CAN-API

Part 1:
Function Description

Software Manual

esd electronic system design gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
www.esd-electronics.com • Fax: 0511/37 29 8-68
Phone: 0511/37 29 80 • International: +49-5 11-37 29 80

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd** gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd** gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd-electronics.com
Internet: www.esd-electronics.com

USA / Canada:
esd electronics Inc.
525 Bernardston Road
Suite 1
Greenfield, MA 01301
USA

Phone: +1-800-732-8006
Fax: +1-800-732-8093
E-mail: us-sales@esd-electronics.com
Internet: www.esd-electronics.us

Manual File:	I:\texte\Doku\MANUALS\PROGRAM\CAN\Schicht2\ENGLISCH\Univers\CAN-API-Functions_30.en9
Manual Order no.:	C.2001.21 This order no. covers the two parts of the API manual: Part 1: 'CAN API, Function Description' (this manual) and Part 2: 'CAN API, Installation Guide'.
Date of Print:	2007-07-26

Described Software	Revision/ Date
Windows SDK	from V2.x
Windows 95/98/ME VxD-Driver	from V1.4.x
Windows NT Device Driver	from V2.5.x
Windows 2000 Windows 2003 Server Windows Vista (32-Bit) Windows Vista (64-Bit) Windows XP Windows XP x64 Edition	from V.2.5.x
Linux Driver	V2.4.x V3.x.x
LynxOS Driver	V1.2.x
PowerMAX OS Driver	V1.1.1
Solaris-Driver	V1.2.x
SGI-IRIX6.5 Driver	V2.1.x
AIX Driver	V1.3.0
VxWorks Driver	V2.5.x
QNX4 Driver	V2.1.x
QNX6 Driver	V3.7.x
RTOS-UH Driver	V2.x.x
RTX Driver	V2.3.x

Implementation on the Following Boards	Order No.
EtherCAN	C.2050.xx
CAN-Bluetooth	C.2065.xx
CAN-ISA/200	C.2011.xx
CAN-ISA/331	C.2010.xx
CAN-PC104/200	C.2013.xx
CAN-PC104/331	C.2012.xx
CAN-PCI104/200	C.2046.xx
CAN-PCI/200	C.2021.xx
CAN-PCIe/200	C.2042.xx
CAN-PCI/266	C.2036.xx
CAN-PCI/331	C.2020.xx
CAN-PCI/360	C.2022.xx
CAN-PCI/405	C.2023.xx
PMC-CAN/266	C.2040.xx
PMC-CAN/331	C.2025.xx
PMC-CPU/405	V.2025.xx
CPCI-CAN/200	C.2035.xx
CPCI-CAN/331	C.2027.xx
CPCI-CAN/360	C.2026.xx
CPCI-405	I.2306.xx
CPCI-CPU/750	I.2402.xx
CAN-PCC	C.2422.xx
CAN-USB-Mini	C.2464.xx
CAN-USB/2	C.2066.xx
VME-CAN2	V.1405.xx
VME-CAN4	V.1408.xx

Changes in the Software and/or Documentation

Chapter	Alterations in this manual versus previous version	Alterations in software	Alterations in documentation
3.2	Chapter : “CAN Error and Fault Confinement” inserted	-	x
3.9	Further CAN modules added to table	-	x
4.3.2	*canSet Baudrate()* extended with *UBRN*	x	x
4.3.4	CAN Events extended	x	x
7.2	Chapter about WinCANTest deleted	x	x

Technical details are subject to change without notice.

Contents

Page

1. Introduction 9

2. Device Driver and NTCAN-API 10
2.1 Overview 10
2.2 Features of Device Drivers and the NTCAN-API 12
2.2.1 Summary of NTCAN-API Features 12
2.2.2 Integration Into Operating System 12
2.2.3 Plug&Play Support 12
2.2.4 Multitasking/Multithreading Support 13
2.2.5 Interaction 13
2.2.6 Multiprocessor Support 13
2.2.7 Firmware Update 13
2.2.8 Background Bus-Off Recovery 14
2.2.9 Hardware Independence 14
2.2.10 Operating System Independence 14
2.2.11 Blocking and Non-Blocking Calls 14
2.2.12 Events 14
2.2.13 Timestamps 14
2.2.14 Listen Only Mode 15
2.2.15 Scheduling 15
2.2.16 Auto Answer 15
2.2.17 Extended Error Information 15
2.2.18 Baud rate Detection 15
2.2.19 Smart Disconnect 15

3. CAN Communication with NTCAN-API 16
3.1 Basics of the CAN Communication 16
3.2 CAN Errors and Fault Confinement 17
3.3 Receiving and Transmitting in ‘FIFO Mode’ 18
3.3.1 Introduction to the FIFO Mode 18
3.3.2 Typical Usage of FIFO Mode for Reception/Transmission of CAN-Frames 19
3.4 Object Mode (Receive) 20
3.4.1 Introduction to the Object Mode 20
3.4.2 Typical Usage of Object Mode for Reception of CAN Frames 20
3.5 Object Mode (Transmit) 21
3.5.1 Scheduling of Tx-Messages 21
3.5.1.1 Introduction to the Scheduling 21
3.5.1.2 How to Use Scheduling 21
3.5.2 Auto Answer 22
3.5.2.1 Introduction to ‘Auto Answer’ mode 22
3.5.2.2 How to use ‘Auto Answer’ mode 22
3.6 Transmitting and Receiving CAN 2.0B (29-Bit) Messages 23
3.6.1 Introduction to the CAN 2.0B Message Handling 23
3.6.2 Typical Usage of the NTCAN-API to Receive CAN Messages With 29-Bit IDs 24
3.7 Listen-Only Mode 25
3.7.1 Introduction to the Listen Only Mode 25
3.7.2 Typical Sequence of Function Calls to Accomplish Reception of CAN Frames 25
3.8 Event Interface 26

3.8.1 Introduction to the Event Interface . . . 26
3.8.2 Reception of Events . . . 26
3.9 Timestamps . . . 27
3.9.1 Introduction to the Timestamping . . . 27
3.9.2 Typical Usage of the Timestamping Interface . . . 27
3.10 Matrix of Supported Operating Systems and Features . . . 28
3.11 Notes on 'Matrix of Supported Operating Systems and Features' . . . 32

4. Application Programming Interface . . . 33
4.1 Simple Data Types . . . 33
4.2 Compound Data Types . . . 35
4.2.1 CAN Message (CMSG) Data Structure . . . 35
4.2.2 Timestamped CAN Message (CMSG_T) Data Structure . . . 38
4.2.3 Event Message (EVMSG) Data Structure . . . 39
4.2.4 CAN Interface Status (CAN_IF_STATUS) Data Structure . . . 41
4.3 NTCAN API Calls . . . 43
4.3.1 Opening and Closing a CAN Channel . . . 43
canOpen() . . . 43
canClose() . . . 46
4.3.2 Configuring a CAN Channel . . . 47
canSetBaudrate() . . . 47
canGetBaudrate() . . . 50
canIdAdd() . . . 51
canIdDelete() . . . 52
canIoctl() . . . 53
4.3.3 Receiving and Transmitting CAN messages . . . 58
canTake() . . . 58
canRead() . . . 59
canTakeT() . . . 62
canReadT() . . . 63
canWrite() . . . 64
canSend() . . . 65
canGetOverlappedResult() . . . 66
4.3.4 Receiving and Transmitting Events . . . 67
canReadEvent() . . . 67
canSendEvent() . . . 68
CAN Events . . . 69
4.3.5 Retrieving Status and Information . . . 72
canStatus() . . . 72

5. Return Codes . . . 73
5.1 General Return Codes . . . 73
5.2 Special Return Values of the EtherCAN Driver . . . 83

6. Example Programs . . . 84
6.1 Example Program: Receiving Messages . . . 84
6.2 Example Program: Transmitting Messages . . . 86

7. Test Programs . . . 88
7.1 cantest for the Command Line as Example Program . . . 88

7.1.1 Functional Description 88
7.1.2 Special Features of VxWorks Implementation 89
7.1.3 Description of Displayed Board Parameters 89
7.1.4 Parameter Description 90

8. References 92

9. Appendix 93
9.1 Bit Timing Values (Examples) 93

Note: The software installation is described in the second part of the CAN-API documentation called 'CAN-API, Part 2: Installation Guide'.

Note: esd offers a set of **CAN Software Tools**, that supports efficient setup and analysis of CAN applications and networks with Windows. The tools and the according manuals are part of the CAN-SDK for Windows and can be downloaded for free from the web at:

www.esd-electronics.com/tools

This page is intentionally left blank.

1. Introduction

This manual describes the operation of the device drivers for esd-CAN modules and the use of the standard programming interface NTCAN-API in your own applications.
Additionally, programs which may be used to test the device driver are explained.

The manual has got the following structure:

Chapter 1: *Introduction* - gives an overview of this manual.

Chapter 2: *Device Drivers and NTCAN-API* - gives an overview of the possibilities of the device driver and the NTCAN-API.

Chapter 3: *NTCAN-API* - describes how the NTCAN-API can be used in your own applications to realise a communication on the CAN bus.

Chapter 4: *Programming Interface* - is the reference of NTCAN-API.

Chapter 5: *Returned Values* - describes the error values returned by NTCAN-API functions.

Chapter 6: *Example Programs* - describes two complete, small example applications which clarify the transmission and reception of CAN messages.

Chapter 7: *Test Programs* - describes the example application `cantest`, which is delivered as binary and as source code.
The test programs can be used to verify the operation of the driver as well as to execute simple CAN communication (when used in scripts).

Chapter 8: *References*

Chapter 9: *Appendix* - examples of bit timing values are listed in a table

Note: The **software installation** is described in the second part of the CAN-API documentation called ‘CAN-API, Part 2: Installation Guide’.

You can find fundamental information and further reading on the CAN bus in publications such as:

ISO 11898	Road vehicles – Interchange of digital information – Controller area Network (CAN) for high-speed communication
CAN Specification 2.0	Robert Bosch GmbH, 1991

2. Device Driver and NTCAN-API

2.1 Overview

This chapter should give an overview of the features of the esd-CAN device drivers and the NTCAN Application Programming Layers (API), based on them. The NTCAN-API is independent from the hardware used and offers the same functionality on the following esd-CAN modules:

- CAN-ISA/200, CAN-PC104/200, CAN-PCI/200, CPCI-CAN/200
- CAN-PCI/266, PMC-CAN/266, CAN-PCIe/200, CAN-PCI104/200
- CAN-ISA/331, CAN-PC104/331, CAN-PCI/331, CPCI-CAN/331, PMC-CAN/331
- CAN-PCI/360, CPCI-CAN/360, CAN-PCI/405, CPCI-405, CPCI-CPU/750
- CAN-PCC
- CAN-USB-Mini, CAN-USB/2
- CAN-Bluetooth
- EtherCAN

The NTCAN-API is implemented on the following desktop, embedded, Real Time and UNIX operating systems as communication layer between application and device driver according to the following figure.

- Windows NT, Windows 2000, Windows 2003 Server, Windows XP, Windows XP x64 Edition Windows Vista (32- and 64-bit)
- Windows 95, Windows 98, Windows ME
- Linux, LynxOS, PowerMAX OS, Solaris, SGI-IRIX6.5
- QNX4/6, VxWorks, AIX, RTOS-UH, RTX
- NetOS (on request, please contact our support team: support@esd-electronics.com)

Note: The C-interface is implemented for several operating systems on several esd-CAN boards.

The basic functions are available for each esd CAN module. Depending on hardware design, operating systems and development state of the modules there may be limitations in the functions.
Please note the chapter 'Matrix of Supported Operating Systems and Features' from page 28 on and the individual notes in the description of the calls for further information.

Note: A DOS driver is available for many esd CAN modules as well, but this driver is completely independent from the CAN API and therefore not described in this document. You will find the description of the DOS driver in the Document 'C Interface Library for DOS and Win 3.11'.

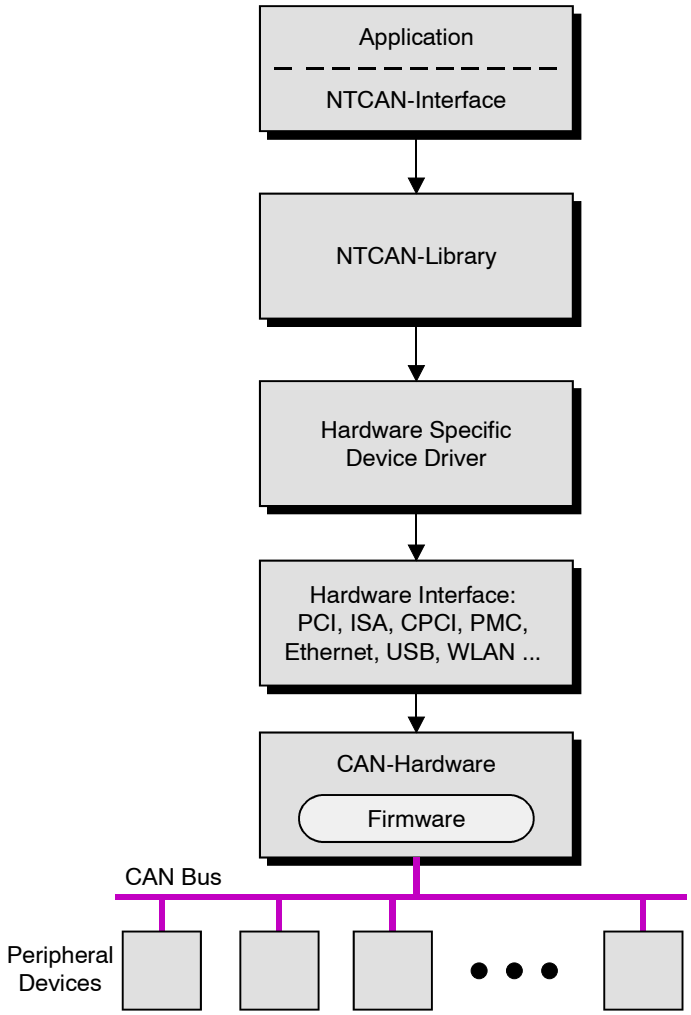

Fig. 2.1.1: Embedding the NTCAN-API into the system structure

2.2 Features of Device Drivers and the NTCAN-API

2.2.1 Summary of NTCAN-API Features

- Best possible driver integration in the operating system
- Plug&Play support for PCI and Hot-Plugging for USB
- Multitasking/multithreading support
- CAN message interaction
- Multiprocessor support
- Background bus-off recovery
- Firmware update for CAN modules with local operating system
- Hardware independent CAN node number mapping
- Operation system independent API
- Blocking and non-blocking function calls
- General and specific event messages
- Timestamps for received frames and events
- Listen only mode for non destructive CAN bus monitoring
- Scheduling (single event or cyclically) Tx-frames
- Auto answer
- Extended error information about CAN bus state
- CAN bus baud rate detection

2.2.2 Integration Into Operating System

In order to integrate new hardware, the device drivers always use the interface made available by the operating system. This ensures that the CAN driver is integrated in the operating system like the drivers of other devices. As a result CAN-applications based on NTCAN automatically benefit from reliability and security mechanisms provided by the operating system (e.g. separation of process address spaces).

The NTCAN-API always uses the mechanisms made available by the operating system to communicate with the device driver according to the previous figure, so that other interfaces with a different functionality and API can be used at the same time as the NTCAN-API.

2.2.3 Plug&Play Support

In order to simplify the driver installation for PCI boards, either the mechanisms of Plug & Play operating systems are used or enumeration of the PCI bus is done by the driver provided that the BIOS has configured the PCI boards correctly. For interfaces such as USB hot plugging is supported.

2.2.4 Multitasking/Multithreading Support

All drivers are able to virtualize the CAN hardware, therefore several applications (processes, tasks) and/or several threads of a task can use the CAN hardware simultaneously.

2.2.5 Interaction

If a CAN message is transmitted on a certain CAN identifier on a certain CAN bus by a process, this message is also received by other processes which wait for CAN messages with this CAN identifier on the same physical CAN bus. This behaviour is represented in the following figure. It is called *Interaction* and offers possibilities such as to operate a CANopen master and a CANopen slave on one computer or on just one interface.

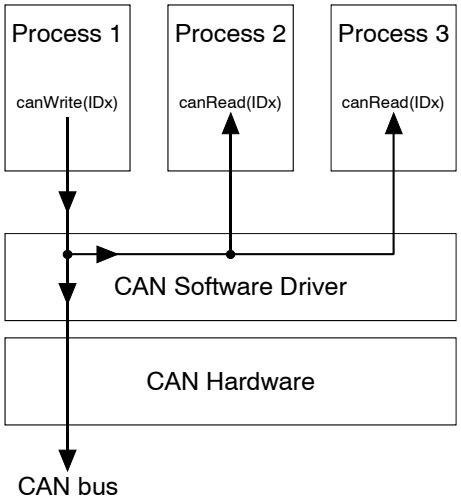

Fig. 2.2.1: Interaction example

The messages are not passed on via interprocess mechanisms of the respective operating system, but are linked to the successful transmission of the message on the CAN bus by the driver. This means that local receivers get the data at the same time as other CAN bus participants. However, the system can only work interactively, if another CAN node acknowledged the reception of the message (meaning: there is a working physical CAN bus with at least two nodes).

2.2.6 Multiprocessor Support

Drivers for operating systems which support more than one processor have been developed with the aim to operate them in a multiprocessor environment.

2.2.7 Firmware Update

All active CAN modules with their own microprocessor have got their own firmware. Most device drivers can update this firmware without running the CAN hardware under another operating system.

2.2.8 Background Bus-Off Recovery

If a CAN controller received too many error frames, it changes into 'Bus Off' status and does not participate in the communication via CAN bus anymore (in accordance to ISO11898). The device driver (passive CAN modules) or the firmware (active CAN modules) tries to reinitialize automatically the CAN controller after a certain time and participates in the communication via the bus again. The application does not have to take special action for this.
Also refer to chapter "CAN Error and Fault Confinement" on page 17.

