NTCAN

Part 1:
C/C++ Software Design Guide

Application Developers Manual

esd electronics gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
http://www.esd.eu
Phone: +49 (0) 511 3 72 98-0 • Fax: +49 (0) 511 3 72 98-68

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. esd electronics 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. In particular descriptions and technical data specified in this document may not be constituted to be guaranteed product features in any legal sense.

esd electronics reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd electronics gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone:	+49-511-372 98-0
Fax:	+49-511-372 98-68
E-Mail:	info@esd.eu
Internet:	www.esd.eu

Document file:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\C.2001.21_NTCAN\API\CAN-API_Manual_en_5_8.odt
Manual Order no.:	C.2001.21 This order no. covers the two parts: Part 1: 'NTCAN, Application Developers manual' (this document) and Part 2: 'NTCAN, Installation Manual'.
Date of print:	2024-01-15

Products covered by this document

CAN-Driver / SDK	(Driver) Revision
CAN SDK for Windows	4.x.y
Windows 95/98/ME VxD-Driver	1.x.y
Windows NT Device Driver	2.x.y
Windows 2000 Windows XP (32/64-Bit) Windows Vista (32/64-Bit) Windows 7 (32/64-Bit) Windows 8 / 8.1 (32/64-Bit) Windows 10 (32/64-Bit) Windows 11 (64-Bit)	2.x.y 3.x.y 4.x.y
Linux Driver (32-/64-Bit)	3.x.y 4.x.y
LynxOS Driver	1.x.y
PowerMAX OS Driver	1.x.y
Solaris-Driver	3.x.y
SGI-IRIX6.5 Driver	2.x.y
AIX Driver	1.x.y
VxWorks 5.x/6.x (Non-VxBus) VxWorks 6.x (VxBus) VxWorks 7.x (VxBus GEN2)	2.x.y 3.x.y 4.x.y
QNX4 Driver	2.x.y
QNX6 / QNX 7 Driver	3.x.y / 4.x.y
RTOS-UH Driver	2.x.y
RTX / RTX64 Driver	3.x.y / 4.x.y
INtime Driver	4.x.y
On Time RTOS-32	3.x.y

CAN-Hardware	Order No.
EtherCAN	C.2050.xx
EtherCAN/2	C.2051.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-PCI/266	C.2036.xx
CAN-PCI/331	C.2020.xx
CAN-PCI/360	C.2022.xx
CAN-PCI/400	C.2048.xx
CAN-PCI/402	C.2049.xx
CAN-PCI/402-FD	C.2049.xx
CAN-PCI/405	C.2023.xx
CAN-PCIe/200	C.2042.xx
CAN-PCIe/400	C.2043.xx
CAN-PCIe/402	C.2045.0x
CAN-PCIe/402-FD	C.2045.xx
CAN-PCIeMini/402	C.2044.xx
CAN-PCIeMini/402-FD	C.2044.xx
CAN-PCIeMiniHS/402-FD	C.2054.xx
CAN-M.2/402-2-FD	C.2074.xx
PMC-CAN/266	C.2040.xx
PMC-CAN/331	C.2025.xx
PMC-CAN/400	C.2047.xx
PMC-CAN/402-FD	C.2028.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-CAN/400	C.2033.xx
CPCI-CAN/402	I.2332.xx
CPCI-CAN/402-FD	I.2332.xx
CPCI-405	I.2306.xx
CPCI-CPU/750	I.2402.xx
CPCIserial-CAN/402	I.3001.04
CPCIserial-CAN/402-FD	I.3001.6x
CAN-PCC	C.2422.xx
CAN-USB/Mini	C.2464.xx
CAN-USB/Micro	C.2068.xx
CAN-USB/2	C.2066.xx
CAN-USB/3-FD	C.2076.xx
CAN-USB/400	C.2069.xx
CAN-USB/400-FD	C.2069.xx
CAN-CBX-AIR/2	C.3051.xx
CAN-CBX-AIR/3	C.3052.xx
VME-CAN2	V.1405.xx
VME-CAN4	V.1408.xx
AMC-CAN4	U.1002.xx
XMC-CAN/402-FD	C.2018.xx

Document History

The changes in the document listed below affect changes in the software as well as changes in the description of the facts, only.

Rev.	Chapter	Changes versus previous version	Date
5.8	N/A	Editorial changes	2024-01-05
		Revised document for INtime driver support	2024-01-05
5.7	N/A	Editorial changes	2023-01-02
	3.7	Revised and refined the chapter of NTCAN event handling with focus on the new event types `NTCAN_EV_GPIO_XXX`.	2023-08-10
	3.16	New chapter covering the *Switchable Bus Termination* support.	2023-07-12
	3.17	New chapter covering the GPIO support.	2023-07-12
	4.2.9	Description of new command for ***canIoctl()***: `NTCAN_IOCTL_SET_TERM_CFG`, `NTCAN_IOCTL_GET_TERM_CFG`, `NTCAN_IOCTL_SET_GPIO_CFG` and `NTCAN_IOCTL_GET_GPIO_CFG`.	2023-07-12
	6.2.2	Feature flags `NTCAN_FEATURE_PROG_TERM` and `NTCAN_FEATURE_GPIO`	2023-01-06
	6.2.11	Description of `EV_GPIO_DATA`.	2023-08-10
	6.2.25	Description of `NTCAN_GPIO_CFG`.	2023-01-06
	6.2.26	Structure `NTCAN_INFO` extended with number of GPIO ports, GPIO configuration capabilities, GPIO core version and the product order number as ASCII string.	2023-01-06
	6.2.26	Table with transceiver types extended.	2023-03-28
5.6	N/A	Editorial changes	2022-12-02
	1.3	Extended the terminology chapter.	2022-12-09
	3.12.1	Extended description of the *Scheduling Mode* to clarify existing restrictions.	2022-10-13
	3.18	OS support revised and updated.	2022-29-02
	4.2.9	Description of new command for ***canIoctl()***: `NTCAN_IOCTL_SET_DAR_MODE` and `NTCAN_IOCTL_GET_DAR_MODE`	2022-11-24
	6.2.26	Structure `NTCAN_INFO` extended with number of LIN ports.	2022-11-30
	8.1	Revised example code for receiving data in FIFO mode	2022-12-07
	8.3	New example code to demonstrate the Rx object mode.	2022-12-02
5.5	N/A	Editorial changes	2022-05-13
	3.2	Extended with the description of the *Disable Automatic Retransmission (DAR)* mode.	2022-05-13
	3.11.1	Added hints how to detected updates in Rx Object Mode.	2022-05-18
	4.4.4 4.4.5 4.4.6	Added failure of initial transmission in DAR mode to the list of reasons the blocking I/O operation is completed.	2022-05-13
	6.2.2	Feature flags `NTCAN_FEATURE_DAR`	2022-05-13
	6.2.3	Description of the flag `NTCAN_DAR`	2022-05-13
	6.2.3	Described new `msg_lost` functionality in Rx Object Mode	2022-05-18

Rev.	Chapter	Changes versus previous version	Date
	7.1	Revised error code descriptions for cancelled transmissions.	2022-05-25
	10.1.2	Revised and updated.	2022-05-17
5.4	N/A	Editorial changes	2021-11-19
	3.3.5	Description of the *Transmit Pause* feature.	2021-10-13
	3.9	Extended description of timestamping	2021-10-21
	3.11.2	Revised description of Rx Object scheduling.	2021-10-28
	3.12.1	Revised description of Tx Object scheduling.	2021-10-15
	3.15	TDC/SSP configuration description enhanced and revised.	2021-11-12
	3.18	Updated for Win11 support and new hardware.	2021-11-19
	4.2.1	Added layout description of user defined bitrate configuration register for CAST IP cores and M_CAN.	2021-03-04

	4.2.9	Description of new command for ***canIoctl()***:
		`NTCAN_IOCTL_TX_OBJ_AUTOANSER_ONCE`
		`NTCAN_IOCTL_LIN_MASTER_SEL`
		`NTCAN_IOCTL_GET_HW_TIMESTAMP`
		`NTCAN_IOCTL_GET_HW_TIMESTAMP_EX`
			2020-07-16

	4.3.1 4.3.2 4.3.3	Corrected misleading general description that the *Rx Object Mode* is limited to 11-bit CAN-IDs which is only true for device driver versions before 3.x.	2021-10-28
	5	Macros `NTCAN_GET/SET_TDC_XXX` revised and extended.	2021-11-19
	6.2.2	Definition for new controller type `NTCAN_CANCTL_CAST`, `NTCAN_CANCTL_ESDLIN`, `NTCAN_CANCTL_MSAM`.	2021-09-06
	6.2.2	Feature flag `NTCAN_FEATURE_TX_PAUSE`	2021-10-13
	6.2.16	Added description of flag `NTCAN_BAUDRATE_FLAG_TXP`	2021-10-13
	6.2.16	Added description of flag `NTCAN_BAUDRATE_FLAG_TDC`	2021-11-17
	6.2.27	Description of `NTCAN_TDC_CFG`.	2011-11-17
	6.2.26	Extended table with transceiver types.	2021-03-04
	6.2.26	Structure `NTCAN_INFO` extended with frequency of the driver internal high resolution (software) timestamp.	2021-10-20
	8.6 - 8.7	Example source code for timestamped Tx transmission.	2020-07-16
	9	Added description to start tests in listen-only/self-test mode,	2021-10-22
5.3	N/A	Editorial changes	2019-07-16
	1.5	New chapter introducing LIN support for some esd electronics boards.	2019-07-15
	3.12	Tx object mode description revised and updated for CAN FD.	2019-07-11
	3.15.2	Fixed description of TDC defines and macros.	2018-08-01
	4.1.1	Revised description of ***canOpen()*** for LIN support.	2019-07-15

	4.2.9	Description of new commands for ***canIoctl()***:
		`NTCAN_IOCTL_TX_OBJ_CREATE_X`, `NTCAN_IOCTL_TX_OBJ_UPDATE_X`
		`NTCAN_IOCTL_TX_OBJ_DESTROY_X`, `NTCAN_IOCTL_TX_MSG_COUNT`
			2019-07-10

	6.1.1	Description of `NTCAN_NO_HANDLE` as invalid handle value.	2018-08-24
	6.2.2	Feature flag `NTCAN_FEATURE_LIN`	2019-07-15
	6.2.9	Description of *dma_stall* for `EV_CAN_ERROR`.	2018-07-03

Rev.	Chapter	Changes versus previous version	Date
	7.1	New error codes NTCAN_NO_XXX_CAPABILITY.	2019-07-15
	8.8	Example source code for Tx scheduling mode.	2019-07-24
	B.1	Corrected interchanged 'Direction' bit in table 38	2018-07-11
5.2	N/A	Editorial changes	2018-06-11
	3.9	Updated and extended description of timestamps.	2018-06-03
	3.18	Updated for QNX7 support and new hardware.	2018-05-08
	4.1.1	Clarified meaning of NTCAN_NO_QUEUE as handle queue size.	2018-01-24
	8.5	Fixed example code	2017-11-24
5.1	N/A	Editorial changes	2017-06-08
	4.5.6	Revised wrong description of ***canFormatEvent()***.	2017-03-24
	Annex B	Annex to describe the NTCAN bus error codes.	2016-11-23
5.0	N/A	Documentation completely revised for CAN FD	2016-11-10
	1.4	New chapter with an introduction of CAN FD.	2016-10-21
	3.3	Revised bit rate configuration description for CAN FD	2016-10-07
	3.3.4	New chapter to describe the triple sampling mode	2016-10-13
	3.4	Revised CAN message structure description for CAN FD	2016-10-07
	3.15	New chapter to describe the CAN FD TDC	2016-10-26
	4.2.3	Description of new API function ***canSetBaudrateX()***	2016-08-24
	4.2.4	Description of new API function ***canGetBaudrateX()***	2016-08-24
	4.2.9	Description of CAN FD TDC commands for ***canIoctl()***	2016-10-26
	4.3.3	Description of new API function ***canTakeX()***	2016-05-31
	4.3.6	Description of new API function ***canReadX()***	2016-05-31
	4.4.3	Description of new API function ***canSendX()***	2016-05-31
	4.4.6	Description of new API function ***canWriteX()***	2016-05-31
	4.5.4	Description of new API function ***canGetOverlappedResultX()***	2016-06-01
	5	Revised macro description for CAN FD and added description for NTCAN_NTCAN_IS_FD, NTCAN_DATASIZE_TO_DLC, NTCAN_LEN_TO_DATASIZE and NTCAN_IS_FD_WITHOUT_BRS	2016-06-01
	5.10 5.16	Added new macros to configure or get TDC configuration.	2016-10-26
	6.2.2	Feature flag NTCAN_FEATURE_TRIPLE_SAMPLING	2016-10-13
	6.2.3	Description of CMSG_X revised for CAN FD.	2016-10-25
	6.2.5	Description of CMSG_X.	2016-10-25
	6.2.8	Extended description of struct EV_BAUD_CHANGE for CAN FD.	2016-10-06
	6.2.14	Description of EVMSG_X.	2016-10-10
	6.2.17	Structure of NTCAN_BITRATE revised for CAN FD.	2016-09-30
	6.2.18	Structure NTCAN_BUS_STATISTIC extended for CAN FD.	2016-10-06
	6.2.26	Structure NTCAN_INFO extended with number of open handles.	2016-10-27
	8	Example source code to transmit/receive CAN FD messages.	2016-11-09

