

### C
- CAN Acceptance Filter ............................. 86, 87, 121
- CAN baud rate ......................................... 85, 86, 110
- CAN baud rate user-defined .................... 120
- CAN cable parameters ............................ 22
- CAN Channel ......................................... 124
- CAN connector .......................................... 21
- CAN Error Counters ............................... 95
- CAN Error Status .................................. 83, 85
- CAN high-speed transceiver .................. 21
- CAN low speed ......................................... 99
- CAN low-speed transceiver ..................... 23
- CAN port .................................................. 46, 79, 99
- CAN single wire ........................................... 99
- CAN single-wire transceiver .................. 23
- CAN Trace File ................................. 49, 51
- CAN_INIT .................................................. 52
- Command Line Parameters ....................... 50
- Connect Control Callback ....................... 59, 60, 125
- ConvertUcanTrace ................................ 51
- Cyclic CAN messages ......................... 49, 77, 96

### D
- DB-9 plug .................................................. 21, 38
- Debug log file ......................................... 42, 55
- Demo ......................................................... 8, 52
- Device Manager .................................... 36
- Device Number ...................................... 37, 67
- DLL ......................................................... 8, 52
- DLL_INIT .................................................. 52

### E
- EMC .......................................................... 9
- Enumeration ............................................ 61
- Enumeration Callback .......................... 61, 130
- Error Codes .......................................... 105
- Event Callback ......................................... 63, 64, 127
- Expansion Port ...................................... 24, 46, 78, 103

### F
- File Structure ........................................... 45
- Firmware performance ......................... 46
- Firmware Version .................................. 59
- Function ................................................... 52
- AppConnectControlCallback .................. 125
- AppConnectControlCallbackEx ............... 126
- AppEnumCallback ................................. 130
- AppEventCallback .................................. 127
- AppEventCallbackEx ................................ 128
- UcanConfigUserPort .................................. 103
- UcanDefineCyclicCanMsg ....................... 97
- UcanDeinitCan ......................................... 73
- UcanDeinitCanEx ....................................... 73
- UcanDeinitHardware .............................. 66
- UcanDeinitHwConnectControl .................. 60
- UcanEnableCyclicCanMsg ....................... 99
- UcanEnumerateHardware ......................... 61
- UcanGetCanErrorCounter ....................... 95
- UcanGetFwVersion ..................................... 59
- UcanGetHardwareInfo .............................. 74
- UcanGetHardwareInfoEx ......................... 75
- UcanGetModuleTime .................................... 56
- UcanGetMsgCountInfo ............................. 82
- UcanGetMsgCountInfoEx ......................... 82
- UcanGetMsgPending .................................... 94
- UcanGetStatus ......................................... 83
- UcanGetStatusEx ....................................... 85
- UcanGetVersion ........................................ 57
- UcanGetVersionEx ...................................... 57
- UcanInitCan ............................................. 67
- UcanInitCanEx .......................................... 68
- UcanInitCanEx2 ........................................ 69
- UcanInitHardware ..................................... 63
- UcanInitHardwareEx .................................. 64
- UcanInitHardwareEx2 ................................ 65
- UcanInitHwConnectControl ..................... 59
- UcanInitHwConnectControlEx .................. 60
- UcanReadCanMsg ....................................... 88
- UcanReadCanMsgEx .................................... 90
- UcanReadCanPort ...................................... 101
- UcanReadCanPortEx .................................... 102
- UcanReadCyclicCanMsg ......................... 98
- UcanReadUserPort ...................................... 104
- UcanResetCan .......................................... 70
- UcanResetCanEx ........................................ 71
- UcanSetAcceptance ................................... 86
- UcanSetAcceptanceEx ............................. 87
- UcanSetBaudrate ..................................... 85
- UcanSetBaudrateEx .................................... 86
- UcanSetDebugMode .................................... 55
- UcanSetDeviceNr ...................................... 67
- UcanSetTxTimeout ...................................... 70
- UcanWriteCanMsg ....................................... 91
- UcanWriteCanMsgEx ................................... 92
- UcanWriteCanPort ..................................... 100
- UcanWriteCanPortEx ................................... 101
- UcanWriteUserPort ..................................... 103