2.2.9 Hardware Independence

A device driver supports the simultaneous operation of up to five CAN modules. Depending on the CAN hardware used, each module offers up to four physical networks.
This means that with five CAN-PCI/405 boards a single PC could access 20 CAN networks simultaneously.

In addition, device drivers for different esd-CAN modules can be run simultaneously in various operating systems. When the device driver is started, logical network numbers are assigned to the physical networks. Via these numbers the individual networks can be accessed from the application via the NTCAN-API. By means of the network numbers the CAN bus can be accessed independent from the hardware, so that it is possible to change from one esd-CAN module type to another easily without influencing the application.

2.2.10 Operating System Independence

Because of the operating system-independent definition of the NTCAN-API, applications developed on this base are portable between different operating systems regarding the CAN communication.

2.2.11 Blocking and Non-Blocking Calls

The driver supports non-blocking function calls for programmers that prefer polling operation for their application and blocking function calls for event-driven programming. One should mention, that blocking calls naturally offer better error handling.

2.2.12 Events

The driver supports general events for all CAN modules and firmware and/or hardware specific events. These events communicate amongst other things errors on the CAN bus.

2.2.13 Timestamps

Timestamps for received CAN frames or events are supported by the driver using special calls. This offers the possibility to correlate received traffic of several CAN busses on a time scale.

2.2.14 Listen Only Mode

This mode is designed for CAN bus monitoring without any influence on other CAN nodes or in combination with the baud rate detection for implementation of nondestructive 'hot plugging' of CAN nodes.

2.2.15 Scheduling

The CAN driver supports scheduling of Tx-frames: Transmission of frames can be initiated at a defined moment or cyclically. This feature is essentially useful for residual bus simulation or generation of sync-frames etc..

2.2.16 Auto Answer

The driver/CAN controller (supported by Full-CAN controllers, e.g. Intel 82527) automatically answers a received CAN-RTR frame with a user configurable CAN frame.

2.2.17 Extended Error Information

This feature provides additional detailed information about the state of the CAN bus, if the employed CAN controller supports error analysis. This comprises information about protocol errors, error counter, etc..
Also refer to chapter "CAN Error and Fault Confinement" on page 17.

2.2.18 Baud rate Detection

If supported by the hardware a device driver can determine the baud rate of a physical CAN bus without affecting the communication on the bus in a destructive way and afterwards seamlessly join the communication on the bus.
This is only possible for the default bit rates of the esd bit rate table, which covers the CiA bit timing recommendation and if there is active communication on the bus.

2.2.19 Smart Disconnect

The default behaviour of a device driver is to stay active on the CAN bus with the last configured baud rate, even if there is no application running, using this physical CAN port. This behaviour is adjuvant if you want to have an active CAN node connected to the bus, e.g. to acknowledge CAN messages.
Some drivers are able to leave the CAN bus if there is not any application using this physical CAN port. The configuration of this behaviour is done during load time of the driver and can not be changed during runtime. The configuration method is operating system dependent and is described in the driver installation manual.

3. CAN Communication with NTCAN-API

This chapter offers an overview over the operation of the CAN communication with the NTCAN-API, before the API will be explained in greater detail in the following chapters.

3.1 Basics of the CAN Communication

An application which wants to access the CAN bus has to open a logical link to the device driver which is represented by a handle. This handle has to be provided as parameter in all API calls. Different processes can open several handles simultaneously.

The logical network number specified when a handle is opened clearly links a handle with a physical interface and the CAN bus connected to it. Each handle is assigned a set of parameters for receive and transmit buffer as well as reception and transmission timeout. An individual filter can be configured for each handle to select the received messages independently for each application on the basis of 11-bit CAN identifiers (standard format). For the filter of 29-bit CAN-identifiers (extended format) for each handle one acceptance mask can be configured (see page 23). The CAN identifiers of the messages to be transmitted do not have to be reported explicitly to the device driver before.

Before data can be transmitted, the baud rate for the CAN interface has to be initialised once, either by directly setting it or by using the feature ‘automatic baud rate detection’. The initialisation is then valid for this logical network number in the whole system and therefore for the assigned physical interface. In order to prevent two applications from using the same interface with different baud rates, you can request the configured baud rate and then act correspondingly.

In order to transmit messages the API offers blocking and non-blocking services. This enables the calling process or thread to either *block* until the message has been transmitted successfully on the CAN bus or the timeout assigned to the handle has expired (synchronous transmission), or *return* immediately and make the device driver execute the transmission of messages asynchronously to the caller. Synchronous transmission provides the means of gathering information about the transmission of every CAN frame.

In order to receive messages the API offers blocking and non-blocking calls. This enables the caller either to check whether new data is available in the receive buffer (*polling*), or to *block* until one or more messages corresponding to the filter criteria of this handle have been received or the timeout assigned to this handle has expired.

In the case of an error each API call returns a corresponding error code. The individual error codes will be explained in greater detail from page 73 on.

3.2 CAN Errors and Fault Confinement

Because CAN nodes are able to distinguish between permanent failure and temorary disturbances, fault confinement is possible on the bus.
Within each CAN node an 8-bit transmit error counter and an 8-bit receive error counter are used. If one of the five error types CRC error, stuff error, form error, bit error or Acknowledgement (ACK) error is detected, the corresponding error counter is increased.
If a reception or transmission is completed successfully, the corresponding error counter is decremented. Consequently permanent failures result in large counts, whereas temporary disturbances result in small counts that recover back to zero in a running system.

If an error is detected during reception, the Rx error counter is increased by 1.The error counter is increased by 8, if the first bit after transmission of the error flag is dominant, which suggests that the error is not detected by other nodes.
The Tx error counter is always increased by 8, if an error is detected while the node is transmitting.
After a successful reception the Rx-error counter is decreased by 1 and after a successful transmission the Tx error counter is decreased by 1. Thus only the error counters of a node will increment rapidly, if a fault is local to this node.
Depending on the value of its error counters the node takes one of the three states *error active*, *error passive* or *bus off* (see CAN Events `NTCAN_EV_CAN_ERROR`, page 69). .

error active — regular operational state of the node, with both counts less than 128.
In this state the node can participate in usual communication.
If it detects any error during communication, an ERROR ACTIVE FLAG, consisting of 6 dominant bits, is transmitted. This blocks the current transmission.

error passive — when either counter exceeds 127, the node is declared error passive. This indicates that there is an abnormal level of errors. The node still participates in transmission and reception, but it has an additional time delay after a message transmission before it can initiate a new message transfer of its own. This extra delay for the *error passive* node which is known as *suspended transmission* results from transmission of 8 additional recessive bits at the end of the frame. This means that an error passive node loses arbitration to any error active node regardless of the priority of their IDs.
When an *error passive* node detects an error during communication, it transmits an ERROR PASSIVE FLAG consisting of 6 recessive bits. These will not disturb the current transmission (assuming another node is the transmitter) if the error turns out to be local to the *error passive* node.

bus off — when the transmit error count exceeds 255 the node is declared *bus off*. This indicates that the node has experienced consistent errors whilst transmitting.
This state restricts the node from sending any further transmission. The node will eventually be re-enabled for transmission and become *error active* after it has detected 128 occurrences of 11 consecutive recessive bits on the bus which indicate periods of bus inactivity.

3.3 Receiving and Transmitting in 'FIFO Mode'

3.3.1 Introduction to the FIFO Mode

The CAN communication with the NTCAN-API is based on so-called message queues which are also called FIFO (First-In-First-Out) buffers. They contain sequences of CAN messages.

Each handle is assigned a receive and a transmit FIFO whose size is defined when the handle is opened. In the Rx-FIFO CAN messages are stored in the chronological order of their reception. By calling a read operation (*canRead()* or *canTake()*) one or more CAN messages are copied from the handle FIFO into the address space of the application. By calling a write operation (*canWrite()* or *canSend()*) one or more CAN messages are copied from the address space of the application into the transmit FIFO and the CAN messages are transmitted on the CAN bus in their chronological order.

A *canRead()* call returns with new data until the receive FIFO is completely empty. Then the calling thread either blocks until new data is available or until no new CAN messages have been received for this handle within the reception timeout. A *canTake()* call always returns immediately, if the FIFO is empty, however, it does not copy data into the address space of the application. If the receive FIFO gets overrun by the driver because the application does not process the CAN messages fast enough, the oldest data is overwritten and the error is indicated to the application. This may happen at *canRead()* and *canTake()* calls.

A *canWrite()* call blocks the calling thread until all CAN messages have been transmitted successfully or returns with an error, if a message can not be transmitted within the configured transmission timeout. If there is not enough space in the transmit FIFO to take all data of the application, only the fitting data is transmitted. The user has to evaluate the return values to handle this situation.
A *canSend()* call always returns immediately and the CAN messages are transmitted asynchronously to the calling thread.

If you do not want to make use of the message queues, you can set their length to '1'. This means for reception that the receive FIFO always contains the CAN message it has received last.

3.3.2 Typical Usage of FIFO Mode for Reception/Transmission of CAN-Frames

1. Open a CAN-handle with *canOpen()* (see page 43).
   On opening the handle configure the sizes of the FIFOs and parameterise the timeouts separately for reception and transmission. As result you get a handle to access a specific physical CAN-net.

   Note: The timeouts can be changed later on by using the appropriate *canIoctl()* command (see page 53).

2. Set the desired baud rate with *canSetBaud rate()* (see page 47) for the physical CAN-net.

   Note: This might already be done by another CAN-handle, thread or application. Thus you are advised to check the baud rate of the CAN-bus in advance with *canGetBaudrate()* (see page 50).

3. Configure the message filter using *canIdAdd()* (see page 51).
   If you want to receive CAN traffic, you need to enable at least one CAN-ID, for transmission this step might be skipped.

4.a) Reception of CAN frames:
   Now you can use *canRead()* (see page 59) or *canTake()* (see page 58) to retrieve the messages, which were received from the CAN-bus and passed your message filter.

   Note: Either call is able to retrieve several CAN messages at once, depending on the configured FIFO size and the size of the buffer given by the user. Depending on the actual number of messages received from the CAN bus the user buffer might not be entirely filled or even empty, when these calls return. You need to evaluate the relating <*len*> parameter to cope with this case.

4.b) Transmission of CAN frames:
   Use *canWrite()* (see page 64) or *canSend()* (see page 65) to transmit CAN frames over the CAN bus.

   Note: Either call is able to send multiple messages. But only *canWrite()* can deliver reliable information about the actual transmission status of the CAN-messages.

3.4 Object Mode (Receive)

3.4.1 Introduction to the Object Mode

In addition to the FIFO mode for receiving data and chronologically storing each CAN message which is then evaluated by the application, as described above, in some applications only the current data of a CAN message, the data received last, needs to be evaluated by the application. This is made possible by drivers supporting an object mode which can be activated individually for each handle instead of the default mode. The selection of the reception mode does not influence the transmission of CAN messages with this handle via a FIFO and the possibilities of configuring a reception filter.
If the object mode has been activated for a handle, only the non-blocking *canTake()/canTakeT()* can be called for the reception of data. Calling *canRead()/canReadT()* returns with an error message.

In contrast to calling *canTake()/canTakeT()* in FIFO-mode, the CAN identifiers have to be initialized in the specified data structure (user buffer) before calling, because the device driver uses this information to determine the CAN messages which are of interest to the application. The selection and order of messages can be determined again by the application every time *canTake()/canTakeT()* is called, but it has to correspond to the configuration of the message filter.

At the moment the object mode is supported by Windows drivers, Linux drivers newer than 3.x.x, VxWorks and QNX6 drivers.
Linux drivers older than 3.x.x and drivers of the operating systems LynxOS, PowerMAX OS, Solaris, SGI-IRIX6.5 and AIX do not support the object mode at the moment (see tables from page 28 on).

By calling *canStatus()* and checking the feature bits, you can determine, whether a particular driver supports the object mode.

3.4.2 Typical Usage of Object Mode for Reception of CAN Frames

1. Open a CAN-handle with *canOpen()* (see page 43) and specify at least `NTCAN_MODE_OBJECT` for the mode parameter.
   The parameter *receive timeout* will be ignored. The *receive buffer size* should be set to the number of objects (different CAN-IDs) one wants to receive. If you are unsure about this, use the number of 11-Bit CAN-IDs (2048).

2. Set the desired baud rate with *canSetBaudrate()* (see page 47) for the physical CAN-net.

   Note: This might already be done by another CAN-handle/thread or application. Thus you are advised to check the baud rate of the CAN-bus in advance with *canGetBaudrate()* (see page 50).

3. Configure the message filter using *canIdAdd()* (see page 51).

4. Use *canTake()* (obviously *canRead()* makes no sense in this mode) to retrieve the latest data transmitted on a certain CAN-ID (see page 58). To accomplish this you need to initialize the elements of CMSG structure(s) with the requested CAN-IDs in advance of calling *canTake()*.

3.5 Object Mode (Transmit)

3.5.1 Scheduling of Tx-Messages

3.5.1.1 Introduction to the Scheduling

Scheduling enables you to schedule the transmission of CAN frames and furthermore it is even possible to create 'jobs', which cyclically transmit a certain CAN frame.

3.5.1.2 How to Use Scheduling

1. Open a CAN-handle with *canOpen()* (see page 43).

2. Set the desired baud rate with canSetBaudrate() (see page 47) for the physical CAN-net.

   Note: This might already be done by another CAN-handle, thread or application. Thus you are advised to check the baud rate of the CAN-bus in advance with *canGetBaudrate()* (see page 50).

3. Configure the message filter using *canIdAdd()* (see page 51). If you want to receive CAN traffic, you need to enable at least one CAN-ID, for sole use of scheduling this step is optional.

4. Create objects to use for scheduling using *canIoctl()* (see page 53) with `NTCAN_IOCTL_TX_OBJ_CREATE`. Tx-objects are referenced by their CAN-ID. You can create one Tx-object per physical net and CAN-ID.

5. Configure the objects by using *canIoctl()* with `NTCAN_IOCTL_TX_OBJ_SCHEDULE`.
   Each Tx-object can have a different timing configuration with parameters like:
   - first time of transmission
   - interval of transmission

6. Start the scheduling with *canIoctl()* with `NTCAN_IOCTL_TX_OBJ_SCHED_START`.
   At this point the afore configured schedule is at work (and can not be changed until it is stopped) and the Tx-objects will be send as scheduled.

Note: Special behaviour concerning 'injection' of frames using normal FIFO-mode besides to scheduling.
Example: If an object is created for CAN-ID 5 and scheduled for transmission every 500 ms and *canSend()* (page 65) or *canWrite()* (page 64) are used on the same CAN-handle to send a frame on CAN-ID 5 'out of order', the CAN-data of the object with CAN-ID 5 will be updated to the last data sent. The next (and all following) scheduled transmissions send the data, injected by *canSend()* or *canWrite()*!

Note: It is possible to use several CAN-handles to define different scheduling sets or to do everything else provided by the other CAN-API function, but even for several handles, every CAN-ID can be scheduled only one time for a physical net!

3.5.2 Auto Answer

3.5.2.1 Introduction to 'Auto Answer' mode

'Auto Answer' mode enables you to automatically answer on certain CAN frames (RTR-frames) without having to bother with reception or analysis of CAN messages at all.

Imagine a small application in an autonomous CAN thermometer. You receive temperature values every second from an A/D-converter and want your thermometer to present the temperature on CAN bus. Using the 'Auto Answer' mode this is quite simple. All you have to do, is to generate a 'Auto Answer'-Tx-Object and afterwards update the contained data every time you're a/D-converter has generated a new value.

3.5.2.2 How to use 'Auto Answer' mode

1. Open CAN handle with *canOpen()* (see page 43).

2. Set the desired baud rate with *canSetBaudrate()* (see page 47) for the physical CAN-net.

   Note: This might already be done by another CAN-handle, thread or application. Thus you are advised to check the baud rate of the CAN-bus in advance with *canGetBaudrate()* (see page 50).

3. Configure the message filter using *canIdAdd()* (see page 51). You need to enable at least the one CAN-ID you want to use for 'Auto Answer'.

4. Create Tx-objects to use for 'Auto Answer' using *canIoctl()* (see page 53) with `NTCAN_IOCTL_TX_OBJ_CREATE`. Tx-objects are referenced by their CAN-ID.
   You can create one Tx-object per physical net and CAN-ID.

5. Set the objects into 'Auto Answer' mode (`NTCAN_IOCTL_TX_OBJ_AUTOANSWER_ON`)

6. Use *canIoctl()* with `NTCAN_IOCTL_TX_OBJ_UPDATE` to provide new data to your 'Auto Answer' object as often and anytime you like.

   Note: Of course it is possible to use the 'Auto Answer' mode in combination with e.g. normal FIFO-modes.

3.6 Transmitting and Receiving CAN 2.0B (29-Bit) Messages

3.6.1 Introduction to the CAN 2.0B Message Handling

Nearly all esd-CAN modules also support the transmission and reception of CAN messages with 29-bit identifiers in accordance with the CAN 2.0B-standard. The following restrictions have to be observed for this procedure:

The object mode described above supports only 11-bit identifiers. In FIFO-mode the message filter based on single CAN identifiers can only be used for 11-bit identifiers. If the first 29-bit identifier is activated for a handle in FIFO-mode, all CAN messages with 29-bit identifiers received from that point will be stored in the reception FIFO of that particular handle.

As an option an acceptance mask can be configured for each handle. The acceptance filter is realized by a logical AND-combination of an acceptance code followed by a logical OR-combination with the acceptance mask according to the following figure. The acceptance code is the last 29-bit CAN identifier activated for this handle.