Rev.	Chapter	Changes versus previous version	Date
	9	Description of the CAN FD support in *canTest.*	2016-10-13
4.7	6.2.2	Documentation of hardware status `NTCAN_BSTATUS_XXX`	2015-10-07
	N/A	Revised document for LynxOS 3.x driver support	2015-06-19
	N/A	Fixed `NTCAN_GET_BOARD_STATUS` was falsely documented as `NTCAN_GET_HW_STATUS`.	2015-10-07
	N/A	Editorial changes.	2016-04-07
4.6	2.2	Introduction of the 4.x driver	2015-01-27
	4.2.1	Extended description to configure a user defined BTR	2015-01-28
	6.2.21	New option 'number_of_repeat' for EEI unit	2015-02-09
	Annex A	Annex to describe the (ESDACC) CAN bus timing	2015-02-09
	N/A	Editorial changes.	2015-02-09
4.5	3.3.3	New chapter to describe the Self Test Mode	2013-10-09
	3.8	Acceptance filtering completely revised with detailed description of the *Smart ID Filter*.	2014-09-11
	4.2.1	Description of STM bit in ***canSetBaudrate()***	2013-10-09
	4.2.6/4.2.8	Description of ***canIdRegionAdd()/canIdRegionDelete()***	2014-09-09
	4.2.9	Description of new command for ***canIoctl()***: `NTCAN_IOCTL_GET_INFO`	2014-08-26
	6.2.2	Description of the feature flag `NTCAN_FEATURE_CAN_FD`	2014-08-26
	6.2.17	Partially correct description of bit rate details fixed	2013-10-09
	6.2.23	Description of `NTCAN_FILTER_MASK`.	2014-09-05
	6.2.26	Description of `NTCAN_INFO`.	2014-08-26
	11	Pictures of VIs updated	2014-05-27
	11.1.1	Chapter restructured, VI icons updated	2014-05-27
	11.1.2	All VI icons and the LabView block diagram of CanObjectPoll updated	2014-05-27
	11.1.1.2.5	Block diagrams updated and description of error injection related IO controls inserted	2014-05-27
	11.1.1.5.3	New chapter 'canFormatFrame'	2014-05-27
	N/A	Editorial changes.	2014-09-11
4.4	6.2.2	Definition for new supported CAN controller types `NTCAN_CANCTL_SITARA / NTCAN_CANCTL_MCP2515`.	2013-08-07
	9	Description of new test 20 for feature *Timestamped TX.*	2013-02-18
	N/A	Editorial changes.	2013-02-18
4.3	1.6	Added feature *Timestamped TX*	2013-01-09
	2.3.4	Added *Timestamped TX* to feature overview	2013-01-31
	3.14	Added *Timestamped TX* as chapter 3.14	2013-01-31
	3.18	Corrected features of C331 driver for Windows / RTX	2013-01-18
	3.18	EPPC-405-HR/EPPC-405-UC Operating System Support	2013-02-05
	4.1.1	Description of `NTCAN_MODE_TIMESTAMPED_TX` for ***canOpen()***.	2013-01-07
	4.2.5	Remarks to the ***canIdAdd()*** behaviour with USB devices.	2013-01-07

Rev.	Chapter	Changes versus previous version	Date
	4.2.9	Description of new commands for ***canIoctl()***: `NTCAN_IOCTL_SET_TX_TS_WIN`, `NTCAN_IOCTL_GET_TX_TS_WIN`, `NTCAN_IOCTL_SET_TX_TS_TIMEOUT`, `NTCAN_IOCTL_GET_TX_TS_TIMEOUT`	2013-01-09
	4.4.2	Added information on mode *Timestamped TX*	2013-01-09
	4.4.4	Fixed description of value returned in len parameter	2013-01-31
	4.4.5	Changed for *Timestamped TX*	2013-01-31
	6.2.2	Description of the new feature flags `NTCAN_FEATURE_PXI` and `NTCAN_FEATURE_TIMESTAMPED_TX`.	2013-01-09
	N/A	Editorial changes.	2013-01-09
4.2	3.2	More detailed description of communication error types.	2012-12-03
	9	Updated for *canTest* V 2.11.1 and later	2012-11-16
	11	Added chapter 'Attachment' (LabVIEW documentation)	2012-12-13
	N/A	Editorial changes.	2012-11-13
4.1	4.2.9	Fixed example for `NTCAN_IOCTL_GET_SERIAL`.	2012-04-26
	5	Description of the new macros `NTCAN_DLC, NTCAN_IS_RTR, NTCAN_DLC_AND_TYPE and NTCAN_IS_INTERACTION`	2012-07-18
	6.2.2	New controller type `NTCAN_CANCTL_FLEXCAN`.	2012-05-07
	N/A	Editorial changes.	2012-05-02
4.0	N/A	The complete document is revised, large parts of the text are rewritten and the figures are updated.	
	3.4	New chapter to describe *NTCAN-ID* concept.	
	3.5	New chapter to describe the *Interaction* concept.	
	3.6	New chapter to describe the CAN bus diagnostic.	
	3.8	New chapter to describe the message acceptance filtering.	
	3.13	New chapter to describe the *Error Injection* concept.	
	4.1.1	Description of flag `NTCAN_MODE_LOCAL_ECHO` for ***canOpen()***.	

	4.2.9	Description of new commands for ***canIoctl()***:
		`NTCAN_IOCTL_SET_BUSLOAD_INTERVAL`
		`NTCAN_IOCTL_GET_BUSLOAD_INTERVAL`
		`NTCAN_IOCTL_GET_BUS_STATISTIC`
		`NTCAN_IOCTL_GET_CTRL_STATUS`
		`NTCAN_IOCTL_GET_BITRATE_DETAILS`
		`NTCAN_IOCTL_GET_NATIVE_HANDLE`
		`NTCAN_IOCTL_SET_HND_FILTER`	

	4	Removed description of deprecated ***canReadEvent()*** and ***canSendEvent().***	
	4.4.2	Description of new API function ***canSendT()***	
	4.4.5	Description of new API function ***canWriteT()***	
	4.5.3	Description of new API function ***canGetOverlappedResultT()***	
	4.5.5	Description of new API function ***canFormatError()***.	
	4.5.6	Description of new API function ***canFormatEvent()***.	
	5	New chapter for NTCAN-API macros.	

Rev.	Chapter	Changes versus previous version	Date
	6.2.2	Description of the feature flags: NTCAN_FEATURE_LOCAL_ECHO NTCAN_FEATURE_SCHEDULING NTCAN_FEATURE_DIAGNOSTIC NTCAN_FEATURE_SMART_ID_FILTER	
	6.2.2	Revised description of member *boardstatus* in CAN_IF_STATUS..	
	6.2.7	Description of CSCHED and scheduling mode flags.	
	6.2.8-6.2.10	Description of the NTCAN events EV_BAUD_CHANGE, EV_CAN_ERROR and EV_CAN_ERROR_EXT.	
	6.2.13	Description of EVMSG_T.	
	6.2.17	Description of NTCAN_BITRATE.	
	6.2.18	Description of NTCAN_BUS_STATISTIC.	
	6.2.19	Description of NTCAN_CTRL_STATE.	
	6.2.20 - 6.2.21	Description of NTCAN_EEI_STATUS and NTCAN_EEI_UNIT.	
	6.2.22	Description of NTCAN_FORMATEVENT_PARAMS.	
	6.2.24	Description of NTCAN_FRAME_COUNT.	
	7.1	New error codes NTCAN_CONTR_ERR_PASSIVE, NTCAN_ERROR_LOM and NTCAN_ERROR_NO_BAUDRATE.	
	9	Description of *canTest* console application completely revised.	
	10	New chapter to describe the development process for NTCAN based applications and the Windows CAN SDK	

Technical details are subject to change without further notice.

Table of contents

Index of Tables

List of Figures

Typographical Conventions

Throughout this manual the following typographical conventions are used to distinguish technical terms.

Convention	Example
File and path names	**`/dev/null`** or **`<stdio.h>`**
Function names	***open()***
Programming constants	`NULL`
Programming data types	`uint32_t`
Variable names	*Count*

The following indicators are used to highlight noticeable descriptions.

Notes to point out something important or useful.

Caution: Cautions to tell you about operations which might have unwanted side effects.

Number Representation

All numbers in this document are **base 10** unless designated otherwise. Hexadecimal numbers have a prefix of **0x**, and binary numbers have a prefix of **0b**. For example, 42 is represented as 0x2A in hexadecimal and 0b101010 in binary.

Abbreviations

ABI	Application Binary Interface
ACK	Acknowledgement
ACR	Acceptance Code Register
AMC	Advanced Mezzanine Cards
AMR	Acceptance Mask Register
API	Application Programming Interface
BIF	Basic ID Filter
BRP	Baud Rate Prescaler
BRS	Bit Rate Switch
BTR	Bus Timing Register
CAN	Controller Area Network
CBFF	Classical Base Frame Format
CEFF	Classical Extended Frame Format
CPU	Central Processing Unit
CiA	CAN in Automation
CLI	Command Line Interface
CPCI	Compact Peripheral Component Interconnect (Computer Bus)
CRC	Cyclic Redundancy Check
DAR	Disable Automatic Retransmission
DLC	Data Length Code
DLL	Dynamic Link Library
DMA	Direct Memory Access
EFF	Extended Frame Format
esdACC	**esd electronics** Advanced CAN Controller
ESI	Error State Indication
FBFF	FD Base Frame Format
FD	Flexible Data.
FEFF	FD Extended Frame Format
FIFO	First-In-First-Out
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HW	Hardware
I/O	Input/Output
ISA	Industry Standard Architecture (Computer Bus)
IPC	Interprocess Communication
IRIG	Inter-range Instrumentation Group
LabVIEW®	Laboratory Virtual Instrumentation Engineering Workbench (Application)
LIN	Local Interconnect Network
LOM	Listen Only Mode
LSB	Least Significant Bit
LSW	Least Significant Word
MCU	Micro Controller Unit
MSB	Most Significant Bit
MSW	Most Significant Word
mtq	Minimum Time Quantum = CAN clock period
tq	Time Quantum
N/A	Not Applicable
OS	Operating System
PCI	Peripheral Component Interconnect (Computer Bus)
PCIe	Peripheral Component Interconnect Express (Computer Bus)
PMC	PCI Mezzanine Card
PXI	PCI Extension for Instrumentation (Computer Bus)
REC	Receive Error Counter
SDK	Software Development Kit
SIF	Smart ID Filter
SoC	System on Chip

SFF	Standard (Base) Frame Format
SJW	Synchronous Jump Width
SP	Sample Point
SSP	Second Sample Point
SSPO	Second Sample Point Offset
SSPS	Second Sample Point Shift
STM	Self Test Mode
TD	Transmitter Delay
TDC	Transmitter Delay Compensation
TEC	Transmit Error Counter
TSEG1	Time Segment before sample point
TSEG2	Time Segment after sample point
USB	Universal Serial Bus
VME	Versa Module Eurocard (Computer Bus)
XMC	Express Mezzanine Card

Reference

/1/ esd electronics gmbh, *NTCAN-API Installation Guide*, Revision 4.8, 2024

/2/ ISO 11898-1, *Road vehicles – Controller area network (CAN) – Data link layer and physical signalling, 2015*

/3/ CAN in Automation, CiA 301, *CANopen application layer and communication profile*, Revision 4.2, February 2011

/4/ CAN in Automation, AN 801, *Automatic bit-rate detection*, Revision 1.1, November 2009

/5/ Philips Semiconductors, *Data sheet SJA1000 Stand-alone CAN controller*, January 2000

/6/ esd electronic system design, *CAN Error Injection, a simple but versatile approach*, Paper from international CAN Conference (iCC) 2012

/7/ Bit Time Requirements for CAN FD, Florian Hartwich, Robert Bosch GmbH, 14th International CAN Conference, iCC 2013 Paris

/8/ CAN in Automation, CiA 601-1, CAN FD Node and System Design: Physical Interface Implementation, V1.0.1, June 2016

/9/ esd electronics gmbh, *NTLIN-API Application Developers Manual*, Revision 1.0, 2019

This page intentionally left blank

1. Introduction

This document describes the software design and the application layer of the cross platform communication interface for Classical Controller Area Network (CAN CC) and Controller Area Network with Flexible Data Rate (CAN FD) **esd electronics** hardware. The well structured Application Programming Interface (API) allows an easy integration into any application. The functional range and versatility of the implementation provide all necessary mechanisms to control, configure and monitor CAN CC / CAN FD networks. Many sophisticated features make the API ideally suited to implement higher layer CAN CC / CAN FD based protocols on top of it. Implementations for most of the prevalent protocols are already available by **esd electronics**.
Within this document the API is referred as **NTCAN-API** and the common implementation as a combination of a device driver and a library as **NTCAN**. The name has its origin in the initial implementation for Windows NT but it is now the common API for all Operating Systems (OS).

1.1 Scope

This document covers the description of the NTCAN architecture which usually consists of an OS and CAN CC / CAN FD hardware specific device driver and a (shared) library which exports the application interface to integrate CAN CC / CAN FD I/O into an application.

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

The data link layer for CAN CC / CAN FD is internationally standardized in /2/.

1.2 Overview

Chapter 1 contains a general overview on the structure of this manual.
Chapter 2 provides general overview about the features of the NTCAN implementation.
Chapter 3 describes the NTCAN concepts and how the API can be integrated into an application to realize a CAN bus based communication.
Chapter 4 describes the Application Programming Interface (API) with all functions followed by **Chapter 5** with a description of macros and **Chapter 6** which contains the reference to the simple and complex data types used with NTCAN-API.
Chapter 7 is a description of the error codes which are returned by the NTCAN-API functions described in the previous chapters in case of a failure.
Chapter 8 contains two complete, small example applications which demonstrate the transmission and reception of CAN messages.
Chapter 9 describes the example application *canTest*, which is delivered as binary and as source code. This console test program can be used to verify the installation and to test the CAN communication.
Chapter 10 introduces the NTCAN based application development especially with the CAN SDK for Windows.
Chapter 11 is an attachment of this document which covers the NTCAN support with other programming languages than C/C++.