### G
- General API functions ............................... 55
- Getting Started ....................................... 28

### H
- Hardware ................................................. 9
- Hardware Assistant .............................. 33
- Hardware Information ............................ 74, 75
- High Resolution Time Stamp .................. 69
- HW_INIT .................................................. 52
I
Installation .................................. 29
Introduction .................................. 8
J
Jumper ........................................... 22
L
LabView ........................................... 8
LIB .................................................. 52
Linux ............................................. 135
Listen Only Mode ................................ 69
M
Macro .............................................. 77, 107, 122
USBCAN_CALCULATE_ACR .................. 122
USBCAN_CALCULATE_AMR .................. 122
USBCAN_CHECK_ERROR ....................... 108
USBCAN_CHECK_ERROR_CMD ................ 109
USBCAN_CHECK_IS_G1 ....................... 80
USBCAN_CHECK_IS_G2 ....................... 80
USBCAN_CHECK_IS_G3 ....................... 80
USBCAN_CHECK_IS_G4 ....................... 80
USBCAN_CHECK_IS_SYSWORXX ............. 80
USBCAN_CHECK_SUPPORT_CYCLIC_MSG ..... 77
USBCAN_CHECK_SUPPORT_RBCAN_PORT ...... 79
USBCAN_CHECK_SUPPORT_RBUSER_PORT ..... 79
USBCAN_CHECK_SUPPORT_TERM_RESISTOR ... 79
USBCAN_CHECK_SUPPORT_TWO_CHANNEL ...... 78
USBCAN_CHECK_STATE_UCANNET ............ 78
USBCAN_CHECK_SUPPORT_USER_PORT ...... 78
USBCAN_CHECK_TX_NOTALL ................ 108
USBCAN_CHECK_TX_OK ....................... 107
USBCAN_CHECK_TX_SUCCESS ............... 108
USBCAN_CHECK_VALID_RXCANMSG .......... 107
USBCAN_CHECK_WARNING ................... 108
Message Counters ................................ 82
Module Time Stamp ................................ 66
N
Network Driver .................................. 43, 79
P
Parallel Mode .................................... 96
PCANView (USBCAN) ......................... 38, 47
Pending CAN messages .................... 94
Plug & Play .................................... 8, 125
Power LED ...................................... 26
Product Code .................................... 61, 76, 77
R
Reset CAN Channel ......................... 71
Reset CAN Controller ..................... 70
Reset Flags .................................... 71
S
Sequential Mode ................................ 96
Socket CAN .................................... 135
Software ....................................... 45
Software States ................................ 52
Status LED .................................... 26, 83
Status Timeout ................................ 84
Structure
tCanMsgStruct .................................. 88
tStatusStruct ................................... 83
tUcanChannelInfo ............................... 76
tUcanHardwareInfo .............................. 74
tUcanHardwareInfoEx ......................... 75
tUcanSoftwareInitInfo ......................... 131
tUcanInitCanParam ............................. 68
tUcanMsgCountInfo ............................ 82
Supply Voltage .................................. 21
System requirements ...................... 28
sysWORXX .................................... 10, 12, 80
T
Technical Data .................................. 10, 13, 16, 19
Termination resistor ....................... 21, 23, 38, 78, 100
Traffic LED ..................................... 26
TX Echo ......................................... 69
TX Timeout ..................................... 70
U
UCANRET ....................................... 105
Uninstallation .................................. 43
Update ......................................... 34
USB ................................................ 8
USB-CANmodul Control ..................... 46
USB-CANmodul1 ................................ 10
USB-CANmodul16 .............................. 18
USB-CANmodul2 .............................. 12
USB-CANmodul8 .............................. 15
V
Verification ..................................... 36
Version ......................................... 57
W
Watchdog Timeout ......................... 84
Do you have any suggestions for improving this manual?

Have you found any mistakes in this manual?

Sent from:

Customer number:
Name:
Company:
Address:

Send to:
SYS TEC electronic AG
Am Windrad 2
D - 08468 Heinsdorfergrund
GERMANY
Fax: +49 (0) 37 65 / 38600-4100
Email: info@systec-electronic.com