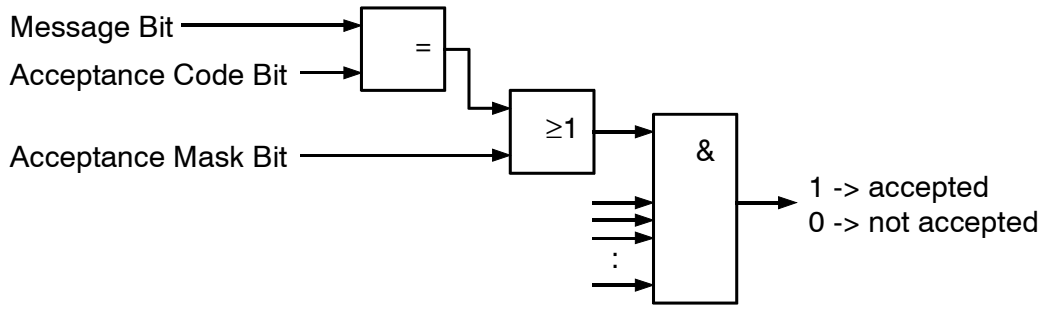

Fig. 3.4.1: Operation of the acceptance filter

The figure above shows that an active bit within the acceptance mask results in a *don't care* condition for the result of the comparison of received message bit and acceptance code bit. It is possible to limit the filter exactly to one 29-bit CAN identifier or one group of 29-bit CAN identifiers. The following table shows some examples of bit combinations of the acceptance mask and acceptance code and their meaning for the filter:

Acceptance Code	Acceptance Mask	Filter
0x00000100	0x00000000	Only 29-bit messages with the CAN identifier 0x100 are stored in the receive FIFO of the handle.
0x00000100	0x000000FF	All 29-bit CAN messages within the identifier area 0x100 ... 0x1FF are stored in the receive FIFO of the handle.
any	0x1FFFFFFF	All 29-bit CAN messages are stored in the receive FIFO of the handle (open mask). **default value**

Table 3.4.1: Combination examples of acceptance mask and acceptance code

To be reverse compatible the combination in the last row of the table above shows the default function for the reception of 29-bit CAN messages, if no mask is configured.

With one handle you can receive CAN messages with 11-bit (standard format) *and* CAN messages with 29-bit (extended format). You can use the table-based filter for 11-bit CAN identifier *and* the mask-based filter for 29-bit CAN identifier simultaneously.

3.6.2 Typical Usage of the NTCAN-API to Receive CAN Messages With 29-Bit IDs

In general this works the same way as described for 'FIFO mode'.

1. Open a CAN-handle with *canOpen()* (see page 43).

2. If needed, set baud rate with *canSetBaudrate()* (see page 47).

3. Configure the message filter:

3.a) In the simplest case, you are generally interested in CAN frames with any 29-Bit identifier. All you need to do is add any 29-Bit identifier with *canIdAdd()* (see page 51).

3.b) If you are interested in CAN frames with certain 29-Bit identifiers, you need to set an 'acceptance code' and an 'acceptance mask' as shown in the table above. The 'acceptance code' is set with *canIdAdd()* and the 'acceptance mask' is set with *canIoctl()* (see page 53) using the `NTCAN_IOCTL_SET_20B_HND_FILTER` command.

> **Note:** Of course you can combine any way of 29-Bit acceptance filtering together with normal 11-Bit ID-filtering as described earlier.

4. Use *canRead()* (see page 59) or *canTake()* (see page 58) to receive CAN frames.

3.7 Listen-Only Mode

3.7.1 Introduction to the Listen Only Mode

Many esd CAN boards support a special bus monitoring mode, the so called *listen-only mode*. If this mode is configured (via *canSetBaudrate()*, see page 47), the reception of valid CAN messages is possible, but starting a transmission of CAN messages is impossible. Neither acknowledgment nor error frames are send out upon successful or erroneous reception of a CAN message.

This mode supports applications that require monitoring the bus without any influence on other CAN nodes or which want to implement nondestructive 'hot plugging' onto a CAN bus.

The support of this mode is very hardware (CAN controller) dependent. By calling *canStatus()* and checking the feature bits, the application can determine, whether the particular driver/hardware supports the listen only mode.

3.7.2 Typical Sequence of Function Calls to Accomplish Reception of CAN Frames

In the *listen-only mode* this works exactly the same way as reception of CAN frames in FIFO mode, with the only exception of setting a 'special' baud rate.

1. Open CAN handle with *canOpen()* (see page 43).

2. Use *canSetBaudrate()* to set the baud rate for this physical CAN net and combine the baud rate index with the `NTCAN_LISTEN_ONLY_MODE` flag with a bitwise OR (`baudrate | NTCAN_LISTEN_ONLY_MODE`)

3. Configure the message filter using *canIdAdd()* (see page 51).

4. Use *canRead()* (see page 59) or *canTake()* (see page 58) to receive CAN frames.

> **Note:** Obviously you are not able to send frames in 'Listen-Only mode' and you need to pay attention, that no other CAN-handle, thread or application sets a 'normal' baud rate.

3.8 Event Interface

3.8.1 Introduction to the Event Interface

In addition to error return values and API calls for status information, the driver can indicate errors and/or status changes, like e.g. a CAN bus-off has occurred, asynchronously with an event mechanism. These events are also timestamped if this feature is supported, so these events can be related to the CAN messages with respect to the time of occurrence.

3.8.2 Reception of Events

Reception of events works basically the same way as the reception of CAN messages:

1. Open CAN handle with *canOpen()* (see page 43).

2. Configure the message filter using *canIdAdd()* (see page 51) for receiving the event IDs you are interested in.

3. Receive the events:

   3.a) Use *canRead()* (see page 59) as you are used to. If you have enabled 'normal' CAN IDs in addition to event IDs you need to distinguish between 'normal' CAN frames and events by looking at bit 30 of the CAN identifier. Cast the CMSG to an EVMSG, if the frame received was an event.
   Of course this works with timestamps, too. Just use *canReadT()* (see page 63) or *canTakeT()* (see page 62) instead.

   3.b) DEPRECATED:
   Use *canReadEvent()* just as you would use *canRead()* to receive the event message (works only, if there are only event IDs activated).

3.9 Timestamps

3.9.1 Introduction to the Timestamping

Several esd modules support time stamping of received CAN messages as they arrive or of events as they occur. The time stamping is done, depending on the CAN module, either in hardware or in software by the driver. In case of hardware timestamps the highest accuracy is reached, in case of software timestamps the jitter depends on the real-time behaviour of the operating system.

By calling *canStatus()* and checking the corresponding feature bit, the application can determine, whether the particular driver/hardware supports timestamping.

Timestamped CAN messages can be received in FIFO as well as in object mode using *canReadT()* or *canTakeT()* in the same way as described above for *canRead()* and *canTake()*.

Using the NTCAN API the timestamps have no default resolution to prevent time consuming calculations in the driver. Instead, the timestamps are realized as 64-bit free-running counter with the most accurate available time stamping source. The application can query the frequency of the time stamping source and the current timestamp in order to scale timestamps online or offline and to link the timestamp with the absolute system time.

3.9.2 Typical Usage of the Timestamping Interface

In the following we just describe the differences to normal 'FIFO mode':

1. Open CAN handle with *canOpen()* (see page 43).

2. You will most likely want to do this step somewhere at the beginning of your application.

   To get the most out of the timestamping feature you need the following information:
   - The frequency of the timestamp-counter (this may vary over different CAN hardware or operating systems). This can be accomplished by using *canIoctl()* (see page 53) with the `NTCAN_IOCTL_GET_TIMESTAMP_FREQ` command.

   - The current state of the timestamp counter, in order to correlate received timestamps with your system time. This can be accomplished by using *canIoctl()* with the `NTCAN_IOCTL_GET_TIMESTAMP` command.

3. Set the desired baud rate with *canSetBaudrate()* (see page 47) for the physical CAN-net.

4. Configure the message filter using *canIdAdd()* (see page 51).

5. Use *canReadT()* (see page 63) or *canTakeT()* (see page 62) instead of *canRead()* or *canTake()* to receive CAN frames.
   Have a look at the description of the timestamped CAN message structure CMSG_T (see page 38) to see, how to retrieve the actual timestamp from received CAN frames.

3.10 Matrix of Supported Operating Systems and Features

CAN Module	Order no.	Non-Real Time Windows Operating Systems					Real Time Windows OS	
		Windows95	Windows98 WindowsME	WindowsNT	Windows2000 Windows Server 2003 WindowsXP Windows Vista (32-bit)	Windows XP x64 Edition Windos Vista (64-bit)	Windows CE .NET (Win CE .4.2)	Windows RTX
EtherCAN	C.2050.xx	-	-	4	4	4	-	-
CAN-Bluetooth	C.2065.xx	-	-	-	4	4	-	-
CAN-USB-Mini	C.2422.xx	-	4, 5, 10	-	4, 5, 6, 8, 10, 11, 12, 16	4, 5, 6, 8, 10, 11, 12, 16	4, 5, 10	-
CAN-USB/2	C.2466.xx	-	-	-	4, 5, 6, 8, 10, 11, 12, 16	4, 5, 6, 8, 10, 11, 12, 16	4, 5, 10	-
CAN-PCC	C.2422.xx	4, 5	4, 5	4, 5	1, 4, 5	-	-	-
CAN-ISA/200	C.2011.xx	3, 4, 5, 16	3, 4, 5, 16	4, 5, 16	1, 4, 5, 16	-	-	-
CAN-ISA/331	C.2010.xx	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	1, 4, 5, 9, 10, 16	-	-	-
CAN-PC104/200 (SJA1000 version)	C.2013.xx	3, 4, 5, 16	3, 4, 5, 16	4, 5, 16	1, 4, 5, 16	-	-	-
CAN-PC104/200 (82527 version)	C.2013.xx	-	-	-	-	-	-	-
CAN-PC104/331	C.2012.xx	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	1, 4, 5, 9, 10, 16	-	-	-
CAN-PCI104/200	C.2046.xx	-	-	-	4, 5, 6, 8, 10, 11, 12, 13, 16	4, 5, 6, 8, 10, 11, 12, 13, 16		5, 7, 8, 11, 16
CAN-PCI/200	C.2021.xx	4, 5, 16	4, 5, 16	4, 5, 16	4, 5, 6, 8, 10, 11, 12, 13, 16	4, 5, 6, 8, 10, 11, 12, 13, 16	-	5, 7, 8, 11, 16
CAN-PCIe/200	C.2042.xx	-	-	-	4, 5, 6, 8, 10, 11, 12, 13, 16	4, 5, 6, 8, 10, 11, 12, 13, 16	-	5, 7, 8, 11, 16
CAN-PCI/266	C.2036.xx	-	-	4, 5, 16	4, 5, 6, 8, 10, 11, 12, 13, 16	4, 5, 6, 8, 10, 11, 12, 13, 16	-	5, 7, 8, 11, 16
CAN-PCI/331	C.2020.xx	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 8, 9, 10, 16	4, 5, 8, 9, 10, 16	4, 5, 10, 16	5, 7, 8, 10, 16
CAN-PCI/360	C.2022.xx	4, 5, 10, 16	4, 5, 10, 16	4, 5, 10, 16	4, 5, 10, 16	4, 5, 10, 16	-	5, 7, 8, 10, 16
CAN-PCI/405	C.2023.xx	-	-	-	4, 5, 7, 8, 10, 16	-	-	5, 7, 8, 10, 16
PMC-CAN/266	C.2040.xx	-	-	4, 5, 16	4, 5, 6, 8, 10, 11, 12, 13, 16	4, 5, 6, 8, 10, 11, 12, 13, 16	-	5, 7, 8, 11, 16
PMC-CAN/331	C.2025.xx	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	-	5, 7, 8, 10, 16
CPCI-CAN/200	C.2035.xx	4, 5, 16	4, 5, 16	4, 5, 16	4, 5, 16	4, 5, 16	-	5, 7, 8, 11, 16
CPCI-CAN/331	C.2027.xx	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 9, 10, 16	4, 5, 10, 16	5, 7, 8, 10, 16
CPCI-CAN/360	C.2026.xx	4, 5, 10, 16	4, 5, 10, 16	4, 5, 10, 16	4, 5, 10, 16	4, 5, 10, 16	-	5, 7, 8, 10, 16
VME-CAN2	V.1405.xx	-	-	-	-	-	-	-
VME-CAN4	V.1408.xx	-	-	-	-	-	-	-

... supports only 11-bit-identifiers

... supports 11-bit and 29-bit-identifiers

\- ... no support for this CAN module using this operating system

1... x ... further notes on supported features see page 32

Table 3.9.1: Supported Windows operating systems and driver features

CAN Module	Order no.	Unix Operating Systems					
		Linux	LynxOS	PowerMAX OS	Solaris	SGI-IRIX6.5	AIX
EtherCAN	C.2050.xx	15, 16	-	-	-	-	-
CAN-Bluetooth	C.2065.xx	-	-	-	-	-	-
CAN-USB-Mini	C.2422.xx	5, 7, 8, 10, 12, 16	-	-	-	-	-
CAN-USB/2	C.2066.xx	5, 7, 8, 10, 12, 16	-	-	-	-	-
CAN-PCC	C.2422.xx	-	-	-	-	-	-
CAN-ISA/200	C.2011.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	-
CAN-ISA/331	C.2010.xx	5, 7, 8, 10, 12, 16		-		-	-
CAN-PC104/200 (SJA1000 version)	C.2013.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	-
CAN-PC104/200 (82527 version)	C.2013.xx	5, 7, 8, 11, 12, 16	-	-	-	-	-
CAN-PC104/331	C.2012.xx	5, 7, 8, 10, 12, 16		-		-	-
CAN-PCI104/200	C.2064.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	
CAN-PCI/200	C.2021.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	-
CAN-PCIe/200	C.2042.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	
CAN-PCI/266	C.2036.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	-
CAN-PCI/331	C.2020.xx	5, 7, 8, 10, 12, 16		-		2	
CAN-PCI/360	C.2022.xx	5, 7, 8, 10, 12, 16	-	-	-	-	-
CAN-PCI/405	C.2023.xx	5, 7, 8, 10, 12, 13, 14, 16	-	-	-	2	-
PMC-CAN/266	C.2040.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	-
PMC-CAN/331	C.2025.xx	5, 7, 8, 10, 12, 16		-		2	
CPCI-CAN/200	C.2035.xx	5, 6, 7, 8, 11, 12, 13, 14, 16	-	-	-	-	-
CPCI-CAN/331	C.2027.xx	5, 7, 8, 10, 12, 16		-		2	
CPCI-CAN/360	C.2026.xx	5, 7, 8, 10, 16	-	-	-	-	-
VME-CAN2	V.1405.xx	2	2		-	-	-
VME-CAN4	V.1408.xx	2	2			-	-

... supports only 11-bit-identifiers

... supports 11-bit and 29-bit-identifiers

\- ... no support for this CAN module using this operating system

1... x ... further notes on supported features see page 32

Table 3.9.2: Supported Unix operating systems and driver features

CAN Module	Order no.	Real Time Operating Systems			
		VxWorks (5.4, 5.5, 6.x)	QNX4	QNX6	RTOS-UH
EtherCAN	C.2050.xx	-	-	-	-
CAN-Bluetooth	C.2065.xx	-	-	-	-
CAN-USB-Mini	C.2422.xx	-	-	-	-
CAN-USB/2	C.2066.xx	-	-	-	-
CAN-PCC	C.2422.xx	-	-	-	-
CAN-ISA/200	C.2011.xx	5		5, 7, 8, 11, 13, 16	-
CAN-ISA/331	C.2010.xx	5, 10	10	5, 7, 8, 10, 16	-
CAN-PC104/200 (SJA1000 version)	C.2013.xx	5		5, 7, 8, 11, 13, 16	-
CAN-PC104/200 (82527 version)	C.2013.xx	-	-	5, 7, 8, 11, 16	-
CAN-PC104/331	C.2012.xx	5	10	5, 7, 8, 10, 16	-
CAN-PCI104/200	C.2046.xx	5, 6, 8, 11, 12, 13, 16		5, 6, 7, 8, 11, 13, 16	
CAN-PCI/200	C.2021.xx	5, 6, 8, 11, 12, 13, 16	-	5, 6, 7, 8, 11, 13, 16	-
CAN-PCIe/200	C.2042.xx	5, 6, 8, 11, 12, 13, 16		5, 6, 7, 8, 11, 13, 16	
CAN-PCI/266	C.2036.xx	5, 6, 8, 11, 12, 13, 16	-	5, 6, 7, 8, 11, 13, 16	-
CAN-PCI/331	C.2020.xx	5, 8, 10, 16	10	5, 7, 8, 10, 16	-
CAN-PCI/360	C.2022.xx	5, 8, 10, 16	-	5, 7, 8, 10, 16	-
CAN-PCI/405	C.2023.xx	-	-	5, 7, 8, 10, 13, 16	-
PMC-CAN/266	C.2040.xx	5, 6, 8, 11, 12, 13, 16	-	5, 7, 8, 11, 13, 16	-
PMC-CAN/331	C.2025.xx	5, 8,10, 16	10	5, 7, 8, 10, 16	-
CPCI-CAN/200	C.2035.xx	5, 6, 8, 11, 12, 13, 16	-	5, 6, 7, 8, 11, 13, 16	-
CPCI-CAN/331	C.2027.xx	5, 8, 10, 16	10	5, 7, 8, 10, 16	-
CPCI-CAN/360	C.2026.xx	5, 8, 10, 16	-	5, 7, 8, 16	-
VME-CAN2	V.1405.xx	-	-	-	16
VME-CAN4	V.1408.xx	5	-	-	-

... supports only 11-bit-identifiers

... supports 11-bit and 29-bit-identifiers

\- ... no support for this CAN module using this operating system

1... x ... further notes on supported features see page 32

Table 3.9.3: Supported real time operating systems and driver features

esd additionally offers hardware devices with a *local* operating system, e.g. CPCI host boards or embedded systems, that support custom applications.

This driver is generated from the same source as used for the drivers for the CAN modules listed above and therefore also supports the functions described in this manual.