1.3 Terminology

Within this manual you will encounter the following terms:

Term	Description
Arbitration Phase	Phase in which the nominal bit time is used.
Base Frame Format	CAN messages with 11-bit CAN-IDs according to /2/
Base Net	The Base Net of a CAN Board is the logical net number which is assigned to the first physical CAN port.
CAN	**C**ontroller **A**rea **N**etwork A serial bus system (also known as CAN bus) that was originally designed for use in vehicles but is now also used in automation technology. With the standardization of CAN FD the original CAN is also referred to as “Classical CAN or short CAN CC” by the CiA to distinguish it from the enhanced standard.
CAN Board	A CAN board is a hardware which makes one or more physical CAN ports available for use by an application. This is either an **esd electronics** *CAN Interface* or an embedded system with an on-board *CAN Controller*.
CAN CC	Short form of *Classical CAN*.
CAN Controller	A chip whose hardware processes the CAN bus protocols. This can be a stand alone chip which is solely dedicated to this function or a *System on Chip* (SoC) which integrates one or more CAN controllers as external interface.
CAN Device	The (logical) application view to a physical CAN port.
CAN FD	CAN FD (CAN with Flexible Data Rate) is an enhancement of the CAN CC protocol. The main differences to standard CAN are the extended payload from 8 up to 64 bytes and the ability to send this payload with a higher data rate. A CAN FD controller always supports the CAN CC protocol as well
CAN Handle	Logical link between the application and a physical CAN port. An application can open several CAN handles to the same or to different CAN ports.
CAN-ID	Identifier of a CAN message either in the *Standard Frame Format* (11-bit) or the *Extended Frame Format* (29-bit)
CAN Interface	A CAN interface is a dedicated **esd electronics** hardware which is either connected to a local bus (PCI, USB, PC/104, etc.) of a CPU or remotely (Ethernet, Wireless, etc.) to a host system.
CAN Node	All hardware connected to the CAN bus. This can be any hardware with a CAN port ranging from a simple sensor up to a complex control system.
CAN Message	Logical unit which consists of a CAN-ID and a payload either as data frame or as remote request frame.
CAN Port	The physical connector to a CAN bus which is handled by a CAN controller.

Term	Description
Classical Base Frame Format (CBFF)	Format for Data Frames or Remote Frames using an 11-bit identifier, which are transmitted with one single bit rate and up to and including 8 data bytes.
Classical Extended Frame Format (CEFF)	Format for Data Frames or Remote Frames using a 29-bit identifier, which are transmitted with one single bit rate and up to and including 8 data bytes.
Classical CAN	The term *Classical CAN* (or CAN CC) is used if it is necessary to emphasize differences to the CAN FD protocol.
Data Bit Rate	Number of bits per time during data phase.
Data Bit Time	Duration of one bit in the data phase.
Data Frame	Frame which contains up to 8 bytes of data either in the Standard Frame Format or the Extended Frame Format.
Data Phase	Phase in which the data bit time is used.
Event-ID	Identifier of an NTCAN Event.
Extended Frame Format	CAN messages with 29-bit CAN-IDs according to /2/
FD Base Frame Format (FBFF)	Format for Data Frames using an 11-bit identifier, which are transmitted with a flexible bit rate and up to and including 64 data bytes according to /2/.
FD Extended Frame Format (FEFF)	Format for Data Frames using a 29-bit identifier, which are transmitted with a flexible bit rate and up to and including 64 data bytes according to /2/.
Interaction Frame	No official technical term. Describes in this document a frame which is transmitted and received on the same CAN port.
IRIG B	Time code format used to provide time-of-day information to communication systems which have to correlate data (reception) with time.
LabVIEW®	System design platform and development environment for a visual programming language from National Instruments.
LIN	**LIN** (**L**ocal **I**nterconnect **N**etwork) A serial bus system (also known as LIN bus) designed for use in vehicles as a less performant (and less expensive) supplement to the CAN bus with a completely different physical layer but distantly related to the message format of CAN CC Base Format Messages.
Linux CAN	Linux CAN is the standard support of CAN in the Linux kernel. The CAN device driver are implemented as network device driver.
Nominal Bit Rate	Number of bits per time during arbitration phase.
Nominal Bit Time	Duration of one bit in the arbitration phase.
NTCAN Event	Logical unit which consists of an Event-ID and a payload to describe the reason of the event (errors, warnings, state changes, ...).

Term	Description
NTCAN-ID	Identifier (32-Bit) of a received message which allows to distinguish between an NTCAN Event and a CAN Message.
PXI	PXI is an open bus standard for test, measurement, and control based on the CPCI.
RTR Frame	A frame transmitted to request the transmission of data frame.
SocketCAN	See Linux CAN.
Standard Frame Format	Same as *Base Frame Format.*

1.4 CAN FD

1.4.1 In a nutshell

In 2011, after more than 20 years of the introduction of the CAN CC standard, Bosch started in cooperation with CAN experts from the automotive and the automation industry the development of an enhanced protocol version to meet especially the demands of increased bandwidth. In relation to main operating principle this improved version was standardized /2/ as CAN FD (flexible data rate) in 2015.

The picture below shows the main idea. During the arbitration phase CAN FD uses the same mechanism and nominal bit (limited to 1 Mbit/s) rate as CAN CC. In the consecutive data phase the bit rate can be increased as only one node is transmitting data and synchronization between the nodes is not required. In the acknowledge phase the nodes are again re-synchronized at the nominal bit rate. In addition to the higher bit rate during the data phase the maximum payload size was increased from 8 to 64 byte.

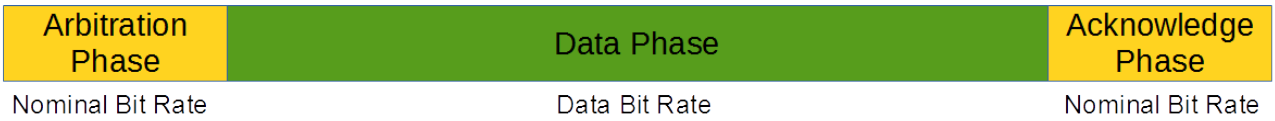


Figure 1: CAN FD Operating Principle

The combination of these two enhancements result either in a higher throughput or a reduced latency. The picture below shows these two effects for the comparison between CAN CC and CAN FD with an assumed ration of 1:8 between CAN FD nominal bit rate and data bit rate.

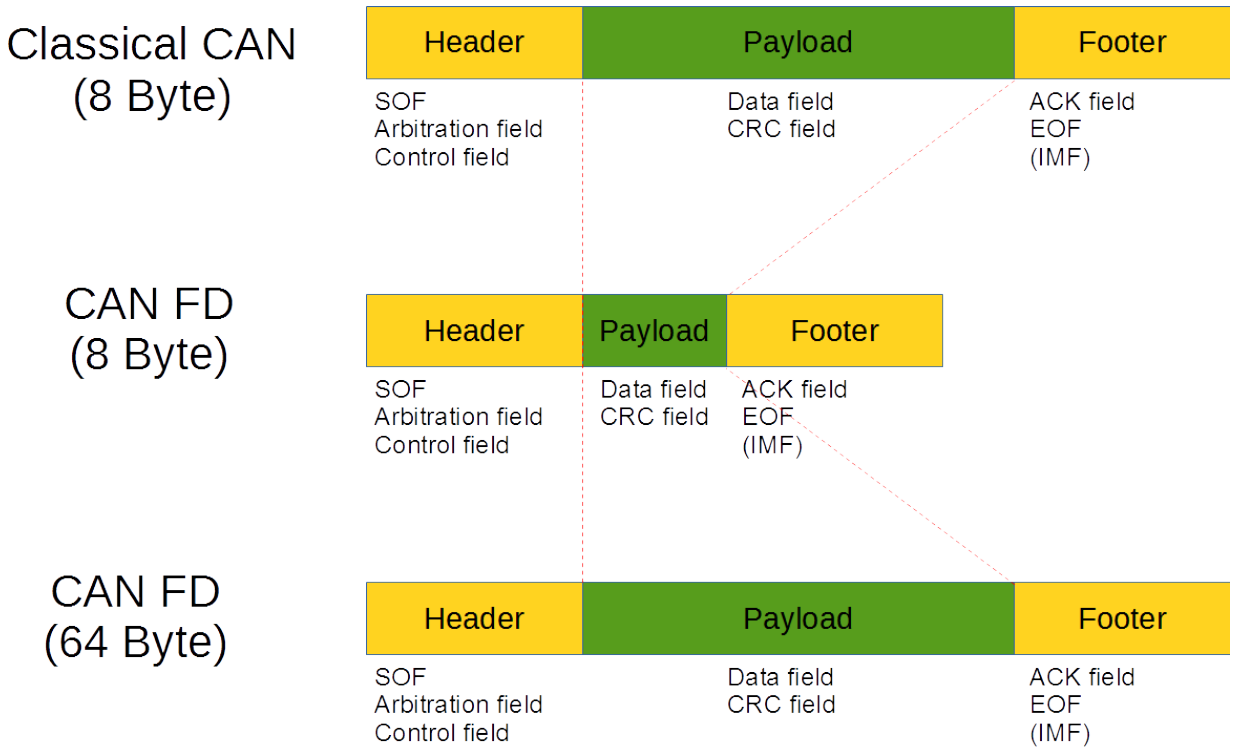


Figure 2: CAN FD vs. CAN CC Performance Improvement

In addition to these most prominent changes several other aspects of the protocol have been changed or enhanced

- CAN FD implements enhanced CRCs to improve the already very good error detection capability of CAN CC.
- To overcome the conflicts between physical limitations of the basic CAN communication principals and a much higher bit rate in the data phase a Transmitter Delay Compensation (TDC) mechanism and a Second Sample Point (SSP) for the transmitting node was introduced (refer to chapter 3.15 for more details).
- Remote Request Transmission (RTR) frames are undefined for the CAN FD frame format.
- Every transmitting node indicates its error state (active/passive) with each transmitted frame via an Error State Indication (ESI) bit.

Attention: CAN CC controller do not tolerate CAN FD frames !

Every CAN FD controller is backward compatible to the CAN CC protocol. CAN CC nodes and CAN FD nodes can communicate with each other as long as the CAN FD frame format remains unused.

1.4.2 Integration and Migration

This chapter is intended for application engineers which already use the NTCAN API for CAN CC to get a quick overview on the way CAN FD is integrated into the API and what has to be done to migrate an existing application from CAN CC to CAN FD.

The main objective integrating CAN FD support into the NTCAN library and the underlying device driver was backward compatibility so any application written for a CAN CC only version of the API will work without any changes on CAN FD capable as well as CAN CC **esd electronics** hardware.

The following changes and enhancement integrate CAN FD support into the NTCAN API:

- The increased data size of up to 64 bytes requires the new CAN message types `CMSG_X` and `EVMSG_X` which are always timestamped. Previously reserved bits in the meta data part of the CAN message *len* parameter are used to mark it as CAN FD type transmitted or received with or without bit rate switch during the data phase. The CAN FD Error State Indication flag (ESI) is reflected for received messages.
- Several macros are provided to write an application in a frame type aware way especially conversion routines between the data size and the frame´s Data Length Code (DLC).
- The new message types require the introduction of the CAN I/O functions ***canTakeX()***, ***canReadX()***, ***canSendX()*** and ***canWriteX()*** which complement the functionality of the existing I/O functions for the `CMSG` and the `CMSG_T` format. These functions can be used to send CAN FD messages as well as CAN CC messages which can be defined on a per message base.
- To prevent a legacy application from transmitting/receiving CAN FD messages inadvertently because the previously reserved bit in the parameter *len* of a CAN message was not reset as expected a CAN handle for CAN FD I/O has to be opened with the new mode flag `NTCAN_MODE_FD` set for ***canOpen()***.

- To configure the nominal bit rate and the data bit rate as an atomic operation the new function ***canSetBaudrateX()*** is introduced which can be used as well to configure just the CAN CC operation mode. Call ***canGetBaudrateX()*** to request the current bit rate configuration of a device. A call to the legacy ***canGetBaudrate(),*** if the device is configured in the CAN FD mode, will return the value `NTCAN_BAUD_FD`.
- Changing the bit rate to a CAN FD operation mode will cause two consecutive `EV_BAUD_CHANGE` events for the nominal and the data bit rate.
- Reserved fields of the structure `NTCAN_BITRATE` are used to indicate the bit rate details of a configured data bit rate.
- Reserved fields of the `NTCAN_BUS_STATISTIC` are used to count the transmitted /received CAN FD messages for statistical purposes.

1.5 LIN

The Local Interconnect Network (LIN) bus is a serial bus system designed for use in vehicles as a less performant (and less expensive) supplement to the CAN bus with a completely different physical layer.

Dedicated esd electronics boards can be extended with an add-on hardware to provide LIN ports in addition to their CAN ports. The information content of a LIN frame is similar to that of a CAN CC Base Format message. For this reason the device driver for the respective esd electronics boards also handle the LIN communication instead of using a dedicated device driver for this. The NTLIN-API [9] is implemented on top of the NTCAN library instead of accessing the device driver directly.

References to the LIN support in this document are only made to describe references inside the NTCAN-API header **`<ntcan.h>`** to LIN.

1.6 Features

The **esd electronics** NTCAN-API is a compact and easy to handle programming interface for integrating the control of CAN CC / CAN FD based networks in (real-time) applications. The implementation provides the following features* which are described in more detail in chapter 2.3.

- Device driver support of OS specific features
- Support for Plug & Play and hot-pluggable CAN devices
- Multitasking/multi-threading support
- Support for Classic CAN (CAN CC) as well as CAN with Flexible Data Rate (CAN FD)
- Event driven and/or polled CAN CC / CAN FD I/O
- CAN CC / CAN FD message interaction
- Multiprocessor and multi-core support
- Background bus-off recovery
- Firmware update for CAN CC / CAN FD modules with local operating system
- Hardware independent CAN CC / CAN FD node number mapping
- Common OS independent API on all platforms
- Sophisticated acceptance filtering for messages in base and extended frame format
- Blocking and non-blocking CAN CC / CAN FD I/O
- Event based status and error indication
- High resolution timestamps for received/transmitted frames and events
- Flexible bit rate configuration
- Intelligent CAN FD TDC Auto Configuration Mode.
- Listen only mode for non destructive CAN CC / CAN FD bus monitoring
- CAN CC bus bitrate detection
- Scheduling (single shot or cyclically) of Tx messages
- Support for extended device driver based CAN CC auto answering mechanisms
- Extended error information about CAN bus state
- CAN CC / CAN FD Error Injection for dedicated CAN hardware
- Time-triggered transmission (Timestamped TX)
- Support to disable the automatic retransmission (aka single-shot mode)
- LIN support on appropriate hardware
- GPIO support on appropriate hardware
- Programmatically switchable CAN bus termination on appropriate hardware

* Some of the features require special (CAN) hardware or operating system (OS) support. Please refer to chapter 3.18 for details which features are supported by your hardware/OS combination.

2. NTCAN-API and Device Driver

This chapter contains an overview about the features of the **esd electronics** CAN device drivers and the NTCAN Application Programming Interface (API).

2.1 Abstraction Layer

The NTCAN-API is CAN hardware and OS independent providing the same functionality for the following list of **esd electronics** CAN boards and many **esd electronics** embedded CPU boards not listed below.

Bus	CAN board
ISA	CAN-ISA/200, CAN-ISA/331
PC/104	CAN-PC104/200, CAN-PC104/331
PCI/104	CAN-PCI104/200
PCI	CAN-PCI/200, CAN-PCI/266, CAN-PCI/331, CAN-PCI/360*, CAN-PCI/400, CAN-PCI/405, CAN-PCI/402, CAN-PCI/402-FD
PCIe	CAN-PCIe/200, CAN-PCIe/400, CAN-PCIe/402, CAN-PCIe/402-FD
PCIe Mini	CAN-PCIeMini/402, CAN-PCIeMini/402-FD, CAN-M.2/402-2-FD
CPCI	CPCI-CAN/200, CPCI-CAN/331, CPCI-CAN/360*, CPCI-CAN/400, CPCI-CAN/402, CPCI-CAN/402-FD
CPCIserial	CPCIserial-CAN/402, CPCIserial-CAN/402-FD
PMC	PMC-CAN/266, PMC-CAN/331, PMC-CAN/400, PMC-CAN/402-FD
XMC	XMC-CAN/402-FD
AMC	AMC-CAN4
VME	VME-CAN2*, VME-CAN4*
Parallel Port	CAN-PCC**
USB	CAN-USB/Mini*, CAN-USB/Micro, CAN-USB/2, CAN-USB/400, CAN-USB/400-FD, CAN-USB/3-FD
Ethernet	EtherCAN**, EtherCAN/2
Wireless	CAN-AIR/2

Table 1: Supported Bus Systems

CAN / CAN FD boards marked with one asterisk (*) in the table above are not recommended for new designs and CAN boards marked with two asterisks (**) are out of production.

The NTCAN-API is implemented for the following desktop, embedded, real-time and UNIX operating systems as communication layer between application and device driver.

Windows*	UNIX	Real-Time OS
Windows 10 (32-/64 Bit) Windows 11 (64 Bit)	Linux (32-/64-Bit)	INtime 6.4 / 7.x LynxOS QNX 6.x/7.x (32-/64-Bit) RTOS-UH RTX64 VxWorks 5.x/6.x/7.x
Legacy Support:	Legacy Support:	Legacy Support:
Windows 9x/ME Windows NT Windows 2000 Windows XP (32-/64-Bit) Windows Vista (32-/64-Bit) Windows 7 (32-/64 Bit) Windows 8+ (32-/64 Bit)	PowerMAX OS SGI-IRIX 6.5 Solaris	Windows CE On-Time RTOS-32 QNX 4.x RTX

Table 2: Supported Operating Systems

Attention:

esd electronics gmbh does no longer provide support and maintenance for operating systems which are considered as ***Legacy*** according to the table above.

For several other operating systems (such as Net+OS, ThreadX,..) NTCAN implementations are available: Please contact our support team: support@esd.eu for more information or if you want NTCAN support for a certain OS which is not listed in the table above.

A DOS driver (as source) is also available for many **esd electronics** CAN modules, but this driver has an individual API which is not covered in this document. You will find the description of the DOS driver in the Document '**C Interface Library for DOS and Win 3.11**'.

Using the NTCAN-API gives the application developer the possibility to change the **esd electronics** CAN CC /CAN FD hardware as well as the operating system without the need to change the CAN I/O related parts of the application.

The basic functions are available for any combination. Depending on hardware design, operating system capability and/or software development state. Some extended features described in this manual may not be supported (yet). Please refer to chapter 3.18 for details.

* The list just contains the desktop versions of Windows but the related server versions are also covered.

2.2 Driver History

The development of the NTCAN driver/library architecture started in 1996 with the goal to have a powerful and versatile platform for CAN higher layer protocols and applications which is common on all supported operating systems independent of the CAN controller and bus interface. Since the initial release many features have been added and the internal driver implementation has undergone major changes without breaking the (backward) compatibility of the API. The picture below gives an overview on the changes and the switch from one driver architecture to another as a time line:

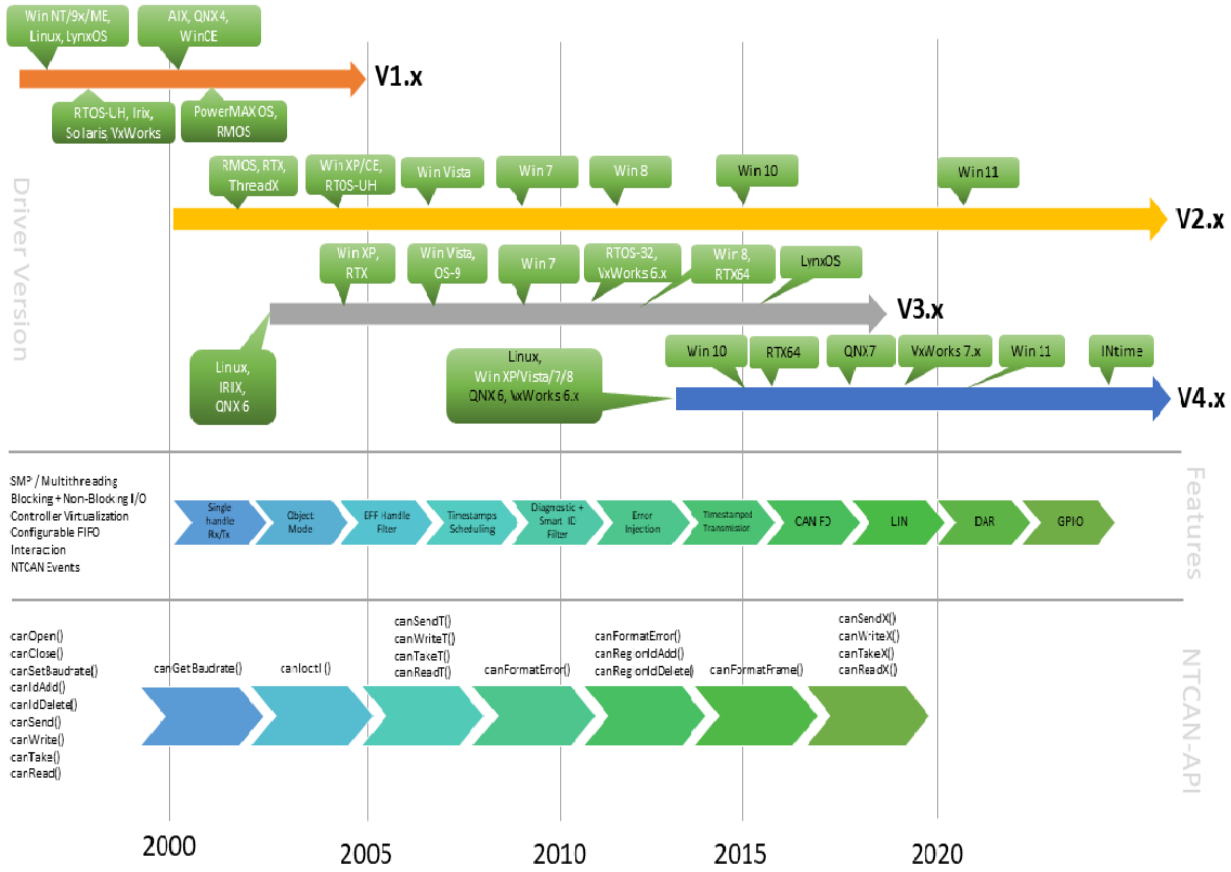


Figure 3: Time line of the NTCAN Development

Revision 1.x:

Initial release internally based on ring buffers as receive and transmit queues for CAN messages. Multiple applications can use the same physical CAN port at the same time via the abstraction of CAN handles and the support of *Interaction*.

Revision 2.x

Major internal change which removes one limitation of the 1.x driver that simultaneous reception and transmission of CAN messages with the same handle was not possible. Many new features as the *Rx Object Mode* for CAN messages with CAN-IDs in *Base Frame Format*, *Timestamps Support* and the *Extended Bus Diagnostic* are added to the device driver.

Revision 3.x

Completely new driver design which separates the CAN core functionality from the OS layer and the CAN layer to improve speed and robustness porting the driver to a new platform. Internally receive and transmit queues are based on linked lists. Several features like an extension of the *Rx Object Mode* for CAN messages with CAN-IDs in *Extended Frame Format*, *Scheduling Mode*, *Smart ID Filter, Error Injection* and *Timestamped TX* are only supported with this driver.

Revision 4.x

Major internal change to support CAN messages with up to 64 bytes and several other improvements which were introduced with the *CAN FD* standard /2/. The support for LIN enabled boards is also integrated only in this device driver branch.

Today the 2.x, 3.x and 4.x driver co-exist (sometimes even on the same OS platform). Please refer to chapter 3.18 to get the details which CAN board is supported by which driver version on which platform.

2.3 Implementation Details Overview

This chapter explains several aspects of NTCAN listed in chapter 1.6 in more detail.

2.3.1 Operating System Integration

The NTCAN implementation is usually a combination of a library which provides the API for the application and a CAN board specific device driver which is implemented on top of the host OS specific device driver interface as shown in the figure below.

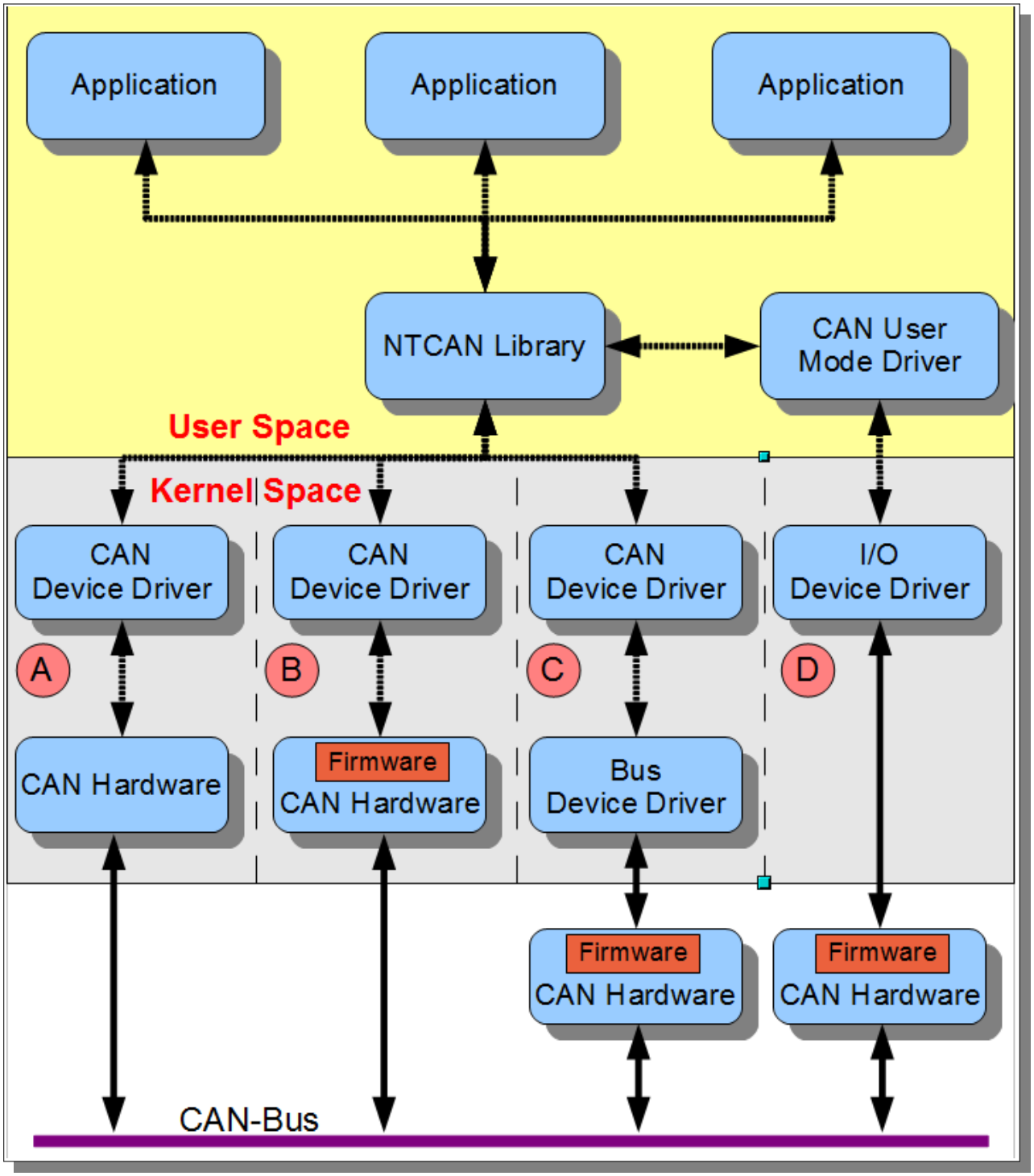


Figure 4: Integration into the Operating System

The approach to integrate the CAN driver into the OS kernel in the same way as device driver for other device classes (Network controller, Serial ports, etc.) prevents compatibility and interoperability issues and allows the use of OS specific mechanisms for driver configuration or resource cleanup mechanisms.