CAN Module	Order no.	Local Operating Systems			
		Linux	VxWorks (5.4, 5.5)	QNX6	RTOS-UH
PMC-CPU/405	V.2020.xx	5, 7, 8, 11, 13, 16	5, 6, 8, 11, 12, 13, 16	5, 7, 8, 11, 13, 16	16
CPCI-405	I.2306.04	5, 7, 8, 11, 13, 16	5, 6, 8, 11, 12, 13, 16	5, 7, 8, 11, 13, 16	16
CPCI-CPU/750	I.2402.xx	5, 7, 8, 11, 13, 16	5, 6, 8, 11, 12, 13, 16	-	-
EPPC-405	I.2001.xx	5, 7, 8, 11, 13, 16	5, 6, 8, 11, 12, 13, 16	5, 7, 8, 11, 13, 16	16
EPPC-405-HR	I.2006.xx	-	-	-	16
EPPC-405-UC	I.2005.xx	-	-	-	-

... supports only 11-bit-identifiers

... supports 11-bit and 29-bit-identifiers

\- ... no support for this CAN module using this operating system

1... x ... further notes on supported features see page 32

Table 3.9.4: Operating systems and driver features of host boards and embedded systems

3.11 Notes on 'Matrix of Supported Operating Systems and Features'

Note	Description
1	... these modules are run under Windows 2000/XP using the Windows NT driver
2	... the CAN driver of these operating systems for the VMEbus-CAN boards has to be build individually at the customer's system
3	... separate CAN driver for 11-bit IDs and 29-bit-IDs available. The driver can be selected during installation via the Windows hardware wizard. Modules which have still got the CAN controller 82C200 (until about 12/1999) do generally not support 29-bit-IDs.
4	... supports overlapped operations (see page 66)
5	... supports 'Rx-object mode' (see page 20)
6	... supports 'listen only mode' (see page 25)
7	... supports 'Tx-scheduling' (see page 21)
8	... supports 'timestamps' (see page 27)
9	... firmware supports the CAN protocol *DeviceNet*
10	... firmware update is supported (see manual part 2: 'CAN-API, Installation Guide' [1])
11	... supports 'smart disconnect' from CAN bus (see page 15)
12	... supports baud rate change event `NTCAN_EV_BAUD_CHANGE` (see page 70)
13	... supports 'auto baud rate detection' (see page 15)
14	... supports 'extended error information' (see page 15)
15	... supports 'ELLSI' (EtherCAN Low Level Socket Interface) (see manual part 2: 'CAN-API, Installation Guide' [1]))
16	... supports extended 29-bit Rx-filter (see page 23)

Table 3.10.1: Notes on supported features

4. Application Programming Interface

The following chapter describes the simple and compound data types, the constants and application programming interface (API) of the NTCAN library. The error return codes are described in a special chapter from page 73 on.

4.1 Simple Data Types

In 1997 the NTCAN API was defined to support 32-bit CPUs with 32-bit operating systems using standard C data types according to the so called ILP32 data model (see table with data type models below) which is used by all (32-bit) operating systems. With the ongoing move to 64-bit CPUs and 64-bit operating systems, which execute 32-bit code as well as 64-bit code, it became necessary to change the API to use data size neutral abstract data types, as for 64-bit operating systems the LP64 as well as the LLP64 data models are prevalent.

Data type	ILP32	LP64	LLP64
int	32	32	32
long	64	32	64
pointer	32	64	64

Table 4.1.1: Data type sizes in bits for different data models

For this reason newer versions of the NTCAN API header `ntcan.h` utilize the type names defined in ISO C99 standard (included via the header file `stddef.h`) for signed and unsigned fixed-size integer data types.
For compilers which are not C99-compliant these types are defined in the `ntcan.h` header using compiler- and OS-specific knowledge of the native data types. Furthermore the handle and the return values, which are also OS-specific, are represented with a dedicated simple data type. The next table gives an overview about the simple data types used for NTCAN.

Data type	Description
int8_t	8-bit signed integral type.
uint8_t	8-bit unsigned integral type.
int16_t	16-bit signed integral type.
uint16_t	16-bit unsigned integral type.
int32_t	32-bit signed integral type.
uint32_t	32-bit unsigned integral type.
int64_t	64-bit signed integral type.
uint64_t	64-bit unsigned integral type.
NTCAN_HANDLE	OS-specific representation of handle
NTCAN_RESULT	OS-specific representation of return values

Table 4.1.2: Simple data types supported in NTCAN API

Caveats:	The introduction of the abstract data types, as described above, has not affected the binary interface of the NTCAN API. Nevertheless there are several C/C++ compiler which issue warning messages if existing applications based on the NTCAN header with C native data types are re-compiled with a NTCAN header with abstract data types, even if the data type size in bits and the data typed signedness have not been changed.

4.2 Compound Data Types

This chapter describes the compound data types comprising the simple data types described above used by the NTCAN API calls.

4.2.1 CAN Message (CMSG) Data Structure

```
typedef struct
{
   int32_t  id;          /* 11 or 29-bit CAN identifier           [in, out] */
   uint8_t  len;         /* bit 0-3 = number of data bytes 0...8  [in, out] */
                         /* bit 4      = RTR                      [in, out] */
                         /* Bit 5      = No_Data (Object Mode)    [out]     */
                         /*            = Interaction Data (FIFO Mode)       */
                         /* Bit 6-7    = reserved                           */
   uint8_t  msg_lost;    /* counter for lost Rx messages          [out]     */
   uint8_t  reserved[2];/* reserved                                          */
   uint8_t  data[8];     /* 8 data bytes                          [in, out] */
} CMSG;
```

In order to transmit a message with a 29-bit identifier bit 29 of the CAN parameter *id* has to be set in structure CMSG additionally to the CAN identifier. This can be achieved by bitwise OR the constant `NTCAN_20B_BASE` (defined in header `ntcan.h`). If a message with a 29-bit identifier is received, bit 29 of the CAN identifier parameter *id* is set additionally to the CAN identifier.

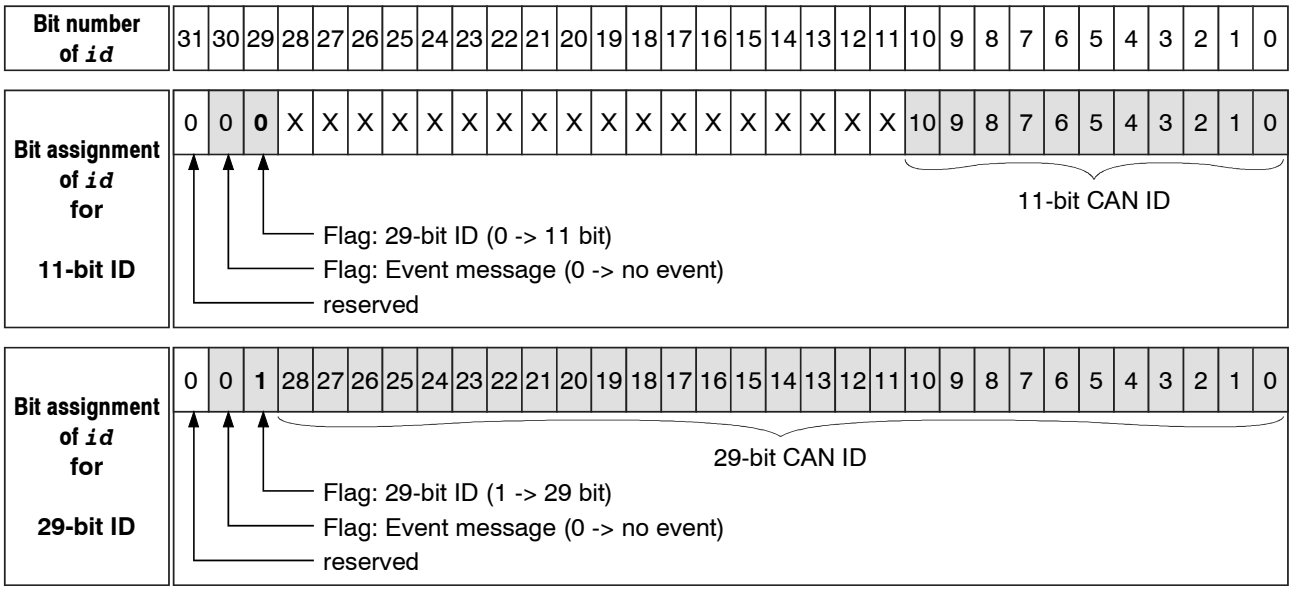

Table 4.2.1: Coding of *id* in CMSG

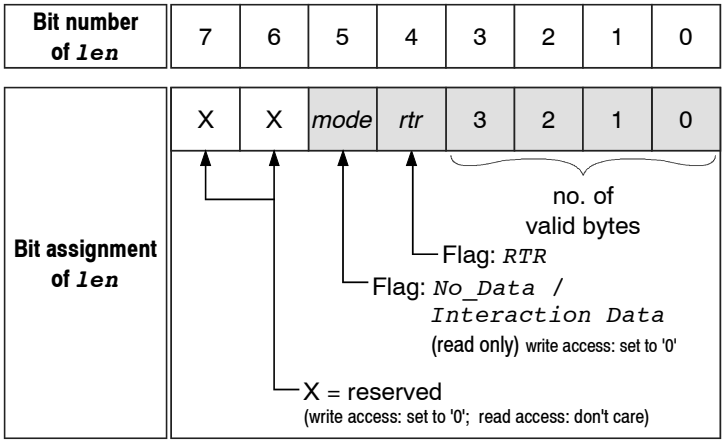

Table 4.2.2: Coding of *len*

The bits 0 to 3 of the parameter *len* in the structure CMSG are used to indicate the number of valid bytes in the data field of the received or transmitted data telegram according to the following table:

Value of *len* [binary] bit7... ...bit0	Number of Valid Data Bytes [bytes]
xxxx 0000	0
xxxx 0001	1
xxxx 0010	2
xxxx 0011	3
xxxx 0100	4
xxxx 0101	5
xxxx 0110	6
xxxx 0111	7
xxxx 1000	8
xxxx 1xxx	8 *

* According to CiA standard for *len* values > 8 are allowed.

Table 4.2.3: Coding of the number of valid data bytes

The *RTR*-bit (bit 4) of field *len* in structure CMSG differentiates a data frame from a remote frame in case of transmitting or receiving CAN messages (in header `ntcan.h NTCAN_RTR` is defined):

Value of Bit *RTR* [binary]	Function
0	transmit/receive data telegram
1	transmit/receive remote request telegram

Table 4.2.4: Function of bit *RTR*

In case of a RTR-frame the bits 0 to 3 of the field *len* reflect the data-length code according to the table at page 35. The data in *data* of structure CMSG is invalid.

Bit 5 of the field len is reserved for transmission of messages. In case of the reception of CAN frames the meaning of the bit differs in the FIFO mode and the Object mode:

Bit 5 in FIFO mode

In FIFO mode this bit indicates if the message is received via the interaction mechanism, i.e. from an application that uses the same logical net (in header `ntcan.h`, `NTCAN_INTERACTION` is defined) or not.
This special marking only results, if the handle defined for the reception is explicitly opened in mode `NTCAN_MODE_MARK_INTERACTION`.

Value of Bit *Interaction* [binary]	Function (FIFO Mode Only)
0	message not received via interaction
1	message received via interaction

Table 4.2.5: Function of bit *Interaction*

Bit 5 in

In Object mode this bit indicates if the data of the related CAN identifier is valid (in header `ntcan.h NTCAN_NO_DATA` is defined) or not:

Value of Bit *No_Data* [binary]	Function (Object Mode Only)
0	data received valid
1	data received invalid

Table 4.2.6: Function of bit *No_Data*

Bits 6...7 are reserved for future applications. During read operations the condition of these bits is undefined. During write operations these bits and also bit 5 should always be set to '0'.

If the receive FIFO of the handle gets overrun by new messages, the oldest messages are overwritten and the *msg_lost* counter is increased.
By means of the *msg_lost* counter the user can detect a data overrun. An increasing counter value indicates that the application program processes the CAN data flow slower than the driver provides new data.

Value of the Message-Lost Counter	Meaning
msg_lost = 0	no lost messages
0 < *msg_lost* < 255	number of lost frames = value of *msg-lost*
msg_lost = 255	number of lost frames ≥ 255

Table 4.2.7: Value range of *msg_lost*

4.2.2 Timestamped CAN Message (CMSG_T) Data Structure

```
typedef struct
{
   int32_t  id;          /* 11 or 29-bit CAN identifier           [in, out] */
   uint8_t  len;         /* bit 0-3 = number of data bytes 0...8  [in, out] */
                         /* bit 4      = RTR                      [in, out] */
                         /* Bit 5      = No_Data (Object Mode)    [out]     */
                         /*            = Interaction Data (FIFO Mode)       */
                         /* Bit 6-7    = reserved                           */
   uint8_t  msg_lost;    /* counter for lost Rx messages          [out]     */
   uint8_t  reserved[2];/* reserved                                          */
   uint8_t  data[8];     /* 8 data bytes                          [in, out] */
   uint64_t timestamp;  /* Timestamp of this message             [in, out] */
} CMSG_T;
```

The CMSG_T structure extends the CMSG structure with a 64-bit timestamp (see page 27 for details).

All other structure members are the same as described for the CMSG structure above.

4.2.3 Event Message (EVMSG) Data Structure

```
typedef struct
{
   int32_t  evid;        /* event ID: 0x40000000 hex ... 0x400000FF          */
   uint8_t  len;         /* number of event-data bytes: bit 0-3 = length 0-8 */
                         /*                             bit 4-7 = reserved   */
   uint8_t  msg_lost;    /* counter for lost messages                 [out]  */
   uint8_t  reserved[2]; /* reserved                                         */
   union {
     uint8_t    c[8];
     uint16_t   s[4];
     uint32_t   l[2];
   } evdata;
} EVMSG;
```

The size in bytes of the structure EVMSG is the same as the size of the structure CMSG. Events are always marked by bit 30 of the parameter *evid* set to '1' so they can be distinguished from the CAN 2.0A and CAN 2.0B messages of the structure CMSG described above, if returned with standard receive I/O operation together with CAN messages. It is perfectly OK to cast a CMSG to an EVMSG, if a frame is received via *canRead()* or *canTake()* and is identified as an event. The same can be done with CMSG_T respectively.
At the moment the range of valid events is limited to 255 and partitioned according to the table below.

Range [hex]	Description
0x40000000 ... 0x4000007F	Common events for all CAN modules
0x40000080 ... 0x400000FF	Firmware and/or hardware specific events

Table 4.2.8: Coding the length *len*

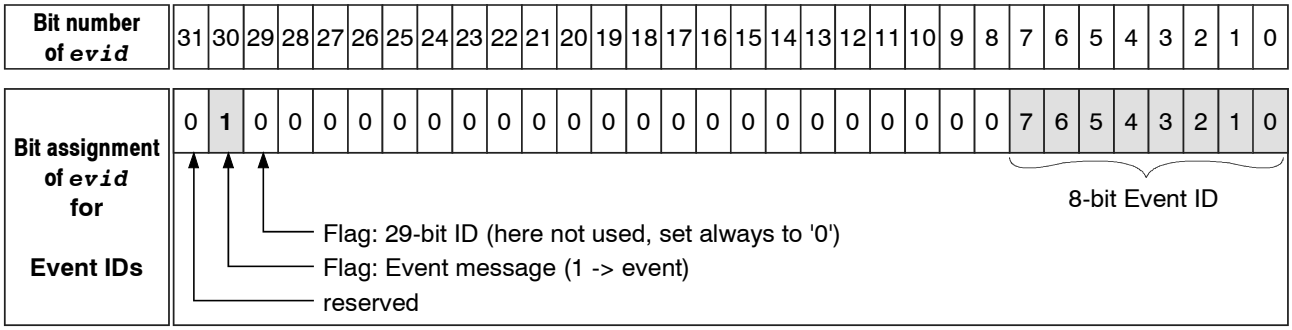

Table 4.2.9: Coding of `evid` in EVMSG

The NTCAN header defines

`NTCAN_EV_BASE` (0x40000000 hex),
`NTCAN_EV_USER` (0x40000080 hex) and
`NTCAN_EV_LAST` (0x400000FF hex).

Value of *len* [binary] bit7... ...bit0	Number of Data Bytes [bytes]
xxxx 0000	0
xxxx 0001	1
xxxx 0010	2
xxxx 0011	3
xxxx 0100	4
xxxx 0101	5
xxxx 0110	6
xxxx 0111	7
xxxx 1000	8

Table 4.2.10: Coding the length *len* in struct EVMSG

Bits 7...4 are reserved for future applications. For read accesses the value of these bits is undefined. For write accesses these bits are to be set to ‘0’.

4.2.4 CAN Interface Status (CAN_IF_STATUS) Data Structure

Data Structure of Status Message

```
typedef struct
{
   uint16_t hardware;          /* hardware revision number of the board  */
   uint16_t firmware;          /* revision number of the used firmware   */
   uint16_t driver;            /* revision number of the used software driver */
   uint16_t dll;               /* revision number of the used dll        */
   uint32_t board_status;      /* error status of the board              */
   uint8_t  board_id [14];     /* board ID name (zero terminated)        */
   uint16_t features;          /* properties of hardware and driver      */
} CAN_IF_STATUS;
```

Coding of parameters:

hardware, firmware, driver, dll

The current version number of the hardware, firmware and the software driver is returned as a four-digit hexadecimal number.
The MSB corresponds to a major and minor number, each containing 4 bits, and the LSB to a revision number, containing 8 bits.
The value 0x120C (hex) has to be read as version number ‘1.2.12’.