As shown in figure 4 the device driver implementation can be classified according to the table below:

Class	Driver Type Description
A	A kernel mode driver which directly handles a CAN controller on a CAN board that is connected to an internal bus (PCI, PCIe, CPCI, …) of the host. Examples for this class are diver for the passive CAN200 or CAN400 boards.
B	A kernel mode driver which handles a CAN board with an additional MCU and a firmware that is connected to an internal bus (PCI, PCIe, CPCI, …) of the host. Examples for this class are diver for the active CAN331 or CAN405 boards.
C	A kernel mode driver which handles a CAN board with an additional MCU and a firmware that is connected to an external bus (USB) of the host. This driver type is usually implemented on top of an OS bus driver. Examples for this class are diver for the CAN-USB/Mini, CAN-USB/Micro, CAN-USB/2 and CAN-USB/400 boards.
D	A user mode driver which handles a CAN board with an additional MCU and a firmware that is externally connected to the host without using directly an internal or external bus. This driver type is usually implemented a user mode driver on top of an OS protocol driver. Examples for this class are diver for the EtherCAN boards.

Table 3: Classes of Device Driver Implementations

Multiprocessor and multi-cores processor support

Device driver for operating systems which support more than one processor or core have been developed and tested to support this environment.

Plug & Play (P'n'P) and Hot Plugging support

In order to simplify the driver installation of devices for P'n'P capable buses (PCI, USB, ...) the mechanisms provided by the OS are supported to enumerate the already configured devices. For USB based interfaces hot plugging is supported.

Support for multiple CAN ports

Due to the device driver approach driver types of all classes can co-exist on one host and each driver is able to support simultaneously several CAN boards of its device class which have one or more physical CAN ports.

To make the underlying **esd electronics** CAN hardware transparent from the application point of view each physical port is assigned an individual logical net number in the range from 0 to 255. The details about the configuration of the logical net numbers are OS specific and described in the /1/.

As the link between a physical CAN port and the application is based on the logical net numbers it is possible to switch easily between different **esd electronics** CAN board types without the need to change the application.

Multitasking / Multithreading support

The NTCAN implementation is not limited to create just one link to a CAN port but allows creating several simultaneous links to the same port with different tasks/processes or even different threads of the same process. The support for this behaviour is based on NTCAN handles. Each handle virtualizes a CAN controller so the underlying physical CAN port can be used by several processes/threads at the same time.

If the host OS supports the handle concept it is used by the driver otherwise it is emulated.

2.3.2 Interaction

A standard CAN controller usually does not receive its own transmitted CAN messages. In many cases exactly this is necessary if e.g. there is a CAN based control application and you want to run in parallel a CAN monitor tool in another process on the same host using the identical physical CAN port. This feature as shown in the picture below is supported in all **esd electronics** CAN driver implementation and called ***Interaction***. Refer to chapter 3.5 for further details on the ***Interaction*** mechanism.

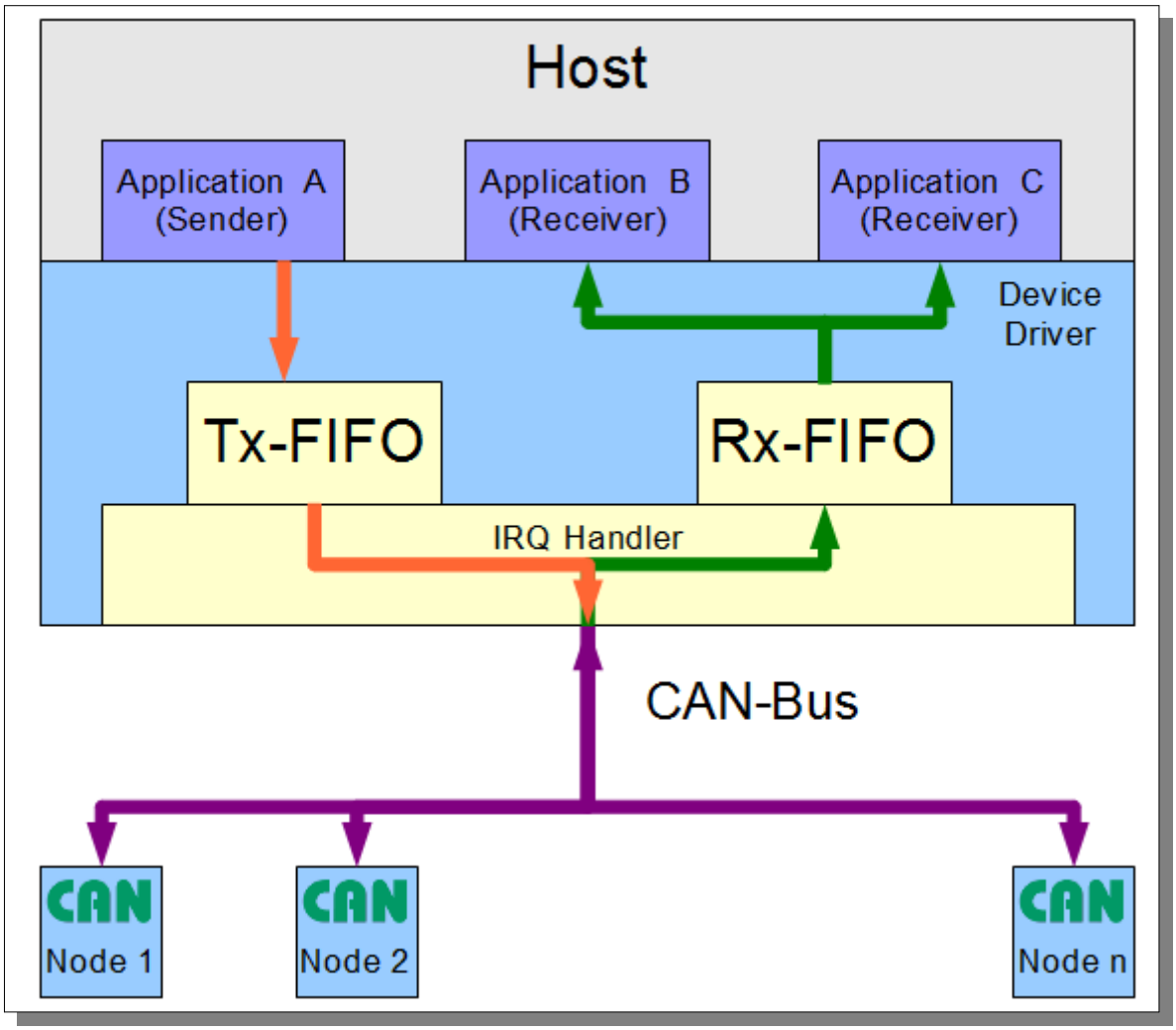


Figure 5: CAN Message Interaction

2.3.3 CAN Bit Rate Configuration

The NTCAN-API supports configuring the CAN bit rate for the **CAN CC** mode and the **CAN FD** mode in various ways to reach a maximum of flexibility (refer to chapter 3.3 for more details).

- Configuration based on the **esd electronics** bit rate table Table for the common standard bit rates which cover the recommended values of the CiA.
- Configuration based on the direct manipulation of the BTR registers of the CAN controller.
- Configuration based on numerical values and a built-in algorithm to calculate an optimal CAN controller configuration.

Listen Only Mode

This mode is intended for CAN bus monitoring without any influence on other CAN nodes (refer to chapter 3.3.2 for more details) or in combination with the baud rate detection mechanism described below.

Automatic bit rate detection

If the *Listen Only* mode described above is supported by the CAN board a device driver or the firmware can determine the bit rate of a physical CAN bus without affecting the CAN bus state in a destructive way. Afterward it can seamlessly join the communication on the bus. The *Automatic Bitrate Detection* is only supported in the Classic CAN mode and based on the following preconditions:

- There is an ongoing data exchange between at least two CAN nodes.
- The bit rate is among the bit rates which are part of the **esd electronics** bit rate table (refer to chapter 3.3.7 for more details).

Smart Disconnect

The default behavior of a device driver is to stay active on the CAN bus with the last configured bit rate, even if there is no application running, using this physical CAN port. This behavior is desired if you want to have an active CAN node connected to the bus, such as to acknowledge CAN messages.

Several device driver can be configured to leave the CAN bus if no application is using this physical CAN port (see chapter 3.3.8 for more details) which is called *Smart Disconnect* in this manual.

Automatic recovery from CAN controller state *Bus Off*

If a CAN controller received too many error frames, it changes into the state *Bus Off* and stops participating in the communication via CAN bus (refer to chapter 3.2 for more details). The device driver (passive CAN boards) or the firmware (active CAN boards) re-initializes the CAN controller automatically after a certain period of time. The application does not have to take special action for this but gets an indication about the state change.

2.3.4 Extended Features

Firmware Update

All active CAN boards with an MCU require a CAN firmware. It is usually possible to update this firmware with a dedicated firmware update tool in the field to add new features or to improve the performance.

Blocking and Non-Blocking CAN I/O

The NTCAN-API implements non-blocking I/O functions for reading and writing CAN data used by applications following the polling principle as well as blocking I/O functions for event-driven applications. The trade-off of the non-blocking calls which return immediately is the lack of immediate feedback in case of communication errors.

NTCAN Events

To indicate a state change or error situation to the application the NTCAN-API supports a mechanism which allows an application to receive such events in the same way and with identical API calls as CAN messages (refer to chapter 3.7 for more details). This allows an easy correlation of these events with the stream of CAN data.

Timestamp Support

The NTCAN-API supports a timestamp for received/transmitted CAN messages and NTCAN events. Depending on the CAN board this timestamp is either captured in software by the device driver in the interrupt handler or, more accurate, applied by the (active) CAN board (refer to chapter 3.9 for more details).

Scheduling

The driver supports scheduling of CAN messages. Transmission of frames can be initiated in a single-shot mode at a given point in time or cyclically. This feature is useful for residual bus simulation or generation of sync-frames.

Timestamped TX

Timestamps are supported in TX direction as well. This can be used to defer the transmission of CAN frames up to a given point in time without the need to set up special Scheduling objects. Please refer to chapter 3.14 for additional information.

Autoanswer

The CAN driver supports a fast automatically generated answer for a received CAN-RTR frame with a user configurable CAN frame. This feature is also supported for Basic CAN Controller without a limitation to the number of objects. For a given CAN-ID only one auto-answer object can be defined.

Extended Error Information

This feature provides additional detailed information about the state of the CAN bus, if the CAN controller supports a detailed error analysis. This comprises information about protocol errors, error counter, etc.

Refer to chapter 3.2 for further details.

Disable Automatic Retransmission (DAR) mode

This feature supports the possibility to disable the default behavior of a CAN controller to automatically repeat the transmission of a CAN message which has failed due to errors or a lost arbitration procedure. According to the CAN controller capabilities this can be configured globally or on a frame based basis. This behavior is often also referred to as *single-shot* or *one-shot* mode.

Refer to chapter 3.2 for further details.

Switchable Bus Termination

The CAN board allows to activate or deactivate a bus termination resistor programmatically.

Refer to chapter 3.16 for further details.

General Purpose Input/Output (GPIO) support

The CAN board supports up to 32 GPIO channels which are configured and controlled with NTCAN API.

Refer to chapter 3.17 for further details.

3. CAN Communication with NTCAN-API

This chapter provides an overview of the general functionality of the CAN communication with the NTCAN-API, before the API function calls, data types, etc. will be explained in greater detail in the following chapters.

3.1 Overview

An application which wants to access the CAN bus has to create a logical link to a physical CAN port with ***canOpen()***. This link is represented by an opaque handle of the data type `NTCAN_HANDLE` which is an input parameter for most NTCAN-API calls. A process can use multiple handles to the same or to different physical CAN ports simultaneously.

To distinguish between different physical CAN ports the device driver assigns a logical net number to each port and an application uses this number as an input parameter of ***canOpen()*** to reference the CAN port. This mechanism allows the use of CAN boards with more than one physical CAN port as well as the simultaneous use of several CAN boards of the same or different board type. The process of assigning different logical net numbers to physical CAN ports is hardware and operating system specific (please refer to /1/ for further details).