15 12	11 8	7 0
Major version number	Minor version number	Revision number

board_status

The status of the board is coded as follows:
0 ... OK
< > ... Error

board_id

The ID name of the detected board as zero-terminated ASCII string.

features

The parameter contains flags which show a particular configuration or particular properties of hardware and/or device driver. The following flags are defined:

Bit	Flag	Meaning
0	NTCAN_FEATURE_FULL_CAN	If the flag is set, a FullCAN controller (such as Intel 82527) is used on the CAN module, otherwise a BasicCAN controller (such as Philips SJA1000) is used.
1	NTCAN_FEATURE_CAN_20B	If the flag is set, CAN messages with 29-bit-IDs (CAN 2.0B mode) can be transmitted and received. If the flag is not set, only messages with 11-bit-IDs (CAN 2.0A mode) can be transmitted and received.
2	NTCAN_FEATURE_DEVICE_NET	If the flag is set, the firmware supports the CAN protocol *DeviceNet*.
3	NTCAN_FEATURE_CYCLIC_TX	If the flag is set, the firmware has got a customized extension to cyclically transmit Tx-messages (CAN-PCI/360 only).
4	NTCAN_FEATURE_RX_OBJECT_MODE	If the flag is set, a support for the *Rx-Object-Mode* is available.
5	NTCAN_FEATURE_TIMESTAMP	If the flag is set, the CAN hardware/firmware supports timestamping of messages (see page 27).
6	NTCAN_FEATURE_LISTEN_ONLY_MODE	If the flag is set, the CAN hardware supports the Listen-Only mode (see page 25)
7	NTCAN_FEATURE_SMART_DISCONNECT	If the flag is set, the CAN hardware/firmware supports leaving the CAN bus if no application is using a physical CAN port (see page 15)

Table 4.2.11: Flags of parameter *features*

4.3 NTCAN API Calls

4.3.1 Opening and Closing a CAN Channel

canOpen()

Name: **canOpen()** - Getting a handle for read and write operations

Synopsis:

```
NTCAN_RESULT canOpen
        (
        int             net,          /* network number */
        uint32_t        flags,
        int32_t         txqueuesize,  /* number of entries in the message queue */
        int32_t         rxqueuesize,  /* number of entries in the message queue */
        int32_t         txtimeout,    /* Tx timeout in ms */
        int32_t         rxtimeout,    /* Rx timeout in ms */
        NTCAN_HANDLE    *handle       /* out: CAN handle */
        )
```

Description: By means of this function a CAN handle is returned for all further I/O calls. The maximum number of handles is limited to 1024 by the driver at the moment. Please note that the maximum number of handles can also be limited by process limits and system-global limits, depending on the operating system used.

> *canOpen()* has to be called **at first**, before any other function is called !
> This is necessary, because all other API entries need the returned handle of *canOpen()* as logical input parameter.

The handle is assigned to a logical network via the *net* parameter.

Parameter *flags* determines special properties of the handle and configures the message-type filter. The following flags are defined in header `ntcan.h` at the moment.

Flag	Description
NTCAN_MODE_OVERLAPPED	**Only for Windows operating systems (except RTX):** This flag opens the handle for overlapped-I/O-operations. If this flag is set, only overlapped I/O-operations are possible with this handle! That means that the overlapped parameters of *canRead()* and *canWrite()* have to be supplied.
NTCAN_MODE_OBJECT	This flag opens the handle for reception in object mode instead of FIFO-mode. *canRead()* cannot be called again for reception and the message filter flags described afterwards are without effect.
NTCAN_MODE_NO_RTR	This flag configures the message-type filter in such a way that CAN remote request messages (RTRs) will not be received.
NTCAN_MODE_NO_DATA	This flag configures the message-type filter in such a way that no CAN data frames will be received.
NTCAN_MODE_NO_ INTERACTION	This flag configures the message-type filter in such a way that no CAN messages will be received via interaction mechanism.
NTCAN_MODE_MARK_ INTERACTION	This flag configures the handle in such a way that CAN messages that have been received via the interaction mechanism will be marked in the length field of the message.

Table 4.3.1: Flags for special handle properties in header `ntcan.h`

txqueuesize defines the size of the handle Tx-FIFO. The absolute maximum value for *txqueuesize* is defined in the header file `ntcan.h` by `NTCAN_MAX_TX_QUEUESIZE` (at the moment 16383).

rxqueuesize defines the size of the handle Rx-FIFO. The absolute maximum value for *rxqueuesize* is defined in the header file `ntcan.h` by `NTCAN_MAX_RX_QUEUESIZE` (at the moment 16383).

txtimeout defines the timeout interval in ms for write accesses with timeout. At *canWrite()* calls, the Tx message has to be transmitted within the time *txtimeout*.

txtimeout = 0 (no timeout, function waits endlessly)
1...65535 (0xFFFF HEX) (timeout in ms)

rxtimeout defines the timeout interval in ms for read accesses with timeout. The function *canRead()* is terminated with a timeout error, if not at least one Rx message is received within the time defined by *rxtimeout*.

rxtimeout = 0 (no timeout, function waits endlessly)
1...65535 (0xFFFF HEX) (timeout in ms)

If the function returns `NTCAN_SUCCESS`, the CAN handle is returned in *handle*.

Return: `NTCAN_SUCCESS` is returned after a successful execution, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

canClose()

Name: **canClose()** - Closing a handle

Synopsis:

```
NTCAN_RESULT canClose
        (
        NTCAN_HANDLE    handle      /* CAN handle */
        )
```

Description: By means of this function a CAN handle and all assigned resources are released. For all CAN IDs which are still enabled in the handle filter mask, *canIdDelete()* is called implicitly.

Return: `NTCAN_SUCCESS` is returned after a successful execution, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

Note: Frames sent with *canSend()* are not necessarily transmitted when closing the handle. If you need to assure this, send a single frame using *canWrite()* before closing the handle.

4.3.2 Configuring a CAN Channel

canSetBaudrate()

Name: **canSetBaudrate()** - Initialization of the CAN interface

Synopsis:

```
NTCAN_RESULT canSetBaudrate
        (
        NTCAN_HANDLE    handle,     /* CAN handle          */
        uint32_t        baud        /* bit rate constant   */
        )
```

Description: By means of this function the CAN interface is initialized and the bit rate *baud* is set. A module is passive on the CAN bus until the bit rate has been set at least once.

The structure of the 32 bit argument *baud* is defined below:

31	30	29	28... ...24	23... ...16	15... ...8	7... ...0
UBR	LOM	UBRN				
0	*LOM*	0	reserved		*Table index*	
0	*LOM*	1	reserved	numerical value		
1	*LOM*	0	reserved		*BTR0*	*BTR1*

The combination *UBR* = *UBRN* = 1 should not be used!

Bit(s)	Value	Description
UBR	0	Use pre-defined bit rate table (*Table index*) (in combination with *UBRN*)
	1	Set bit rate register of CAN controller directly (*BTR0/BTR1*)
LOM	0	Configure bit rate in active mode (normal operation)
	1	Configure bit rate in listen-only mode
UBRN	0	Use the pre-defined bit rate table (in combination with *UBR*)
	1	Set bit rate by numerical value
Table index	x	Use bit rate of pre-defined table printed on page 48
BTR0/BTR1	x	Bit rate register of CAN controller

Table 4.3.2: Description of argument *baud*

If the 'User Bit Rate' flag *(UBR*) and the 'User Bit Rate Numerical' flag *(UBRN*) are set to '0' the bits 0...15 are evaluated as index of a pre-defined bit rate table in order to configure common CAN bit rates without knowing any hardware details of the CAN controller.

Table index [hex]	Bit Rate * [kBit/s]	Constant
0	1000	NTCAN_BAUD_1000
E *2)	800	NTCAN_BAUD_800
1	666.6	-
2	500	NTCAN_BAUD_500
3	333.3	-
4	250	NTCAN_BAUD_250
5	166	-
6	125	NTCAN_BAUD_125
7	100	NTCAN_BAUD_100
10 *2)	83.3	-
8	66.6	-
9	50	NTCAN_BAUD_50
A	33.3	-
B	20	NTCAN_BAUD_20
C	12.5	-
D	10	NTCAN_BAUD_10

*1) The configuration of the predefined bit rates complies to the CiA (CAN in Automation e.V.) recommendations.

*2) Available not on every hardware platform..

Table 4.3.3: Pre-defined bit rate table

In order to force the hardware to leave the CAN bus and return to the state after boot up the special constant NTCAN_NO_BAUDRATE can be used as argument for the parameter *baud*. The support of this feature is hardware/firmware dependent (see chapter 3.10)

In order to initiate the auto baud detection the special constant NTCAN_AUTOBAUD can be used as argument for the parameter *baud*. The support of this feature is hardware/firmware dependent (see chapter 3.10). The detection process is performed asynchronously by the device driver. The auto baud rate detection is either cancelled by the driver if a valid baud rate is detected, which is indicated to the application with a baud rate change event (see page 70), or can be explicitly cancelled by calling *canSetBaudrate()* with another value for the parameter *baud*.

If the *UBR* flag is set to '1' and the 'User Bit Rate Numerical' flag *(UBRN*) is set to '0' the bits 0...15 are used to configure the bit rate register of the CAN controller directly with the given values. The constant `NTCAN_USER_BAUDRATE` is defined for the *UBR* flag.

The direct definition of bit rate register is very hardware dependent (CAN controller, clock frequency). See appendix 'Bit Timing Value Examples'.

If the *UBR* flag is '0' and the *UBRN* flag is set to '1', the bits 0...23 represent the baud rate in bit/s as numerical value.
The constant `NTCAN_USER_BAUDRATE_NUM` is defined for the UBRN flag.

UBR and UBRN obviously can not be set at the same time!

Note:	When using UBRN the BTR values are generated with a heuristic and might differ from the values in the baud rate table.

If the 'Listen Only Mode' (*LOM*) flag is set to '0' the CAN controller operates in normal active mode with this bit rate which means messages can be received and transmitted. If the LOM flag is set to '1' the CAN controller operates in the listen only mode (see page 25 for details) with this bit rate and can only receive messages. The constant `NTCAN_LISTEN_ONLY_MODE` is defined for the *LOM* flag.

Return: `NTCAN_SUCCESS` is returned after this function has been executed successfully, otherwise one of the error codes described from page 73 on is returned.

Header: `ntcan.h`

canGetBaudrate()

Name: **canGetBaudrate()** - determining the baud rate configured

Synopsis:

```
NTCAN_RESULT canGetBaudrate
        (
        NTCAN_HANDLE    handle,         /* CAN handle     */
        uint32_t        *baud           /* baud rate returned */
        )
```

Description: By means of this function the configured bit rate can be requested. The same baud rate values and formats are returned as described for *canSetBaudrate*. The returned value can either correspond to one of the 16 values of the baud rate table or to the contents of the bit-timing register *BTR0* and *BTR1*.
Furthermore you can determine whether the board has been initialized at all:
If the value `NTCAN_NO_BAUDRATE` is returned in **baud*, the CAN interface is not initialised.

Return: `NTCAN_SUCCESS` is returned after the function has been executed successfully, otherwise one of the error codes as described from page 73 on is returned.

Header: `ntcan.h`

canIdAdd()

Name: **canIdAdd()** - Selection of an Rx identifier or an event ID for the reception of data

Synopsis:

```
NTCAN_RESULT canIdAdd
        (
        NTCAN_HANDLE    handle,     /* read-handle */
        int32_t         id          /* Rx CAN identifier or event ID */
        )
```

Description: This function adds an Rx identifier to the handle filtermask. If no Rx identifier is enabled, no data will be stored in the Rx-FIFO and a call on *canRead()* or *canTake()* will return with an error!

The identifier selection is done by software. The filter is realized in the software driver for passive CAN modules and is realized by the local firmware of the module for intelligent CAN modules.
The number of IDs to be enabled depends on the application. If 11-bit IDs are used, all 2047 IDs can be enabled without loss of performance.

The same ID can be enabled for up to 15 different handles per physical CAN node.

In case of internal transmit-FIFO overrun *canIdAdd()* will return with an error (see error message `NTCAN_CONTR_BUSY`).

An ID-value is evaluated as 29-bit identifier, if bit 29 is set to '1' (identifier value in bit 0...28). The CAN module can receive all messages with 29-bit Rx, as soon as a single 29-bit-Rx identifier has been defined.

id = 0...07FF (hex) (11-bit-Rx identifier)
0...2047 (dec)

id = 0x20000000...0x3FFFFFFF (hex) (29-bit-Rx identifier)

id = *event-id* (in case of an event ID)

event-id = 0x40000000 (hex) ... 0x400000FF (hex)
`NTCAN_EV_BASE` ... `NTCAN_EV_LAST`

> **Note:** An overview of combinations of CAN modules and operating systems which support 29-bit-identifiers can be found on page 28.
> For further information about the mask configuration see page 23.

Return: After the function has successfully been executed `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

canIdDelete()

Name: **canIdDelete()** - Deleting an Rx identifier or an event ID

Synopsis:

```
NTCAN_RESULT canIdDelete
        (
        NTCAN_HANDLE    handle,     /* CAN handle */
        int32_t         id          /* Rx identifier or event ID */
        )
```

Description: This function deletes the selected Rx identifier from the message filter of the handle.

id = 0...07FF (hex) (11-bit identifier)
0...2047 (dec)

id = 0x20000000...0x3FFF.FFFF (hex) (29-bit identifier)

id = *event-id* (in case of an event ID)

event-id = 0x40000000 (hex) ... 0x400000FF (hex)
NTCAN_EV_BASE ... NTCAN_EV_LAST

Return: After successfully executing NTCAN_SUCCESS is returned, otherwise an error code as described from page 73 on is returned.

Header: ntcan.h

Note: If a CAN ID is no longer configured in the filter mask of any handle, the data of this ID is no longer updated in the object mode buffer and is consistently marked as invalid again.

canIoctl()

Name: **canIoctl()** - Set/get hardware and driver options

Synopsis:

```
NTCAN_RESULT canIoctl
        (
        NTCAN_HANDLE    handle,     /* CAN handle */
        uint32_t        ulCmd,      /* In: command  */
        void            *pArg       /* In/Out: pointer to argument  */
        )
```

Description: This function is an universal entry to set and determine hardware and driver options. Type and structure of the in/output data pointed by *pArg*, depends on the control command *ulCmd*. The following overview shows the implemented control commands. If a command *ulCmd* does not need an argument, *pArg* has to be set to NULL.

General I/O-Control Commands:

NTCAN_IOCTL_FLUSH_RX_FIFO	Argument: -	-
Deletes all CAN messages actually stored in the handle's Rx-queue.		

NTCAN_IOCTL_GET_RX_MSG_COUNT	Argument: uint32_t	Out
The number of CAN messages actually stored in the handle's Rx-queue is stored as unsigned 32-bit value at the memory address defined by *pArg*.		

NTCAN_IOCTL_GET_RX_TIMEOUT	Argument: uint32_t	Out
The Rx-timeout defined for this handle is stored as unsigned 32-bit value in [ms] at the memory address defined by *pArg*. This value may differ from the value defined in *canOpen()* due to operating system specific necessary rounding.		

NTCAN_IOCTL_GET_TX_TIMEOUT	Argument: uint32_t	Out
The Tx-timeout defined for this handle is stored as unsigned 32-bit value in [ms] at the memory address defined by *pArg*. This value may differ from the value defined in *canOpen()* due to operating system specific necessary rounding.		

NTCAN_IOCTL_SET_20B_HND_FILTER	Argument: uint32_t	In
Configures a handle specific acceptance mask for the reception of 29-bit (extended) identifiers acc. to CAN 2.0B specification (see page 23).		

NTCAN_IOCTL_GET_SERIAL	Argument: uint32_t	Out
The hardware serial number of this CAN module is stored as unsigned 32-bit value at the memory address defined by *pArg*. (only, if supported by the module). Each of the two upper nibbles of the hex value is coding one of the leading letters of the production lot number: 0 => A 1 => B : F_h => P The following nibbles each represents one digit of the six-digit series number. e.g.: $1D012345_h$ => serial number = BN012345 If serial number reading is not supported, always the serial no. AA000000 is returned.		

NTCAN_IOCTL_SET_TX_TIMEOUT	Argument: uint32_t	In
The Tx-timeout configured for this handle is set to a value. The argument is the new Tx-timeout in ms as unsigned 32-bit value for this handle. The new value will not be valid until the next blocking write function is executed.		

NTCAN_IOCTL_SET_RX_TIMEOUT	Argument: uint32_t	In
The Rx-timeout configured for this handle is set to a value. The argument is the new Rx-timeout in ms as unsigned 32-bit value for this handle. The new value will not be valid until the next blocking read function is executed.		

NTCAN_IOCTL_ABORT_TX	Argument: -	-
An active blocking write operation using this handle is aborted but the handle is not closed.		

NTCAN_IOCTL_ABORT_RX	Argument: -	-
An active blocking read operation using this handle is aborted but the handle is not closed.		

NTCAN_IOCTL_GET_TIMESTAMP_FREQ	Argument: uint64_t	Out
The resolution of the timestamp counter in Hz is returned, if timestamps are supported by CAN hardware, device driver and operating system.		

NTCAN_IOCTL_GET_TIMESTAMP	Argument: uint64_t	Out
The value of the current timestamp is returned, if timestamps are supported by CAN hardware, device driver and operating system.		

Tx-Object Mode Commands

The function *canIoctl()* can be used to generate scheduled CAN Tx-messages:

NTCAN_IOCTL_TX_OBJ_CREATE	Argument: CMSG	In
Creates an object for a certain CAN-ID, which can be used for scheduling. You will be able to create ONE object per CAN-ID (11-Bit or 29-Bit) per physical CAN-net. For 29-Bit CAN-IDs you might be restricted by memory, obviously it will be impossible to create an object for every single 29-Bit CAN-ID.		

NTCAN_IOCTL_TX_OBJ_DESTROY	Argument: CMSG	In
Destroys a formerly created object (and therefore stops it's scheduling). The object is referenced by it's CAN-ID.		

NTCAN_IOCTL_TX_OBJ_UPDATE	Argument: CMSG	In
Provides a formerly created object with CAN-data. The object is referenced by it's CAN-ID.		

NTCAN_IOCTL_TX_OBJ_SCHEDULE	Argument: CSCHED	In

Configures the scheduling for a certain object. If there are several objects scheduled for the same time, the order of these 'schedule'-calls will determine the order of transmission.
This call can be used as long as scheduling is NOT started yet or stopped.
This function is implemented to reduce problems with transmission order.

```
struct CSCHED {
   int32_t  id;            /* CAN-ID references a formerly created object */
   int32_t  flags;         /* choose between absolute or relative start time */
   int64_t  timeStart;     /* Time of the first transmission (zero = now) */
   int64_t  timeInterval;  /* Interval of cyclic transmission  */
   int8_t   reserved[8];   /* reserved for future use  */
};
```

NTCAN_IOCTL_TX_OBJ_AUTOANSWER_ON	Argument: CMSG	In
Activates autoanswer for a certain object. This way this object will be sent as an answer to a CAN-frame with the related CAN-RTR-ID. Autoanswer can be activated alone or in addition to scheduling.		

NTCAN_IOCTL_TX_OBJ_AUTOANSWER_OFF	Argument: CMSG	In
Deactivates autoanswer for a certain object.		

NTCAN_IOCTL_TX_OBJ_SCHED_START	-	-
This activates all scheduling done with one CAN-handle. All scheduled frames will be transmitted one time, when 'timeStart' has passed and from then on repeatedly every time 'timeInterval' has passed until object is destroyed or scheduling is stopped. As long as scheduling is running, one can not call NTCAN_IOCTL_TX_OBJ_SCHED.		

NTCAN_IOCTL_TX_OBJ_SCHED_STOP	-	-
Deactivates scheduling. In order to reduce problems with transmission order caused by configuration changes, this includes deletion of all schedules of this CAN-handle. Created objects remain and keep their CAN data, but before calling NTCAN_IOCTL_TX_OBJ_SCHED_START again, new schedule needs to be created with NTCAN_IOCTL_TX_OBJ_SCHEDULE.		

Note: Scheduling resolution is below timestamp resolution (dependent on actual hardware and operating system).

Note: For further description of Tx-scheduling see chapter 'How to Use Scheduling' from page 21 on.

Return: After successfully executing NTCAN_SUCCESS is returned, otherwise an error code as described from page 73 on is returned.

Header: ntcan.h

Example: Example to determine the number of messages in the Rx-FIFO and clearing the Rx-FIFO:

```
/*
 *  This file contains an incomplete example of the NTCAN API
 *  manual. Compiling this file will cause warnings of uninitialized
 *  handle value !!!!
 */

#include <stdio.h>
#include "ntcan.h"

void incomplete_read_11_bit(void)
{
  CMSG         cmsg[100];
  NTCAN_RESULT status;
  int32_t      len, i, j;
  NTCAN_HANDLE handle;



  len=100;
  status=canRead(handle, cmsg, &len, NULL);
  if(status == NTCAN_SUCCESS)
    {
      for(i=0; i<len; i++)
     {
       printf("id=%03x len=%x data: ", cmsg[i].id, cmsg[i].len);
       for(j=0; j<cmsg[i].len; j++)
          printf("%02x ", cmsg[i].data[j]);
       printf("\n");
     }
    }
  else
    {
      printf("canRead returned %x\n", status);
    }
.
.
.
```

4.3.3 Receiving and Transmitting CAN messages

canTake()

Name: **canTake()** - Non-blocking reading of current Rx data

Synopsis:

```
NTCAN_RESULT canTake
        (
        NTCAN_HANDLE    handle,   /* handle */
        CMSG            *cmsg,    /* pointer to data buffer */
        int32_t         *len      /* when called:     out: size of the CMSG buffer */
                                  /* when returned:   in: number of read messages */
        )
```

Description: This function returns the messages currently received for this handle without blocking.

FIFO-Mode
If the handle was opened with *canOpen()* in FIFO-mode (default), CAN messages received for this handle according to the active configuration of the message filter are copied in their chronological order into the address space of the application defined by *cmsg*.

The maximum size of the CMSG buffer has to be defined in *len* when calling *canTake()*. The size has to be specified in units of CMSG structures (a structure contains 16 bytes, see page 35). After the transfer the number of copied CAN message structures is stored in *len*. If no data has been received, the returned value for *len* is '0'.

Attention:	Please observe the notes on reading CAN data with 29-bit identifiers. They are listed under function *canRead()*.

Object Mode
If the handle was opened with *canOpen()* in object mode, the ID-fields of the CMSG-structures have to be initialized first with the (11-bit) identifiers of the CAN messages required. The size of the CMSG-buffer has to be specified in *len* when calling *canTake()*.

After the function has returned the CMSG-structures receive the data of the CAN message received last of this identifier. If bit 5 (`NTCAN_NO_DATA`) is set in the *len*-field of the individual CMSG-structure, no data has yet been received for this identifier.

Return: After this function has successfully been executed `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

canRead()

Name: **canRead()** - Blocking reading of current Rx data

Synopsis:

```
NTCAN_RESULT canRead
        (
        NTCAN_HANDLE    handle,     /* handle */
        CMSG            *cmsg,      /* pointer to data buffer */
        int32_t         *len,       /* when called: out: size of the CMSG buffer*/
                                    /* after return:  in: number of read messages*/
        OVERLAPPED      *ovrlppd    /* NULL or overlapped structure */
        )
```

Description: This function returns the messages received currently for this handle. If no new messages were received since the last call, the call blocks until new messages arrive or the Rx-timeout configured with *canOpen()* when the handle was opened has been exceeded.

If the handle was opened with *canOpen()* in FIFO-mode (default), the CAN messages received for this handle according to the active configuration of the message filter are copied in their chronological order into the address space of the application defined by *cmsg*.

The size of the CMSG buffer provided by the application has to be defined in *len* when calling *canRead()*. The size has to be specified in multiples of CMSG structures (a structure contains 16 bytes, see page 35). After the successful return the number of received CAN messages is stored in *len* by the driver. If no data has been received, the returned value for *len* is '0'.

If the handle was opened in object mode, the call immediately returns an error message (`NTCAN_INVALID_PARAMETER`).

Only with Windows operating systems (excepted RTX):
If no overlapped functions are executed, the function returns immediately, if the Rx-FIFO of the handle already contains CAN messages.
Otherwise the function returns, if the timeout interval specified when calling the function has been passed, or as soon as new data has been received within this interval.

During overlapped operations the function *canRead()* can be called like an asynchronous Windows function. Doing this the function returns immediately, no matter if data has been received or not. The status of the transmission (e.g. timeout) can be determined later by using *canGetOverlappedResult()* (see page 66).
Please refer to the Windows SDK for further information about Windows overlapped structures.

All other operating systems (incl. RTX):
Here '0' must always be entered for *ovrlppd*.

29-bit identifiers
In order to be able to receive a message with a 29-bit identifier, a 2.0B identifier has to be activated before. This can be achieved for example for identifier '0' by calling the function *canIdAdd(handle, 0 | NTCAN_20B_BASE).*

The activation of just one CAN 2.0B identifier enables the reception of all CAN 2.0B identifiers on the according handle. This can be restricted afterwards using *canIoctl()* with NTCAN_IOCTL_SET_20B_HND_FILTER (see chapter 3.6).

Depending on your ID filter configuration, the following operation is required to distinguish between CAN 2.0A and CAN 2.0B identifiers:

```
if(cmsg.id & NTCAN_20B_BASE)
    {
    /* 29-Bit ID received */
    id = cmsg.id & (NTCAN_20B_BASE-1);
    }
else
    {
    /* 11-Bit ID received */
    id = cmsg.id;
    }
```

Return: After successfully executing NTCAN_SUCCESS is returned, otherwise an error code as described from page 73 on is returned.

Header: ntcan.h

Example: Example for read accesses (11-bit identifiers):

```
/*
 *     This file contains an incomplete example of the NTCAN API
 *  manual. Compiling this file will cause warnings of uninitialized
 *  handle value !!!!
 */

#include <stdio.h>
#include "ntcan.h"

void incomplete_read_11_bit(void)
{
  CMSG         cmsg[100];
  NTCAN_RESULT status;
  int32_t      len, i, j;
  NTCAN_HANDLE handle;



  len=100;
  status =canRead(handle, cmsg, &len, NULL);
  if(status == NTCAN_SUCCESS)
    {
      for(i=0; i<len; i++)
     {
       printf("id=%03x len=%x data: ", cmsg[i].id, cmsg[i].len);
       for(j=0; j<cmsg[i].len; j++)
          printf("%02x ", cmsg[i].data[j]);
       printf("\n");
     }
    }
  else
    {
      printf("canRead returned %x\n", status);
    }
.
.
.
```

canTakeT()

Name: **canTakeT()** - Non-blocking reading of current Rx data with timestamp

Synopsis:

```
NTCAN_RESULT canTakeT
        (
        NTCAN_HANDLE    handle,    /* handle */
        CMSG_T          *cmsg_t,   /* pointer to data buffer */
        int32_t         *len       /* when called:out: size of the CMSG_T buffer */
                                   /* when returned: in: number of read messages */
        )
```

Description: This function returns the timestamped messages received for this handle without blocking.

> **Note:** See chapter 'Matrix of Supported Operating Systems and Features' from page 28 on to check if your configuration supports timestamps.

FIFO-Mode
If the handle was opened with *canOpen()* in FIFO-mode (default), CAN messages received for this handle according to the active configuration of the message filter are copied in their chronological order into the address space of the application defined by *cmsg*_t.

The maximum size of the CMSG_T buffer has to be defined in *len* when calling *canTakeT()*. The size has to be specified in units of CMSG_T structures (a structure contains 24 bytes, see page 35). On return the number of received CAN message structures is stored in *len*. If no data has been received, the returned value for *len* is '0'.

> **Attention:** Please observe the notes on reading CAN data with 29-bit identifiers. They are listed under function *canRead()* at page 60.

Object Mode
If the handle was opened with *canOpen()* in object mode, the member *id* of the CMSG_T structures have to be initialized first with the (11-bit) identifiers of the requested CAN messages. The size of the CMSG_T-buffer has to be specified in *len* when calling *canTakeT()*.

After the function has returned the CMSG_T-structures are filled with the data of the CAN message received last with these identifiers. If bit 5 (`NTCAN_NO_DATA`) is set in the member *len* of the individual CMSG_T-structure, no data has yet been received for this identifier.

Return: After this function has successfully been executed `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

canReadT()

Name: **canReadT()** - Blocking reading of current Rx data with timestamps

Synopsis:

```
NTCAN_RESULT  canReadT
      (
      NTCAN_HANDLE  handle,     /* handle */
      CMSG_T        *cmsg_t,    /* pointer to data buffer */
      int32_t       *len,       /* when called: out: size of the CMSG_T buffer*/
                                /* after return: in: number of read messages*/
      OVERLAPPED    *ovrlppd    /* NULL or overlapped structure */
      )
```

Description: This function returns the timestamped messages received currently for this handle.

> **Note:** See chapter 'Matrix of Supported Operating Systems and Features' from page 28 on to check if your configuration supports timestamps.

If no new messages were received since the last call, the call blocks until new messages arrive or the Rx-timeout configured with *canOpen()* when the handle was opened has been exceeded.

If the handle was opened with *canOpen()* in FIFO-mode (default), the timestamped CAN messages received for this handle according to the active configuration of the message filter are copied in their chronological order into the address space of the application defined by *cmsg_t*.

The size of the CMSG_T buffer provided by the application has to be defined in *len* when calling *canReadT()*. The size has to be specified in multiples of CMSG_T structures (a structure contains 24 bytes, see page 35). After the successful return the number of received CAN messages is stored in *len* by the driver. If no data has been received, the returned value for *len* is '0'.

If the handle was opened in object mode, the call immediately returns an error message (`NTCAN_INVALID_PARAMETER`).

For details about the parameter *ovrlppd* and a distinction between 11-bit CAN messages and 29-bit CAN messages please refer to the documentation of *canRead()*.

Return: After successfully executing `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

canWrite()

Name: **canWrite()** - Blocking transmission of CAN messages

Synopsis:

```
NTCAN_RESULT canWrite
        (
        NTCAN_HANDLE    handle,     /* handle */
        CMSG            *cmsg,      /* pointer to data buffer */
        int32_t         *len,       /* out:  size of the CMSG buffer   */
                                    /* in:   number or transmitted messages */
        OVERLAPPED      *ovrlppd    /* NULL or overlapped structure */
        )
```

Description: This function initiates the transmission of ***len* CAN messages given in *cmsg*. The capacity of the CMSG buffer has to be defined as input parameter in *len* when calling *canWrite()*. The size has to be specified in multiples of CMSG structures (a structure contains 16 bytes, see page 35).

In case of 'NTCAN_SUCCESS' ***len* returns the number of successfully transmitted CAN frames.

If no overlapped functions are executed, the function returns after the transmission request is completed successfully, or if the timeout interval specified when calling the function has been passed.

The Tx-timeout interval is applied to each CAN message, not for the complete request, therefore the function might block the program for the number of messages multiplied by the timeout interval in worst case!

Only with Windows operating systems (excepted RTX):
At overlapped operations the function *canWrite()* can be called like an asynchronous Windows function. The function returns immediately regardless of whether the transmission request has been terminated successfully. The status of the transmission (e.g. timeout) can later be determined by using *canGetOverlappedResult()* (see page 66).
Please consult the Windows manual for further information about Windows overlapped structures.

All other operating systems (incl. RTX):
Here 'NULL' must always be entered for *ovrlppd*.

Return: After successfully executing NTCAN_SUCCESS is returned, otherwise an error code as described from page 73 on is returned.

Header: ntcan.h

canSend()

Name: **canSend()** - Non-blocking transmission of a message

Synopsis:

```
NTCAN_RESULT canSend
        (
        NTCAN_HANDLE    handle,   /* handle */
        CMSG            *cmsg,    /* pointer to data buffer */
        int32_t         *len      /* when called and after return: */
                                  /* out:  number of transmitted messages */
                                  /* in:   number of messages to be transmitted */
        )
```

Description: By means of this function the transmission of **len* messages starts from the memory structure defined by *cmsg*.
The capacity of the CMSG buffer has to be defined as input parameter in *len*. The size has to be specified in units of CMSG-structures (a structure contains 16 bytes, see page 35).

The function returns at once. In case of `NTCAN_SUCCESS` **len* returns the numbers of CAN frames successfully copied into the Tx queue.

Return: After successfully executing `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

canGetOverlappedResult()

Name: **canGetOverlappedResult()** - Determining the result of overlapped functions

Synopsis:

```
NTCAN_RESULT canGetOverlappedResult
        (
        NTCAN_HANDLE    handle,     /* handle */
        OVERLAPPED      *ovrlppd,   /* pointer to overlapped structure*/
        int32_t         *len,       /* size in multiples of CMSG- or EVMSG- messages */
        BOOL            bwait       /* selection of the return mode*/
        )
```

Description: This function is only implemented **in windows operating systems** not in other operating systems and not in the windows realtime environment RTX. It has to be called in order to evaluate the result of overlapped operations (i.e. *canRead()* with *ovrlppd* != NULL), if the according function was terminated with the result code `NTCAN_IO_PENDING`.

The pointer to the desired overlapped structure has to be defined in **ovrlppd*.

**len* returns the number of CAN messages (CMSG) or event messages (EVMSG).

The parameter *bwait* selects the return mode:

bwait = TRUE *canGetOverlappedResult()* returns, if the according I/O job is finished.

bwait = FALSE *canGetOverlappedResult()* returns at once, even if the I/O job has not yet been finished. The function returns `NTCAN_IO_INCOMPLETE`.

Return: After successfully executing `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

4.3.4 Receiving and Transmitting Events

At the moment only two events for *reading* the error status are defined (`NTCAN_EV_CAN_ERROR` and `NTCAN_EV_BAUD_CHANGE`). Principally events with write access are also possible, therefore the function calls will be described here, too.

Note: *canReadEvent()* and *canSendEvent()* calls are deprecated. Please use *canRead()* and *canSend()* instead!
See usage example at page 26 in chapter 'Event Interface'.

canReadEvent()

Name: **canReadEvent()** - Reading input variables

Synopsis:

```
NTCAN_RESULT canReadEvent
        (
        NTCAN_HANDLE    handle,     /* handle */
        EVMSG           *evmsg,     /* pointer to event message buffer */
        OVERLAPPED      *ovrlppd    /* NULL or overlapped structure */
        )
```

Description: By means of this function the error status can be read via the event `NTCAN_EV_CAN_ERROR` for instance.

If no overlapped functions are executed, the function returns if no event occurs or if the timeout interval specified when calling the function has been passed.

Only with Windows operating systems (excepted RTX):
As overlapped operation the function *canReadEvent()* can be used like an asynchronous Windows function. Doing this the function returns immediately, no matter if an event occurred or not. The event can be evaluated later by means of *canGetOverlappedResult()* (see page 66).
Please consult the Windows manual for further information about Windows overlapped structures.

All other operating systems (incl. RTX) and esd-CAN-Bluetooth module:
Here '0' must always be entered for *ovrlppd*.

Return: After successfully executing `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

canSendEvent()

Name: **canSendEvent()** - Posting events into the CAN driver

Synopsis:

```
NTCAN_RESULT  canSendEvent
        (
        NTCAN_HANDLE    handle,     /* handle */
        EVMSG           *evmsg      /* pointer to event message buffer */
        )
```

Description: This function might be used for setting output variables or posting events.
It can also be used for triggering some special events to prepare them for reading back the input information by *canReadEvent()*.

This function is for future applications and can not be used at the moment!

The function always returns immediately.

Return: After successfully executing `NTCAN_SUCCESS` is returned, otherwise an error code as described from page 73 on is returned.

Header: `ntcan.h`

CAN Events

NTCAN_EV_CAN_ERROR

Event-ID: NTCAN_EV_CAN_ERROR
Event no.: NTCAN_EV_BASE + $0
Reading the event: *canRead(), (canReadEvent())*
Transmitting the event: not possible
Data length: 6 bytes
Data format: evdata.s[0]

Offset Address:	+0	+1	+2	+3	+4	+5
Assignment:	reserved	*ERROR*	reserved	*LOST_ MESSAGE*	reserved	*LOST_FIFO*

Table 4.3.4: Assignment of the event NTCAN_EV_CAN_ERROR

The event is produced, if any of the error parameters, covered by this event, change.

Parameter value ranges:

ERROR ... 0x00 - CAN controller status is *error active* (NTCAN_BUSSTATE_OK)
0x40 - CAN controller status is *error warn* (NTCAN_BUSSTATE_WARN)
one of the error counters (Rx or Tx) exceeds 95.
0x80 - CAN controller status is *error passive* (NTCAN_BUSSTATE_ERRPASSIVE)
one of the error counters (Rx or Tx) exceeds 127.
0xC0 - CAN controller status is *bus off* (NTCAN_BUSSTATE_BUSOFF)
one of the error counters (Rx or Tx) exceeds 255

LOST_MESSAGE ... counter for lost messages of the CAN controller
This counter is set by an error output of the CAN controller. It shows the number of lost CAN frames (receive or transmit messages).