From the application point of view each handle references a virtual CAN controller with an individual set of properties for receive and transmit buffers as well as I/O timeouts. For each handle an individual receive filter for CAN messages in standard frame format (11-bit CAN identifier) can be configured with ***canIdAdd()*** and ***canIdDelete()*** together with an acceptance mask as a filter for CAN messages in extended frame format (29-bit CAN identifier). Since CAN driver revision 3.x even for 29-bit CAN identifier individual filter can be configured with ***canIdRegionAdd()*** and ***canIdRegionDelete()***. Additional filter criteria to receive e.g. no RTR frames can be applied with the help of the parameter *mode* of ***canOpen()***. Refer to chapter 3.8 for all details on acceptance filtering.

Before any CAN I/O can be performed the bit rate of the physical CAN port has to be configured once, either by using ***canSetBaudrate()*** to define a fixed bit rate (based on a table of common values, numerical values or CAN controller bit rate configuration register) or by starting the ‘automatic bit rate detection’ which is supported by many CAN boards. The configured value is valid for all handles opened with the logical network number of this physical CAN port. For CAN FD a 2nd bit rate for the data phase has to be defined which is done with ***canSetBaudrateX()***.

In order to prevent two applications from trying to initialise the same physical CAN port with different bit rates, the current configuration can be request with ***canGetBaudrate()*** or ***canGetBaudrateX()*** for CAN FD and any change will be indicated event driven. Refer to chapter 3.3 for all details on configuring the CAN bit rate.

For CAN message transmission the API offers blocking and non-blocking services. A call to ***canWrite()*** to transmit one or several CAN message(s) in the `CMSG` format will block the calling process or thread until all messages are transmitted successfully on the CAN bus, an I/O error occurred during transmission or the configured transmission timeout for the handle has expired (synchronous transmission). A call to ***canSend()*** will return immediately and make the device driver perform the transmission of messages in background (asynchronous transmission). The main difference between these two modes of operation is the feedback the application gets about the successful transmission.

In order to receive messages the API offers the blocking call ***canTake()*** and the non-blocking call ***canRead()***. 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 which passed the message filter of this handle have been received. The CAN driver can also insert meta information as CAN events into the stream of I/O data (see chapter 3.7 for details).

Optionally a 64-bit high resolution timestamp (see chapter 3.9) might be applied to each received CAN message. For this purpose the extended `CMSG_T` structure format is available and the receive message calls ***canTakeT()*** and ***canReadT().*** If an application wants to get the transmission time of a message it has to be received using *Interaction (see chapter 3.5)*. The timestamps applied to transmitted messages in combination with ***canSendT()*** and ***canWriteT()*** are used since driver version 3.x for message *Scheduling* (see chapter 3.12) or the deferred transmission of CAN messages (see chapter 3.14).

With the introduction of CAN FD and its extended payload of up to 64 bytes the `CMSG_X` structure is introduced as the universal timestamped message format with the related API calls ***canSendX()***, ***canWriteX(), canTakeX()*** and ***canReadX()*** to transmit respectively receive CAN FD as well as Classic CAN messages in a blocking or non-blocking way.

The general purpose API call ***canIoctl()*** is available to set or get further device and or/driver configuration options or to request any kind of (diagnostic) information.

In case of an error each API call returns a corresponding error code. The individual error codes will be explained in greater detail in chapter 7 and the convenience function ***canFormatError()*** is available to return a descriptive English error message.

3.2 CAN Errors and Fault Confinement

Because CAN nodes are able to distinguish between permanent failure and temporary disturbances, an automatic fault confinement is an integral part of the CAN protocol which makes it superior to other bus systems.

A CAN controller can distinguish between the following five types of errors within it's transmit or receive state machine.

Error Type	Description
Bit Error	A CAN controller sending a bit on the bus also monitors it comparing the bit levels (dominant or recessive) detected on the bus with the bit levels transmitted. A *Bit Error* is indicated in case of a difference. The are two allowed exceptions. No bit error is indicated sending a recessive bit and receiving a dominant bit during the arbitration field or the ACK slot and a controller transmitting a passive error flag (described later in this chapter) and receiving a dominant bit does not take this as a *Bit Error*.
Stuff Error	Whenever five consecutive dominant or recessive bits have to be transmitted, an extra complementary (stuff) bit is implicitly inserted into the bit stream as a necessary edge for clock resynchronization. Every receiver automatically removes these extra stuff bits. If a receiver state machine detects a sequence of more than five consecutive recessive or dominant bits during a message field that should be encoded by bit stuffing a *Stuff Error* is indicated.
CRC Error	A *CRC* (Cyclic Redundancy Check) *Error* is indicated by a receiver if the calculate CRC is different from the CRC received in the message.
Form Error	A *Form Error* is indicated by a receiver if a fixed form field contains illegal bits. For example, a fixed form field would be the control field which has two reserved bits (r1 and r0) that need to be sent out as two consecutive dominant bits.
ACK Error	An *Acknowledge Error* is indicated if no dominant bit is received during the ACK slot.

Table 4: CAN Communication Error Types

Any CAN controller transmitting or receiving CAN messages continuously checks the received CAN data for one of the five error types described in the table above. If an error is detected the discovering controller discards this message and an Error Flag is transmitted onto the CAN bus (according to the rules described later in this chapter) to signal the error situation. Other CAN controllers which have not (yet) detected the error themselves will discard the message because of this Error Flag so a system wide data consistency is guaranteed.

According to /2/, the default behaviour of a CAN controller whose transmission has failed due to errors (or a lost arbitration procedure) is to automatically retransmit this message. Optionally this behaviour can be disabled if the CAN controller supports the *Disable Automatic Retransmission* (DAR) mode which is also referred to as *Single-Shot* mode.

Detected errors are usually not indicated directly to the host CPU but only the change of the CAN controller state (described later in this chapter). For this purpose each CAN controller contains an 8-bit Transmit Error Counter (TEC) associated with the controller's transmit state machine and Receive Error Counter (REC) associated with the controller's receive state. The error counters are increased if any of the five error types described above is detected. For a successful transmission or reception the corresponding error counter is decreased. The complex rules for the increments and decrements are defined in /2/ but can be summarized in this simplified way:

- If an error is detected during reception, the REC is increased by 1. The error counter is increased by 8, if this error is not detected by other CAN nodes.
- The TEC is increased by 8, if an error is detected while the node is transmitting.
- After a successful reception the REC is decreased by 1
- After a successful transmission the TEC is decreased by 1.

With the help of this mechanism the error counters of a node will increment more rapidly if a fault is local to the node. Consequently permanent failures result in high counter values, whereas temporary disturbances result in small counts that recover back to zero in a running system. Depending on the value of its error counters the CAN controller is in one of the three states *Error Active*, *Error Passive* or *Bus Off* as shown in the picture below.

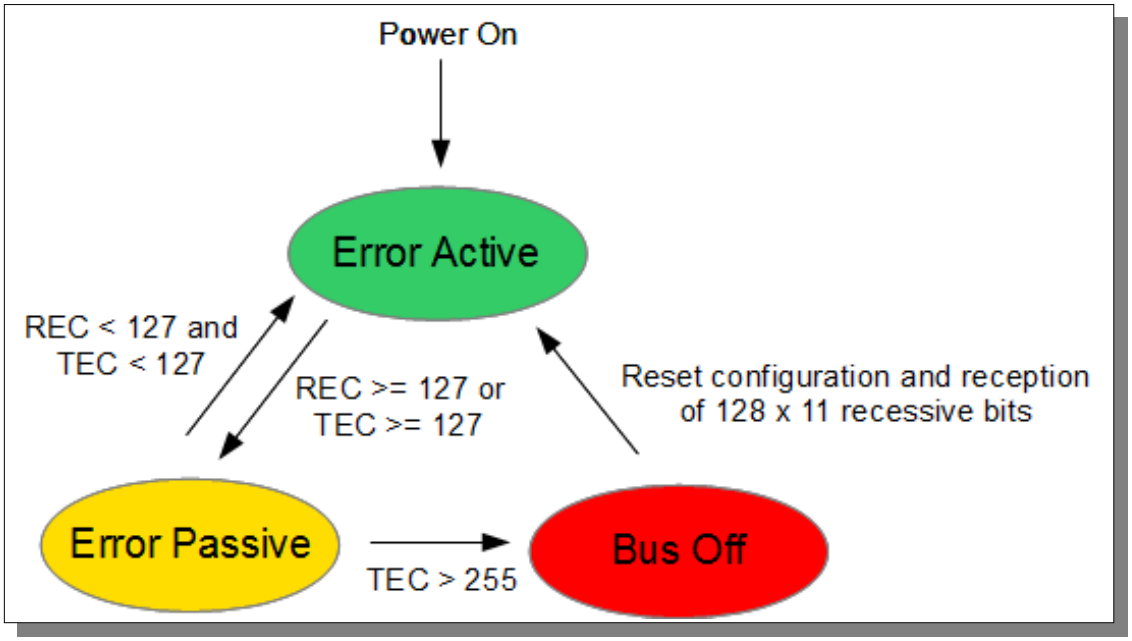


Figure 6: CAN Controller States

With the help of the error counters the CAN controller moves between different (error) states that allow a node to fail in a sophisticated way without blocking the bus as described in the table below.

Error state	Description
Error Active	Regular operational state of the node, with both counters 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 CAN-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 counter 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 CAN message. 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.

Table 5: CAN Controller Error States

A change of the CAN controller state is indicated by the event mechanism of the NTCAN-API (see chapter 3.7 for details).

3.3 Bit Rate Configuration

3.3.1 Overview

The physical layer of the CAN standard /2/ distinguishes between a **CAN CC** and a **CAN FD** enabled implementation of a CAN controller. A CAN CC controller or a CAN FD enabled controller which is operated in the CAN CC mode requires the configuration of a nominal bit rate up to 1000 Kbit/s and enforces with its arbitration and communication mechanism a dependency between configured bit rate and the maximum cable length. A CAN FD enabled controller which is operated in the CAN FD mode also requires the configuration of a further bit rate for the data phase (which has to be equal or higher than the nominal bit rate).

The basic physical structure of a CAN network requires that all nodes are configured to the same bit rate which has to be checked by the system integrator for any single CAN node. Configuring a wrong bit rate for a CAN port is a critical issue because all CAN nodes start with the error handling described in the previous chapter. The process of error detection and signaling only stops when the device with the incorrect bit rate goes into the bus-off state.

Bit timing is a complex issue because it's not just the bit rate itself but also the position of the sample point within the nominal bit time that matters. To increase the interoperability of CAN nodes from different vendors the *CAN in Automation* (CiA) has recommended /3/ some standard bit rates for CAN CC together with a detailed definition of the sample point which should be used in general purpose CAN bus networks. The bit timing configuration for CAN FD even adds some more complexity which is described in /7/.

The detailed register values to configure a given bit rate with a given sample point for a CAN controller are hardware specific and depend on the clock frequency of the CAN controller.

The NTCAN-API simplifies the task of configuring the bit rate by implementing a common bit rate table which is CAN controller and operating system independent. It follows the recommendations of the CiA for the CAN CC standard bit rates but also contains some intermediate as well as common higher layer CAN protocol bit rates.

CAN CC

An application calls ***canSetBaudrate()*** to configure the nominal bit rate and ***canGetBaudrate()*** to obtain the current configuration. Possible parameter for these calls are:

- An index of the esd electronics Nominal Bit Rate Table.
- A hardware (CAN controller) specific value in case the necessary bit rate is not covered by the bit rate table.
- A numerical bit rate value. In this case the driver calculates the necessary hardware specific configuration parameter.

The details of the bit rate configuration can be obtained by the application calling ***canIoctl()*** with `NTCAN_IOCTL_GET_BITRATE_DETAILS` as argument.

CAN FD

An application calls ***canSetBaudrateX()*** to configure the nominal as well as data bit rate and ***canGetBaudrateX()*** to obtain the current configuration. Possible parameter for these calls are:

- A supported combination of index values from the esd electronics Nominal Bit Rate Table and the esd electronics Data Phase Bit Rate Table.
- A bit timing register configuration in a hardware (CAN controller) specific format.
- A bit timing register configuration in a canonical format.
- A supported combination of numerical bit rate values.

The details of the bit rate configuration can be obtained by the application calling ***canIoctl()*** with `NTCAN_IOCTL_GET_BITRATE_DETAILS` as argument.

Related to a configured CAN FD bit rate in the data phase is the Transmitter Delay Compensation (TDC) mechanism and the position of a Second Sample Point (SSP) (refer to chapter 3.15 for further details) which is setup by NTCAN together with the bit rate.

The NTCAN architecture implements an intelligent automatic to setup the TDC and configure an optimal SSP, so usually an application specific deviating configuration is not required.

3.3.2 Listen-Only Mode

Many CAN controller support a so called *listen-only* mode which can be configured in combination with the bit rate calling ***canSetBaudrate()*** or ***canSetBaudrateX()***. In this operation mode the CAN controller would send neither an acknowledge nor an error frame on the CAN-bus but a message transmission is also not possible. This operation mode is ideally suited for monitoring the bus or for a nondestructive hot plugging of CAN nodes to an active bus.

Note that in both cases a physical layer interface must be available including CAN bus lines with a termination.
As not all CAN controller support the listen-only mode the application should check for the feature flag `NTCAN_FEATURE_LISTEN_ONLY_MODE` returned with ***canStatus()***.

3.3.3 Self Test Mode

Many CAN controllers support a self test mode which can be configured in combination with the bit rate calling ***canSetBaudrate()*** or ***canSetBaudrateX()***. The Self Test Mode allows internal transmission of CAN messages without the requirement for an acknowledge from other CAN nodes. This mode is ideally suited for a local self test or for stand alone system development and testing still keeping the timing constraints of the configured bit rate.