LOST_FIFO ... counter for lost messages of the FIFO
This counter is incremented, if messages are lost because of a FIFO overrun (FIFO full).

NTCAN_EV_BAUD_CHANGE

Event-ID:	NTCAN_EV_BAUD_CHANGE
Event no.:	NTCAN_EV_BASE + $1
Reading the event:	*canRead(), (canReadEvent())*
Transmitting the event:	not possible
Data length:	4 bytes or 8 bytes
Data format:	evdata.l[0]

Offset address:	+0	+1	+2	+3	+4	+5	+6	+7
Assignment:	*BAUDRATE*				*BAUDRATE_NUM*			

Table 4.3.5: Assignment of the event NTCAN_EV_BAUD_CHANGE

The event is produced, if the baud rate of the physical CAN channel is changed to a value which differs from the current configuration. The value of this event is the baud rate as described for *canSetBaudrate()*.

If the data length is 8 bytes, the bytes 4 to 7 contain the baud rate as numerical value in bit/s.

NTCAN_EV_CAN_ERROR_EXT

Event ID:	NTCAN_EV_CAN_ERROR_EXT
Event no.:	NTCAN_EV_BASE + $2
Reading the event:	*canRead()* or *(canReadEvent())*
Transmitting the event:	not possible
Data length:	4 bytes
Data format:	evdata.l[0]

This event is only supported by drivers which support 'extended error information' as shown in the 'Matrix of Supported Operating Systems and Features' (page 29).
This event is generated, if the error information changes.

Offset address:	+0	+1	+2	+3
Assignment:	Stat	ECC	Rx-Cnt	Tx-Cnt

Table 4.3.6: Example of the event NTCAN_EV_CAN_ERROR_EXT

4.3.5 Retrieving Status and Information

canStatus()

Name: **canStatus()** - Returns information about hardware and software

Synopsis:

```
NTCAN_RESULT canStatus
        (
        NTCAN_HANDLE    handle,   /* status handle */
        CAN_IF_STATUS   *cstat    /* pointer to status structure */
        )
```

Description: This function provides information about hardware and software of the CAN interface assigned to this CAN handle. The handle is assigned to a CAN interface via the logical network number when opened when calling *canOpen()*.

Return: After it has been successfully executed `NTCAN_SUCCESS` is returned and the status information is written into the memory area pointed at by *cstat*. In case of an error one of the error codes shown on page 73 is returned.

Header: `ntcan.h`

5. Return Codes

All API-calls return a status which starts with the prefix 'NTCAN_' which should always be evaluated. If the call returns an error code, the content of all returned values referenced by pointers are undefined and must not be evaluated by the application.

The constants for the returned values are defined in the header file ntcan.h. It is advisable to refer only to these constants for reasons of portability, because each operating system has got it's own 'number area' for the numerical values of errors. Therefore different numerical values are used for the same return status with different operating systems. Furthermore a few constants for errors which are not CAN specific and are usually generated autonomously by the operating system (such as NTCAN_INVALID_HANDLE) are mapped to already existing error constants of the operating system to increase the portability.

Below all returned values are listed in a table. The values are divided into the error categories *Successful*, *Warning* and *Error*. Furthermore a description of the cause of error and the solution possibilities as well as a list of API-calls in which this problem might arise is listed.

Status codes which are listed in the header file but are not described here, are not returned by the driver anymore and have only not been removed to ensure the compatibility of existing source code.

5.1 General Return Codes

NTCAN_SUCCESS

Successful execution of a call.

Category	Successful
Cause	The call was terminated without errors. The content of all returned values referenced by pointers are valid and must only be evaluated by the application.
Function	All functions

NTCAN_CONTR_BUSY

The capacity of the internal transmit-FIFO has been exceeded.

Category	Error/Warning
Cause	The capacity of the internal transmit-FIFO is too small to receive further messages/commands.
Solution	Repeat the unsuccessful call after a short period.
Function	*canSend(), canSendEvent(), canIdAdd(), canIdDelete()*

NTCAN_CONTR_OFF_BUS

Transmission error.

Category	Error
Cause	The CAN controller has changed into *OFF BUS* status during a blocking transmit operation, because too many CAN error frames have been received.
Solution	Repeat the call after a short period, because the driver automatically tries to recover from the error situation. If the error still occurs, you should check, whether the CAN bus is correctly wired and all bus devices transmit with the same baud rate.
Function	*canWrite()*

NTCAN_CONTR_WARN

Reception error.

Category	Error
Cause	The CAN controller has changed into *Error Passive* status during a transmit operation, because too many CAN error frames have been received.
Solution	Repeat the call after a short period, because the driver automatically tries to recover from the error situation. If the error still occurs, you should check, whether the CAN bus is wired correctly and all bus devices transmit with the same baud rate.
Function	*canWrite()*

NTCAN_ID_ALREADY_ENABLED

The CAN-ID for this handle has already been activated.

Category	Warning
Cause	The CAN identifier for this handle has already been activated.
Solution	Activate each CAN-ID only once per handle. If a CAN 2.0B ID is activated, all other CAN 2.0B IDs are regarded as being activated as well.
Function	*canIdAdd()*

NTCAN_ID_NOT_ENABLED

The CAN-ID has not been activated for this handle.

Category	Warning
Cause	The CAN-ID has not been activated for this handle.
Solution	Deactivate each CAN-ID only once per handle. If a CAN 2.0B ID is deactivated, all other CAN 2.0B IDs are regarded as being deactivated as well.
Function	*canIdDelete()*

NTCAN_INSUFFICIENT_RESOURCES

Insufficient internal resources.

Category	Error
Cause	The operation could not be completed because of insufficient internal resources.
Solution	• If the error occurs when calling *canOpen()*, the handle queue size should be decreased. • If the error occurs when calling *canIdAdd()*, this CAN-ID has already been activated for too many other handles.
Function	*canOpen(), canIdAdd()*

NTCAN_INVALID_DRIVER

Driver and NTCAN library are not compatible.

Category	Error
Cause	The version of the NTCAN library requires a more recent driver version.
Solution	Use a more recent driver version.
Function	*canOpen()*

NTCAN_INVALID_FIRMWARE

Driver and firmware are incompatible.

Category	Error
Cause	The version of the device driver requires a more recent firmware version.
Solution	Update the firmware of the active CAN board.
Function	*canOpen()*

NTCAN_INVALID_HANDLE

Invalid CAN handle.

Category	Error
Cause	An invalid handle was passed to a function call.
Solution	• Check whether the handle was correctly opened with *canOpen()*. • Check whether the handle was not closed previously with *canClose()*. • Check if *canReadEvent()* is called with a handle that has enabled IDs which don't belong to the ID range of events.
Function	All functions except *canOpen()*

NTCAN_INVALID_HARDWARE

Driver and hardware are incompatible.

Category	Error
Cause	The version of the device driver is incompatible to the hardware.
Solution	Use another driver version.
Function	*canOpen()*

NTCAN_INVALID_PARAMETER

Invalid parameter.

Category	Error
Cause	An invalid parameter was passed to the library.
Solution	• Check all arguments for this call for validity. • Call *canRead()* with a handle which was opened for the object mode.
Function	All functions.

NTCAN_IO_INCOMPLETE

Operation has not yet been terminated (Win32 only).

Category	Error/Warning
Cause	The function *canGetOverlappedResult()* was called with FALSE for parameter *bWait* and the asynchronous operation has not been completed. See Win32 platform SDK-help about *canGetOverlappedResult()* for further details.
Solution	See Win32 platform SDK-help about *canGetOverlappedResult()*.
Function	*canGetOverlappedResult()*

NTCAN_IO_PENDING

Operation has not been terminated (Win32 only).

Category	Warning
Cause	An asynchronous read or write operation with valid overlapped structure has not been completed.
Solution	See Win32 platform SDK about 'Asynchronous Input and Output' for further details.
Function	*canRead(), canReadT(), canWrite(), canReadEvent()*

NTCAN_NET_NOT_FOUND

Invalid logical network number.

Category	Error
Cause	The logical network number specified when opening *canOpen()* does not exist.
Solution	• Check whether the network number has been assigned. • Check whether the driver to which this network number should be assigned was started correctly.
Function	*canOpen()*

NTCAN_NO_ID_ENABLED

Read handle without any enabled CAN identifier.

Category	Warning
Cause	For this handle *canIdAdd()* has not been called.
Solution	*canIdAdd()* has to be called before *canRead()* is called.
Function	*canRead(), canReadT(), canTake(), canTakeT()*

NTCAN_OPERATION_ABORTED

Abortion of a blocking transmit/receive operation.

Category	Warning/Error
Cause	A blocking transmit or receive operation was aborted, because another thread called *canIoctl()* with the argument `NTCAN_IOCTL_ABORT_TX` or `NTCAN_IOCTL_ABORT_RX` for this handle.
Solution	If the procedure was unintended by the application, check why I/O operations are being aborted.
Caveats	It isn't possible with all operating systems supported by the NTCAN-API to distinguish between the abort and the forced close case, described below. These operating systems will return this error code even if the reason for the abort was a forced close of the handle.
Function	*canRead(), canReadT(), canWrite(), canReadEvent()*

NTCAN_HANDLE_FORCED_CLOSE

Abortion of a blocking transmit/receive operation.

Category	Warning/Error
Cause	A blocking transmit or receive operation was cancelled, because another thread called *canClose()* for this handle or the driver was terminated. If a call returns with this error code the handle is no longer valid.
Solution	If the procedure was unintended by the application, check why handles are being closed.
Function	*canRead(), canReadT() canWrite(), canReadEvent()*

NTCAN_PENDING_READ

Receive operation could not be executed.

Category	Warning/Error
Cause	No receive operation was initiated, because the handle is already being used by another thread for a receive operation.
Solution	• Use operating system specific synchronization mechanism to avoid that another thread uses the handle simultaneously for reception. • Use threads with different handles.
Function	*canRead(), canReadT(), canTake(), canTakeT()*, *canReadEvent()*

NTCAN_PENDING_WRITE

Transmit operation could not be executed.

Category	Warning/Error
Cause	No transmit operation was initiated, because the handle is already being used by another thread for a transmit operation.
Solution	• Use operating system specific synchronization mechanism to avoid that another thread uses the handle simultaneously for transmission. • Use threads with different handles.
Function	*canWrite(), canSend(), canSendEvent()*

NTCAN_RX_TIMEOUT

Timeout event for blocking receive operation.

Category	Warning
Cause	No data has been received within the Rx-timeout declared in *canOpen()*.
Solution	• Increase Rx-timeout in *canOpen()*. • Ensure that data is transmitted by other CAN devices on the expected CAN-IDs.
Function	*canRead(), canReadT(), canReadEvent()*

NTCAN_TX_ERROR

Timeout event for blocking receive operation.

Category	Error
Cause	The message could not be transmitted within an internal timeout period (typically 1000 ms). This error is replaced by `NTCAN_TX_TIMEOUT`, if a shorter period was selected for the Tx-timeout in *canOpen()*.
Solution	• Check wiring. • Check baud rate.
Function	*canWrite()*

NTCAN_TX_TIMEOUT

Timeout event for blocking transmit operation.

Category	Warning/Error
Cause	The data could not be transmitted within the Tx-timeout declared in *canOpen()*.
Solution	• Increase Tx-timeout in *canOpen()*. • Check wiring. • Check baud rate.
Function	*canWrite()*

NTCAN_WRONG_DEVICE_STATE

The actual device state prevents I/O-operations (Win32 only).

Category	Warning/Error
Cause	The system state is changing to the sleep mode.
Solution	Prevent state changing to the sleep mode.
Function	all functions

NTCAN_NOT_IMPLEMENTED

Command for *canIoctl()* is not implemented.

Category	Error
Cause	The argument *ulCommand* of *canIoctl()* is not implemented or supported by the library, device driver or hardware.
Solution	• Check the *ulCommand* parameter for validity. • If the argument is valid, check if a newer driver/library is available which supports this command.
Function	*canIoctl()*

NTCAN_NOT_SUPPORTED

The argument of the call is valid but not supported.

Category	Error
Cause	The argument of the call is valid but the requested property is not supported because of hardware and/or firmware limitations.
Solution	• Check all arguments for this call for validity. • If the arguments are valid, check if a newer or different firmware for this hardware is available which supports the requested property. • If the requested property can not be supported due to hardware constraints, switch to a different esd CAN board which supports this property. See chapter 'Matrix of Supported Operating Systems and Features' from page 28 on to check, which features are supported. esd is always endeavoured to match the current operating system developments and to keep the CAN drivers up to date. Please visit www.esd-electronics.com to check for the latest software version.
Function	*canIoctl(), canIdAdd(), canIdDelete(), canSetBaudrate()*

5.2 Special Return Values of the EtherCAN Driver

NTCAN_SOCK_CONN_TIMEOUT

Only applicable for EtherCAN module under Linux and Windows:
Within the timeout time `ConnTimeout[x]`, defined under Linux in `/etc/esd-plugin` no network connection can be established.

Category	Error
Cause	no network connection established; connection runtime to exceeded
Solution	• check network connection (ping) • increase timeout • faster connection
Function	*canOpen()*

NTCAN_SOCK_CMD_TIMEOUT

Only applicable for EtherCAN module under Linux and Windows:
TCP-socket timeout while sending a special command to EtherCAN server (under Linux parameter `CmdTimeout[x]` in `/etc/esd-plugin`).

Category	Error
Cause	runtime of TCP/IP-packages exceeded
Solution	• increase timeout • faster connection
Function	all functions

NTCAN_SOCK_HOST_NOT_FOUND

Only applicable for EtherCAN module under Linux and Windows:
Resolving hostname specified by `PeerName[x]` in `/etc/esd-plugin` (under Linux) failed.

Category	Error
Cause	wrong name, incorrect name server configuration, ...
Solution	• use correct name • configure name server correctly
Function	*canOpen()*

6. Example Programs

6.1 Example Program: Receiving Messages

```
#include <stdio.h>
#include <ntcan.h>

/*
 * This example demonstrates how the NTCAN-API can be used to open a handle,
 * set a baudrate and wait for reception of a CAN frame with an
 * identifier that has been previously enabled for this handle.
 * Finally all proper cleanup operations are performed
 */
int example_rx(void)
{
    int          net=0;           /* logical net number (here: 0) */
    uint32_t     mode=0;          /* mode bits for canOpen */
    int32_t      txqueuesize=8;   /* maximum number of messages to transmit */
    int32_t      rxqueuesize=8;   /* maximum number of messages to receive */
    int32_t      txtimeout=100;   /* timeout for transmit in ms */
    int32_t      rxtimeout=10000; /* timeout for receiving data in ms */
    NTCAN_HANDLE rxhandle;        /* can handle returned by canOpen() */
    NTCAN_RESULT retvalue;        /* return values of NTCAN API calls */
    uint32_t     baud=2;          /* configured CAN baudrate (here: 500 kBit/s.) */
    CMSG         cmsg[8];         /* buffer for can messages */
    int          i;               /* loop counter */
    int32_t      len;             /* Buffer size in # of messages for canRead() */

    /* ################################################################ */

    retvalue = canOpen(net,
                       mode,
                       txqueuesize,
                       rxqueuesize,
                       txtimeout,
                       rxtimeout,
                       &rxhandle);

    if (retvalue != NTCAN_SUCCESS)
    {
        printf("canOpen() failed with error %d!\n", retvalue);
        return(-1);
    }

    printf("function canOpen() returned OK !\n");

    /* ################################################################ */

    retvalue = canSetBaudrate( rxhandle, baud);

    if (retvalue != 0)
    {
        printf("canSetBaudrate() failed with error %d!\n", retvalue);
        canClose(rxhandle);
        return(-1);
    }

    printf("function canSetBaudrate() returned OK !\n");

    /* ################################################################ */

    retvalue = canIdAdd(rxhandle, 0); /* Enable CAN-ID 0 */

    if (retvalue != NTCAN_SUCCESS)
    {
        printf("canIdAdd() failed with error %d!\n", retvalue);
        canClose(rxhandle);
        return(-1);
```

```
    }

    printf("function canIdAdd() returned OK !\n");

    /* ################################################################ */

    do {
       /*
        * Set max numbers of messages that can be returned with
        * one canRead() call according to buffer size
        */
        len = 8;

        retvalue = canRead(rxhandle, &cmsg[0], &len, NULL);

        if (retvalue == NTCAN_RX_TIMEOUT)
        {
            printf("canRead() returned timeout\n");
            continue;
        }
        else if(retvalue != NTCAN_SUCCESS)
        {
            printf("canRead() failed with error %d!\n", retvalue);
        }
        else
        {
            printf("function canRead() returned OK !\n");
            printf("Id of received message :%x!\n", cmsg[0].id);
            printf("Len of received message :%x!\n", (cmsg[0].len & 0x0f));
            printf("Rtr of received message :%x!\n", ((cmsg[0].len & 0x10)>>4));
            for (i=0;i<(cmsg[0].len & 0x0f);i++)
                printf("Byte %d of received message :%x!\n", i, cmsg[0].data[i]);
        }

        break;
    } while(1);

    /* ################################################################ */

    retvalue = canIdDelete( rxhandle, 0);

    if (retvalue != NTCAN_SUCCESS)
        printf("canIdDelete() failed with error %d!\n", retvalue);

    printf("function canIdDelete() returned OK !\n");
    /* ################################################################ */

    retvalue = canClose (rxhandle);

    if (retvalue != NTCAN_SUCCESS)
        printf("canClose() failed with error %d!\n", retvalue);

    printf("function canClose returned OK !\n");

    /* ################################################################ */

    return(0);
}
```

6.2 Example Program: Transmitting Messages

```
#include <stdio.h>
#include <ntcan.h>

/*
 * This example demonstrates how the NTCAN-API can be used to open a handle,
 * set a baudrate and transmitting a CAN frame.
 * Finally all proper cleanup operations are performed
*/
int example_tx(void)
{
    int          net=0;          /* logical net number (here: 0) */
    uint32_t     mode=0;         /* mode used for canOpen() */
    int32_t      txqueuesize=8;  /* size of transmit queue */
    int32_t      rxqueuesize=8;  /* size of receive queue */
    int32_t      txtimeout=100;  /* timeout for transmit operations in ms */
    int32_t      rxtimeout=1000; /* timeout for receive operations in ms */
    NTCAN_HANDLE txhandle;       /* can handle returned by canOpen() */
    NTCAN_RESULT retvalue;       /* return values of NTCAN API calls */
    uint32_t     baud=2;         /* configured CAN baudrate (here: 500 kBit/s.) */
    CMSG         cmsg[8];        /* can message buffer */
    int          rtr=0;          /* rtr bit */
    int          i;              /* loop counter */
    int32_t      len;            /* # of CAN messages */

    /* ################################################################ */

    retvalue = canOpen(net,
                       mode,
                       txqueuesize,
                       rxqueuesize,
                       txtimeout,
                       rxtimeout,
                       &txhandle);

    if (retvalue != NTCAN_SUCCESS)
    {
        printf("canOpen() failed with error %d!\n", retvalue);
        return(-1);
    }

    printf("function canOpen() returned OK !\n");

    /* ################################################################ */

    retvalue = canSetBaudrate(txhandle, baud);

    if (retvalue != 0)
    {
        printf("canSetBaudrate() failed with error %d!\n", retvalue);
        canClose(txhandle);
        return(-1);
    }

    printf("function canSetBaudrate() returned OK !\n");

    /* ################################################################ */

    /*
    * Initialize the first message in buffer to CAN id = 0, len = 3
    * and data0 - data2 = 0,1,2
    */
    cmsg[0].id=0x00;
    cmsg[0].len=0x03;
    cmsg[0].len |= cmsg[0].len + (rtr<<4);
    for (i=0;i<3;i++)
        cmsg[0].data[i] = i;
```

```
    len=1;            /* Number of valid messages in cmsg buffer*/

    retvalue =  canWrite(txhandle, &cmsg[0], &len, NULL);

    if (retvalue != NTCAN_SUCCESS)
        printf("canWrite failed() with error %d!\n", retvalue);
    else
        printf("function canWrite() returned OK !\n");

    /* ############################################################## */

    retvalue = canClose (txhandle);

    if (retvalue != NTCAN_SUCCESS)
        printf("canClose failed with error %d!\n", retvalue);
    else
        printf("function canClose() returned OK !\n");

    /* ############################################################## */

    return(0);
}
```

7. Test Programs

7.1 cantest for the Command Line as Example Program

The software package includes the source code of the test program `cantest`. This program is a good example for the use of the CAN API functions.