Note that depending on the implementation of this feature in the CAN controller also for this loop back mode a physical layer interface must be available which might require a proper termination.

As not all CAN controllers support a self test mode the application should check for the feature flag `NTCAN_FEATURE_SELF_TEST`.

3.3.4 Triple Sampling Mode

The default behavior of a CAN controller is to sample the bus once per bit at the configured sample point (SP). Some CAN controller optionally implement a mode where the bus is sampled three times per bit with a majority logic to determine the bit value. The latter mode is usually recommended for low/medium speed to filter spikes.

As not all CAN controller support a triple sampling mode the application should check for the feature flag `NTCAN_FEATURE_TRIPLE_SAMPLING` returned with ***canIoctl()***.

3.3.5 Transmit Pause

The default behavior of a CAN controller is to start the next data transmission immediately after it's previous transmission was completed successfully considering the bus idle time required by the CAN protocol. Especially with modern CAN controller types which are capable to generate a one hundred percent bus utilization a node which needs to transmit CAN messages with a worse arbitration priority may starve.

To overcome this situation a device with an active transmit pause will wait for a controller number of additional bit times after a successful transmission so other nodes are able to transmit their messages with a worse arbitration because they start the next messages arbitration earlier.

Enabling a transmit pause will mitigate situations where the application on a single node performs burst transmissions and protects against a "babbling idiot" error situation.

As not all CAN controller support a transmit pause mode the application should check for the feature flag `NTCAN_FEATURE_TX_PAUSE` returned with ***canIoctl()***.

3.3.6 Disable Automatic Retransmission (DAR) Mode

The default behaviour of a CAN controller is to repeat immediately a transmission which has failed due to errors or a lost arbitration procedure. Although CAN messages can be sent with a very small timeout it is not possible to limit the number of retransmissions to deterministic number.

To overcome this situation some CAN controller support an operation mode which disables this retransmission globally so CAN frames are either transmitted successfully on the 1st attempt or not transmitted at all and repeating the transmission must be requested by the application.

Disabling the automatic retransmission is required to support a time-triggered communication as described in /2/.

As not all CAN controller support a DAR mode the application should check for the feature flag `NTCAN_FEATURE_DAR` and/or `NTCAN_FEATURE_DAR_FRAME` returned with ***canIoctl()***.

Standard CAN controller with DAR support do not distinguish between a transmission failure caused by an arbitration lost situation or a CAN transmission error. The ESDACC allows to configure the bevaiour per physical CAN individually for these two failure situations via ***canIoctl()***.

3.3.7 Automatic Bit Rate Detection

Most device driver for CAN controller with the listen-only mode support also support an automatic bit rate detection for the CAN CC mode. This mode is initialized with ***canSetBaudrate()*** or ***canSetBaudrateX()*** in the same way a fixed bit rate is set. The device driver continuously monitors the CAN bus in a nondestructive way while automatically switching the nominal bit rate until a valid CAN message is received. In the end the application can seamlessly join the communication on the bus. The implementation follows the CiA recommendation /4/ for automatic bit-rate detection.

A successful automatic bit rate detection is only possible if there is already communication on the CAN bus with a bit rate which is part of the NTCAN bit rate table and is only supported in the CAN CC operation mode.

In order to follow the bit rate detection process an application can either poll the current state with ***canGetBaudrate()*** or can wait for a bit rate change event.

3.3.8 Smart Disconnect

The default behavior of the device driver is to leave the CAN controller active on the bus with the last configured bit rate, even if no application has an open handle to the physical CAN port any more. This behavior is desired if you want to have an active CAN node connected to the bus, e.g. to acknowledge CAN messages.

Some drivers support disabling the CAN controller automatically as soon as the last handle to this physical CAN port is closed. The configuration of this *Smart Disconnect* behavior is set during driver initialization and can not be changed during run-time. The configuration method is operating system dependent and is described in the driver installation manual /1/.

An application can check if the *Smart Disconnect* mode is supported and enabled with the feature flag `NTCAN_FEATURE_SMART_DISCONNECT` returned with ***canIoctl()***.

3.4 NTCAN-ID and Structures of Data Exchange

The data exchange between the application and NTCAN library is based on (arrays of) equal sized messages. The main components of each CAN message structure are the

- The NTCAN-ID (4 Bytes)
- The data length (1 Byte)
- The payload (CAN operation mode dependent)

The NTCAN-ID is used to distinguish between CAN messages and NTCAN Events as shown in the figure below:

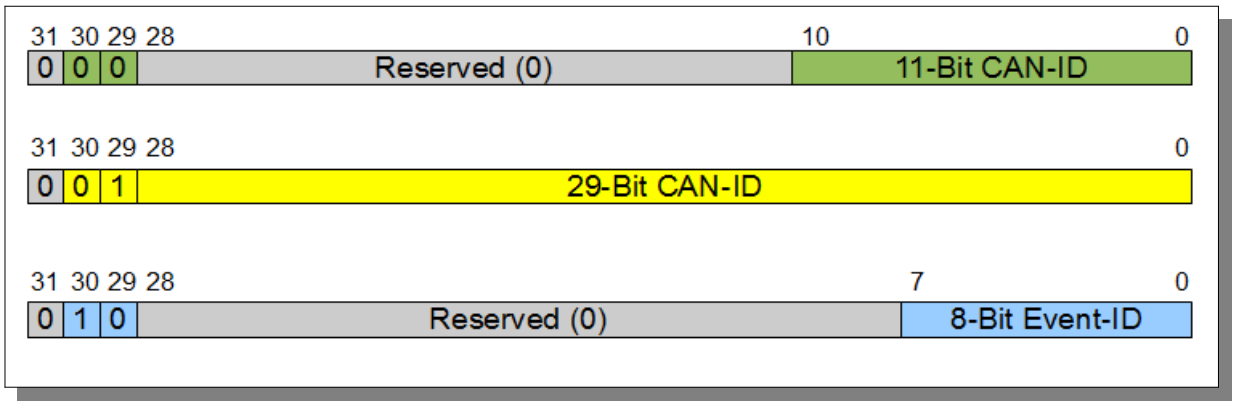


Figure 7: NTCAN-ID layout

Bit 30 is used to distinguish between CAN messages (`CMSG` structures) and NTCAN Events (`EVMSG` structures) which are identical with respect to data size and layout. Thus they can be mixed in related NTCAN-API I/O calls.
For an NTCAN Event (Bit 30 set to '1') the bits 0..7 contain the Event-ID and the bits 8..29 are reserved for future use and should be set to '0'.
For a CAN message (Bit 30 set to '0') the bit 29 of the NTCAN-ID is used to distinguish between 11-bit (SFF) and 29-bit (EFF) CAN messages. In case of a 11-bit CAN message (Bit 29 set to '0') the bits 0..10 contain the CAN-ID of the message and the bits 8..28 are reserved for future use and should be set to '0'. In case of a 29-bit CAN message (Bit 29 set to '1') the bits 0..28 contain the CAN-ID.
The Bit 31 of the NTCAN-ID is reserved for future use and should always be set to '0'.

The lower 4 bit of the data length are the Data Length Code (DLC) of the CAN message or the NTCAN event which indicates the valid data bytes in the payload part. The upper 4 bits are used for additional meta information (e.g. if a CAN message is a data frame or a RTR frame).

For the **CAN CC** mode the payload part contains up to 8 bytes of CAN or event data which requires a structure size of 16 bytes.

For timestamped I/O in **CAN CC** mode the data structures described above are extended with a timestamp (8 bytes) defined in the NTCAN-API as data structures of the type `CMSG_T` and `EVMSG_T`. The timestamp increases the structure size to 24 Bytes.

For the **CAN FD** mode the payload part contains up to 64 bytes of CAN data and always a timestamp. The NTCAN-API defines data structures of the type `CMSG_X` and `EVMSG_X` for this. The additional payload increases the structure size to 80 Bytes.

Because of this size difference an own set of CAN I/O API functions is defined for each structure.

3.5 Interaction

If an application transmits a CAN message as described in chapter 3.10.2 a CAN controller usually does not receive its own transmitted message. As in a multitasking/multithreading environment it's often required by the application logic or it's at least convenient to receive the CAN messages transmitted by another task/thread, the **esd** CAN driver implements a feature called *Interaction* which is illustrated in figure 5. The driver implements this feature in a way that an interaction message is **NOT** passed via Interprocess Communication (IPC) mechanisms of the operating system as shown in figure 8, but is related to the successful transmission of the message on the CAN bus by the driver as shown in figure 9.

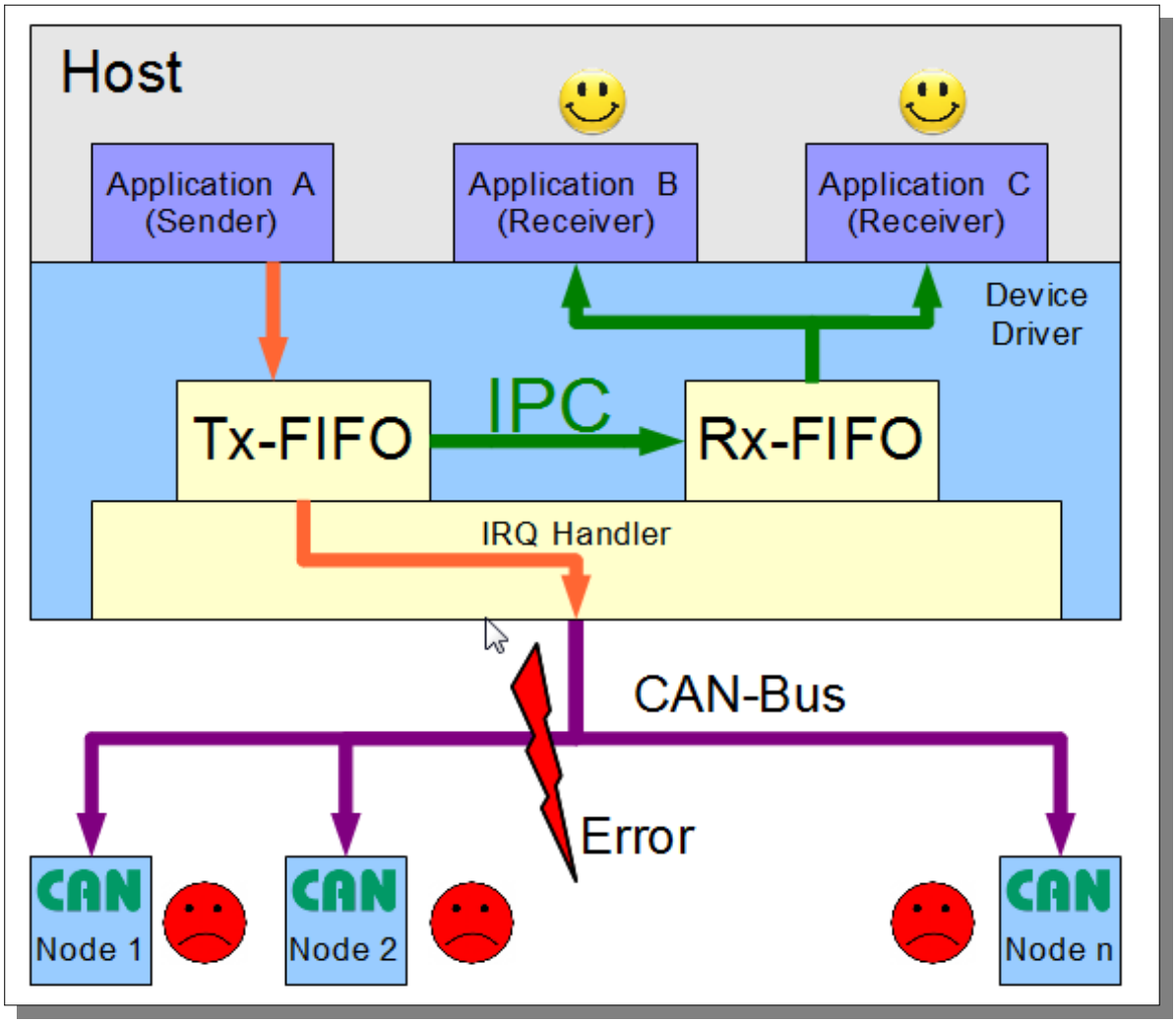


Figure 8: Interaction with IPC

If the interaction would be implemented with IPC mechanisms a local receiver would get a CAN message even if the physical transmission on the CAN bus failed (as show in the picture above) which is an error.
The **esd electronics** driver implementation instead is based on a successful transmission as shown in the figure below. In case of a transmission error local receivers will not get the message, too.