`cantest` runs under all supported operating systems.

The functionality of the test program is described in this chapter. The source code can be taken from the data carrier (disk or CD-ROM). It is not printed in this document.

7.1.1 Functional Description

If `cantest` is called without parameters, the program searches for the installed esd CAN boards in the system and displays board parameters like software revision numbers and board status. Additionally a list with the available functions and the input syntax is displayed.

The following figure shows an example display generated by `cantest`:

```
CAN Test Rev 2.7.2  -- (c) 1997-2005 electronic system design gmbh

Available CAN-Devices:
Net   0: ID=CAN_PCI331 Dll=4.0.01 Driver=2.5.00 Firmware=0.C.20 Serial=n/a
         Hardware=1.1.00 Baudrate=00000002 Status=00000000 Features=0030
         Timestamp=00000ebb2a372619 TimestampFreq=501000000 Hz
Net   1: ID=CAN_PCI331 Dll=4.0.01 Driver=2.5.00 Firmware=0.C.20 Serial=n/a
         Hardware=1.1.00 Baudrate=7fffffff Status=00000000 Features=0030
         Timestamp=00000ebb2a5b6d6c TimestampFreq=501000000 Hz

Syntax: cantest test-Nr [net id-1st id-last count
        txbuf rxbuf txtout rxtout baud testcount data0 data1 ...]
Test  0:  canSend()
Test  1:  canWrite()
Test 51:  canWrite() with incrementing ids
Test  2:  canTake()
Test 12:  canTake() with time-measurement for 10000 can-frames
Test 22:  canTake() in Object-Mode
Test  3:  canRead()
Test 13:  canRead() with time-measurement for 10000 can-frames
Test 23:  canReadT()
Test  4:  canReadEvent()
Test  5:  canSendEvent()
Test  6:  Overlapped-canRead()
Test  7:  Overlapped-canWrite()
Test  8:  Create auto RTR object
Test  9:  Wait for RTR reply
Test 19:  Wait for RTR reply without text-output
```

Fig. 7.1.1: Display of the board status and input syntax generated with `cantest`

The example shows that the two physical CAN nets with the logical network numbers 0 and 1 are available on the hardware CAN-PCI/331 in the system. After that, the input syntax is displayed.

Each parameter has a default value, which is used if the parameter is not set. The parameters must always be set in the order as they are displayed, e.g. if only the value of the `testcount` shall be changed, all previous parameters must be set, too. If the parameters following `testcount` are not entered, they will be assumed with their default values.

7.1.2 Special Features of VxWorks Implementation

The following differences from the general description have to be observed for the VxWorks implementation of cantest.

- The notation of cantest is as follows for the program call: `canTest`

- Under VxWorks the parameters have to be entered in apostrophes ("...").

7.1.3 Description of Displayed Board Parameters

After the net number a number of board parameters is displayed:

`ID`	CAN module identifier name.
`Dll`	Revision number of DLL. (NTCAN Library)
`Driver`	Revision number of CAN driver.
`Firmware`	Revision number of local firmware (if applicable).
`Serial`	Hardware serial number of the CAN module (if supported by the module).
`Hardware`	Hardware revision number.
`Baudrate`	Current baud rate. `Baudrate` = 0 ... E_h => bit rate index as described at page 48. `Baudrate` = $7FFFFFFF_h$ => no bit rate index defined
`Status`	Hardware status of the CAN module (if supported by the module).
`Features`	Supported properties of hardware and/or device driver. Returns the value of the parameter *feature* of data structure CAN_IF_STATUS as described at page 42. Bit value = '1' => according feature is available
`Timestamp`	Current timestamp value captured at the moment `cantest` is called (if supported by the module).
`TimestampFreq`	Frequency of the timestamp counter (if supported by the module).

7.1.4 Parameter Description

All parameters, apart from the identifiers and baud rate, have to be specified as decimal values. For identifiers decimal as well as hexadecimal values (prefix '0x') are valid.

`test-Nr` The available test numbers are listed above. Test number 5 (*canSendEvent()*) has no function at the moment. Test numbers 6 and 7(Overlapped) can only be executed in Windows operating systems (except RTX).

`net` Logical number of the CAN net.

`id-1st,id-last`

read: By means of these two limit values an identifier area can be defined in which messages are to be received. If only messages of one ID are to be evaluated, the same ID value has to be set for both parameters.

write: The transmit identifier is set for `id-1st`. `id-last` is ignored in write operations.

Default: `id-1st` = 0
`id-last` = 0

Number format: Values without further specifications are evaluated as decimal values. The specification '0x' in front of a value marks a hexadecimal value.

Value range for 11-bit identifiers: 0...2047 (dec)
0x0...0x7FF (hex)
example: 0x3F = 63 (dec)

Value range for 29-bit identifiers: 0x20000000...0x3FFFFFFF
example: 0x2000003F = 63 (dec)

`count` The number of CAN Tx messages which are sent with one function call is set with this parameter. This parameter corresponds to the input parameter *len* in *canWrite()*, *canWriteT()*, *canRead()* and *canReadT()* calls.
Default: `count` = 1

`txbuf` Size of Tx queue.
This parameter corresponds to the parameter *txqueuesize* in the *canOpen()* call.
Default: `Tx-buf` = 10

`rxbuf` Size of Rx queue.
This parameter corresponds to the parameter *rxqueuesize* in the *canOpen()* call.
Default: `Rx-buf` = 100

`txtout` Tx timeout in [ms].
This parameter corresponds to the parameter *txtimeout* in the *canOpen()* call.
Default: `txtout` = 1000 ms

`rxtout` Rx timeout in [ms].
This parameter corresponds to the parameter *rxtimeout* in the *canOpen()* call.
Default: `rxtout` = 5000 ms

`baud` Bit rate index (Refer to bit rate table at function *canSetBaudrate()* on page 48.) By setting the most significant bit it is possible to program the BTR registers directly, as described for *canSetBaudrate()*.
Default: baud = 2 (500 kBit/s)

`testcount` Number of test loops that shall be executed.
Default: in transmit operations `testcount` = 10
in receive operations `testcount` = ∞
To set the value to '∞' the parameter `testcount` has to be set to '-1'.

`data0...data7` Data bytes to be transmitted.
If no entry is made for these parameters, two 32-bit counter values will be transmitted at transmission commands. If one to eight bytes are specified, the specified bytes will be transmitted. The data bytes have to be specified as decimal numbers !

8. References

[1] CAN-API Manual, Part 2: Installation Guide
esd electronic system design gmbh

[2] ISO 11898
Road vehicles – Interchange of digital information – Controller area Network (CAN) for high-speed communication

[3] CAN Specification 2.0
Robert Bosch GmbH, 1991

9. Appendix

9.1 Bit Timing Values (Examples)

For the determination of the bit timing and the bit rate by means of the register values please consult the manuals of controllers SJA1000 (Philips), 82C200 (Philips) or 82527 (Intel) or MB90F543 (Fujitsu).

The SJA1000 manual can be downloaded from the internet, for instance from the Philips homepage at:

`http://www-us7.semiconductors.philips.com/pip/SJA1000.html`

Note:	Please note that in order to work a CAN network does not only require that all CAN participants have the same bit rate, but that the other timing parameters should be identical as well!

Bit Rate Bus Length	Nominal Bit Time t_B	Location of Sample Point *) [% of t_B]	Setting of Register BTR0 and BTR1		
			SJA1000/ 16 MHz [HEX]	SJA1000/ 20 MHz [HEX]	FUJI MB90F543 16 MHz [HEX]
1 Mbit/s 25 m	1 µs	75 %	00 14	00 16	2B 00
800 kBit/s 50 m	1,25 µs	80 %	00 16	00 18	2F 00
500 kBit/s 100 m	2 µs	87,5 %	00 1C	00 2F	2B 01
250 kBit/s 250 m	4 µs	87,5 %	01 1C	01 2F	2B 03
125 kBit/s 500 m	8 µs	87,5 %	03 1C	04 1C	2B 07
100 kBit/s 650 m	10 µs	87,5 %	43 2F	04 2F	3E 07
50 kBit/s 1 km	20 µs	87,5 %	47 2F	09 2F	3E 0F
20 kBit/s 2,5 km	50 µs	87,5 %	53 2F	18 2F	7F DF
10 kBit/s 5 km	100 µs	87,5 %	67 2F	31 2F	7F FF

*) The ‘Location of Sampling Point’ is selected according to CiA recommendations.

Table: Bit-timing values

Index

29-bit identifier 23, 28, 51
82527 93
82C200 93

A
acceptance filter 23
AIX 29
Auto Answer 22

B
Baudrate 47
board_id 41
board_status 41
BTR0/1 47

C
CAN 2.0B 28
CAN-Bluetooth 28-30
CAN-ISA/200 28-30
CAN-ISA/331 28-30
CAN-PC104/200 28-30
CAN-PC104/331 28-30
CAN-PCC 28-30
CAN-PCI/200 28-30
CAN-PCI/266 28-30
CAN-PCI/331 28-30
CAN-PCI/360 28-30
CAN-PCI/405 28-30
CAN-USB-Mini 28-30
CAN_IF_STATUS 41
canClose() 46
canGetBaudrate() 50
canGetOverlappedResult() 66
canIdAdd() 51
canIdDelete() 52
canIoctl() 53
canRead() 59
canReadT() 63
canSend() 65
canSetBaudrate() 47
canTake() 58
canTakeT() 62
canWrite() 64
CMSG buffer 35, 58
CMSG_T buffer 38, 62
count 90
CPCI-405 31
CPCI-CAN/200 28-30
CPCI-CAN/331 28-30
CPCI-CAN/360 28-30
CPCI-CPU/750 31

D
data types 33

E
EPPC-405 31
EPPC-405-HR 31
EPPC-405-UC 31
ERROR 69
EtherCAN 28-30
Event IDs 39
Event Interface 26
EVMSG buffer 39
example program
    receive 84
    send 86

F
Features 12
FIFO Mode 18

I
id 35
int8_t 34
Interaction 13, 37

L
len 36
Linux 29
Listen-Only Mode 25
LOST_FIFO 69
LOST_MESSAGE 69
LynxOS 29

M
matrix 28
MB90F543 93

N
No_Data 37
NTCAN_20B_BASE 35
NTCAN_BAUD 48
NTCAN_CONTR_BUSY 51, 74
NTCAN_CONTR_OFF_BUS 74
NTCAN_CONTR_WARN 74
NTCAN_EV_BASE 39
NTCAN_EV_BAUD_CHANGE 70
NTCAN_EV_CAN_ERROR 69, 70
NTCAN_EV_LAST 39
NTCAN_EV_USER 39
NTCAN_FEATURE_CAN_20B 42
NTCAN_FEATURE_CYCLIC_TX 42
NTCAN_FEATURE_DEVICE_NET 42
NTCAN_FEATURE_FULL_CAN 42
NTCAN_FEATURE_LISTEN_ONLY_MODE 42
NTCAN_FEATURE_RX_OBJECT_MODE 42
NTCAN_FEATURE_SMART_DISCONNECT 42
NTCAN_FEATURE_TIMESTAMP 42
NTCAN_HANDLE 34
NTCAN_HANDLE_FORCED_CLOSE 79
NTCAN_ID_ALREADY_ENABLED 75
NTCAN_ID_NOT_ENABLED 75
NTCAN_INSUFFICIENT_RESOURCES 75
NTCAN_INTERACTION 37
NTCAN_INVALID_DRIVER 76
NTCAN_INVALID_FIRMWARE 76

NTCAN_INVALID_HANDLE 76
NTCAN_INVALID_HARDWARE 77
NTCAN_INVALID_PARAMETER 59, 77
NTCAN_IO_ INCOMPLETE 66
NTCAN_IO_INCOMPLETE 77
NTCAN_IO_PENDING 66, 78
NTCAN_IOCTL_ABORT_RX 54
NTCAN_IOCTL_ABORT_TX 54
NTCAN_IOCTL_FLUSH_RX_FIFO 53
NTCAN_IOCTL_GET_RX_MSG_COUNT 53
NTCAN_IOCTL_GET_RX_TIMEOUT 53
NTCAN_IOCTL_GET_TIMESTAMP 27, 54
NTCAN_IOCTL_GET_TIMESTAMP_FREQ 27, 54
NTCAN_IOCTL_GET_TX_TIMEOUT 53
NTCAN_IOCTL_SET_20B_HND_FILTER 24, 53
NTCAN_IOCTL_SET_RX_TIMEOUT 54
NTCAN_IOCTL_SET_TX_TIMEOUT 54
NTCAN_IOCTL_TX_OBJ_AUTOANSWER_OFF 55
NTCAN_IOCTL_TX_OBJ_AUTOANSWER_ON 22, 55
NTCAN_IOCTL_TX_OBJ_CREATE 21, 55
NTCAN_IOCTL_TX_OBJ_DESTROY 55
NTCAN_IOCTL_TX_OBJ_SCHED_START 21, 56
NTCAN_IOCTL_TX_OBJ_SCHED_STOP 56
NTCAN_IOCTL_TX_OBJ_SCHEDULE 21, 55
NTCAN_IOCTL_TX_OBJ_UPDATE 22, 55
NTCAN_LISTEN_ONLY_MODE 25, 49
NTCAN_MAX_RX_QUEUESIZE 44
NTCAN_MAX_TX_QUEUESIZE 44
NTCAN_MODE_MARK_INTERACTION 37, 44
NTCAN_MODE_NO_DATA 44
NTCAN_MODE_NO_INTERACTION 44
NTCAN_MODE_NO_RTR 44
NTCAN_MODE_OBJECT 20, 44
NTCAN_MODE_OVERLAPPED 44
NTCAN_NET_NOT_FOUND 78
NTCAN_NO_BAUDRATE
receive 50
NTCAN_NO_DATA 37
NTCAN_NO_ID_ENABLED 78
NTCAN_NOT_IMPLEMENTED 81
NTCAN_NOT_SUPPORTED 82
NTCAN_OPERATION_ABORTED 79
NTCAN_PENDING_READ 79
NTCAN_PENDING_WRITE 80
NTCAN_RESULT 34
NTCAN_RTR 36
NTCAN_RX_TIMEOUT 80
NTCAN_SOCK_CMD_TIMEOUT 83
NTCAN_SOCK_CONN_TIMEOUT 83
NTCAN_SOCK_HOST_NOT_FOUND 83
NTCAN_SUCCESS 73
NTCAN_TX_ERROR 80
NTCAN_TX_TIMEOUT 81
NTCAN_USER_BAUDRATE 49
NTCAN_WRONG_DEVICE_STATE 81

O
Object Mode 20, 21
overlapped 66

P
Plug&Play 12
PMC-CAN/266 28-30
PMC-CAN/331 28, 30
PMC-CPU/405 31
PowerMAX OS 29

Q
QNX4 30
QNX6 30

R
Return Codes 73
RTOS-UH 30
RTR 36
RTX 30
Rx identifier 51
rxbuf 91
rxqueuesize 43
rxtout 91

S
Scheduling 21
SGI-IRIX6.5 29
SJA1000 93
Solaris 29

T
testcount 91
Timestamps 27
txbuf 90
txqueuesize 43
txtimeout 43
txtout 91

V
version number 41
VME-CAN2 28-30
VME-CAN4 28-30

W
Win CE .4.2 28
Windows XP x64 Edition 28
Windows CE .NET 28
Windows RTX 28
Windows Server 2003 28
Windows2000/XP 28
Windows95/98/ME 28
WindowsNT 28