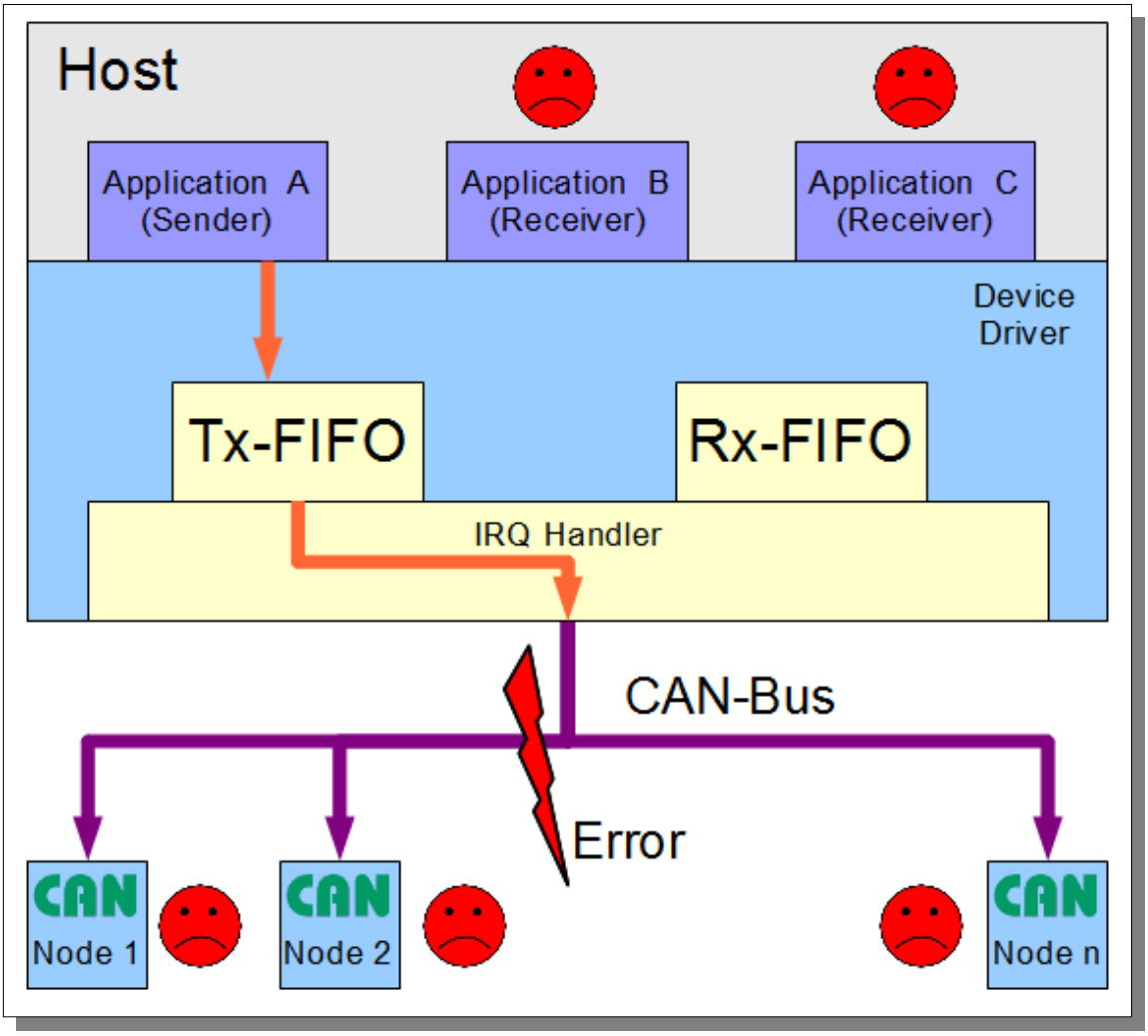


Figure 9: Interaction without IPC

In addition the this implementations guarantees that local receivers get the data at the same time as all other CAN devices connected to this bus. This is important as otherwise the order of received messages, their timing relation and/or their timestamps would be wrong.

The trade-off of the implementation is that ***Interaction*** demands the CAN message to be physically transmitted which requires a physical working CAN bus with at least one additional CAN node.

The default of the ***Interaction*** behaviour is to be enabled for all handles but the handle which transmits the message which follows again the concept of a complete virtual CAN controller (that usually does not receive its own transmitted message, too).

An application can configure the interaction behaviour on a per handle basis in many ways:

- If the application logic does not require or allow CAN messages received via the ***Interaction*** mechanism the CAN handle can be opened with the mode flag `NTCAN_MODE_NO_INTERACTION` so the these messages do not pass the acceptance filter (see chapter 3.8.1 for details).
- If the application logic requires to distinguish CAN messages received via the ***Interaction*** mechanism from CAN messages received on the CAN bus from other CAN devices the CAN handle has to be opened with the mode flag `NTCAN_MODE_MARK_INTERACTION`. All messages received via ***Interaction*** will now have set the `NTCAN_INTERACTION` flag in the length field of the `CMSG` or `CMSG_T`.
- If the application logic requires that transmitted CAN messages are received on the same handle with the ***Interaction*** mechanism this handle has to be opened with the mode flag `NTCAN_MODE_LOCAL_ECHO` to have the same interaction behaviour as other handles.

3.6 Bus Diagnostic

The NTCAN Bus Diagnostic is subdivided into a basic diagnostic support and an extended diagnostic support.

The Extended Bus Diagnostic is not available for every CAN board as it is very CAN controller dependent. An application should check for the feature flag `NTCAN_FEATURE_DIAGNOSTIC` returned with ***canStatus()***.

3.6.1 Basic Support

The basic diagnostic support is available for all NTCAN implementations and is realized as the NTCAN event (see chapter 3.7.1) `NTCAN_EV_CAN_ERROR` in case of CAN controller state changes or CAN I/O overruns (refer to 6.2.9 for more details).

3.6.2 Extended Support

The extended diagnostic support covers event based indications as well as polled diagnostic data.

- Any change of the configured bit rate is indicated with the NTCAN event `NTCAN_EV_BAUD_CHANGE` (refer to chapter 6.2.8 for details).

- In addition to the basic support the NTCAN event `NTCAN_EV_CAN_ERROR_EXT` is indicated every time the CAN controller detects an error on the bus (refer to chapter 3.7.1 and 6.2.10 for more details). The event contains CAN controller dependent information about the error reason and the current values of the controller's TEC and REC. With the help of ***canFormatEvent()*** a textual description of the error can be received.

- The periodic NTCAN event `NTCAN_EV_BUSLOAD` can be configured to indicate the current number of received bits for this CAN port. The event is generated as soon as it is enabled with ***canIdAdd()***. The cycle time of the event can be set/requested with ***canIoctl()*** and the argument `NTCAN_IOCTL_SET_BUSLOAD_INTERVAL/NTCAN_IOCTL_GET_BUSLOAD_INTERVAL`. With the help of two (timestamped) events and the configured bitrate the CAN busload can be calculated in the application or with the help of ***canFormatEvent()***.

- The actual CAN controller state with the controller's TEC and REC is returned by calling ***canIoctl()*** with the command `NTCAN_IOCTL_GET_CTRL_STATUS` (refer to chapter 6.2.19 for details).

- The actual CAN bus statistic for a CAN port can be requested by calling ***canIoctl()*** with the command `NTCAN_IOCTL_GET_BUS_STATISTIC` (refer to chapter 6.2.18 for details).

3.7 NTCAN Events

In addition to error return values and API calls for status and bus diagnostic information, the CAN driver can indicate errors and other state changes, e.g. a change of the CAN controller into *Bus Off* state has occurred, asynchronously with an event mechanism. These events can be received by the application with a separate handle but their main advantage is that they are embedded in the stream of received CAN data so they can be set in a temporal relation to the CAN bus activity. NTCAN events are returned in the data structure `EVMSG`.

NTCAN events can only be received in FIFO mode. There is no support to receive NTCAN events if the handle is opened in object mode.

If timestamps are supported, the NTCAN events can also be returned in the timestamped data structure `EVMSG_T` or `EVMSG_X`.

3.7.1 Event types

A valid NTCAN Event-ID is in the range from `NTCAN_EV_BASE` to `NTCAN_EV_LAST`. The table below shows the supported events and the event specific data types which are embedded in the data part of the `EVMSG`, `EVMSG_T` or `EVMSG_X`. All other Event-IDs are reserved for future use.

Depending on the Event-ID the event can be received, transmitted or both with a direction specific payload length for the event as defined in the table below. If an event can be transmitted **and** received, the transmission of the event is an internal trigger for the device driver to generate the event.

Event-ID	Data Type	Length	Description
`NTCAN_EV_CAN_ERROR`	`EV_CAN_ERROR`	Tx: N/A Rx: 6	General bus diagnostic data.
`NTCAN_EV_BAUD_CHANGE`	`EV_BAUD_CHANGE`	Tx: N/A Rx: 4/8	Bit rate change event. For a CAN FD configuration the application will receive separate events for the nominal bit rate and the data bit rate.
`NTCAN_EV_CAN_ERROR_EXT`	`EV_CAN_ERROR_EXT`	Tx: N/A Rx: 4/5	CAN controller specific diagnostic data.
`NTCAN_EV_BUSLOAD`	`uint64_t`	Tx: N/A Rx: 8	Cyclically transmitted busload event.
`NTCAN_EV_GPIO_SET_DIR`	`EV_GPIO_DATA`	Tx: 8 Rx: N/A	Set the direction of the GPIO channels.
`NTCAN_EV_GPIO_SET_DO`	`EV_GPIO_DATA`	Tx: 8 Rx: N/A	Set the state of the GPIO channels configured as digital outputs.
`NTCAN_EV_GPIO_GET_DO`	`EV_GPIO_DATA`	Tx: 0 Rx: 4	Get the state of the GPIO channels configured as digital outputs. Note: This event must be triggered.
`NTCAN_EV_GPIO_GET_DI`	`EV_GPIO_DATA`	Tx: 0 Rx: 4/8	Get the state of the GPIO channels configured as digital inputs. Note: This event can be triggered.

Table 6: NTCAN events

Only the `NTCAN_EV_CAN_ERROR` event is guaranteed to be supported on all platforms.

The NTCAN-API exports the function ***canFormatEvent()*** to return a textual evaluation of an NTCAN event in English for the CAN bus diagnostic related events.

The application which processes the event has to evaluate the length information of the event message. If this size can vary according to Table 6 and is less than the size of the payload data type, only the part of the data which is reflected by this length information is valid.

3.7.2 Reception

1. Create a `NTCAN_HANDLE` with ***canOpen()*** or use an exiting handle (FIFO mode only).

2. Configure the message filter using ***canIdAdd()***. The application has to enable the Event-Ids of interest.

3. Receive the events

   Use ***canRead()*** or ***canTake()*** in the same way as described in chapter 3.10 to receive CAN messages in FIFO mode. If for this CAN handle 11- or 29-bit CAN-IDs are enabled in addition to the Event-Ids, you need to distinguish between received CAN frames and events by evaluating bit 30 of the (CAN) identifier. If this bit is set, cast the type of the data structure from `CMSG` to `EVMSG` to process the received message as an NTCAN event.

   If timestamps are supported, you can use ***canReadT()*** or ***canTakeT()*** to receive messages of the type `EVMSG_T`.

The NTCAN API also exports a still working entry named ***canReadEvent()*** for backward compatibility. This call is deprecated although it allows to receive the NTCAN events without a cast for the following reasons:

- It covers the same functionality as ***canRead().***
- Events can only be received one at a time.
- Events can not be related temporally to CAN messages.
- There is no version with timestamp support.

Do not use the canReadEvent() API for new applications.

3.7.3 Trigger

The generation of some events can actively be triggered by the application. These events

1. Create a `NTCAN_HANDLE` with ***canOpen()*** or use an exiting handle (FIFO mode only).

2. Trigger the event

   Use ***canWrite()*** or ***canSend()*** in the same way as described in chapter 3.10 to transmit CAN messages in FIFO mode.

The NTCAN API also exports a still working entry named ***canSendEvent()*** for backward compatibility. This call is deprecated although it allows to transmit the NTCAN events without a cast for the following reasons:

- It covers the same functionality as ***canWrite().***
- Events can only be transmitted one at a time.

Do not use the canSendEvent() API for new applications.

3.8 Acceptance Filtering

In order to receive CAN messages the application has to define an acceptance filter for the handle. The NTCAN-API implements sophisticated acceptance filtering mechanisms which can be defined individually for each handle based on the

- CAN message type
- CAN message identifier

The configuration of the acceptance filter has no influence on the transmission of CAN messages with this CAN handle.

The advantage of this implementation is that in most cases no further acceptance filtering within the application is required. It can be handled completely in a much more efficient way within the device driver or even in hardware which reduces the overall system load.

In chapter 3.8.2.3 you will find the complete process of NTCAN acceptance filtering in flow chart form.

3.8.1 Message Type Filter

During handle creation with ***canOpen()*** an individual message type filter can be defined. With the help of this filter type the application can selectively prevent the reception of

- CAN Data Frames
- CAN RTR Frames
- CAN Interaction Frames

or a combination of them. The message type filter can not be changed at runtime and without any filter configuration all message types pass the filter.

3.8.2 Basic ID Filter

An application can define for each handle an individual *Basic ID Filter* (BIF) based on the NTCAN-IDs which should pass the filter for this handle. The BIF which is implemented in CAN device drivers before V 3.9.x implements a table based two-stage filter mechanisms with a first stage solely dedicated to CAN-IDs in the *Base Frame Format* (11-bit identifier) and Event-IDs and a 2nd stage dedicated to CAN-IDs in the *Extended Frame Format* (29-bit identifier).

There are two implementation aspects which are worth mentioning:

- The filter is applied with constant time independent from the filter configuration.
- The filter can be modified at runtime while CAN data is received.

3.8.2.1 First Filter Stage

As a first filter stage the application can define for each handle an individual set of 11-bit CAN-IDs and Event-IDs which should pass the acceptance filter by adding and removing identifier to this set with ***canIdAdd()*** and ***canIdDelete()***. After the handle is created with ***canOpen()*** no NTCAN-ID will pass the filter.

3.8.2.2 Second Filter Stage

As soon as an arbitrary 29-bit CAN-ID is enabled with ***canIdAdd()*** **all** 29-bit CAN-IDs will pass the first filter stage.

For acceptance filtering of 29-bit CAN-IDs the CAN driver implements a 2nd filter stage with a mechanism based on a logical combination of an acceptance code with an acceptance mask. Such a mechanism is implemented by many CAN controller in hardware but the NTCAN-API allows to define an individual filter 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.

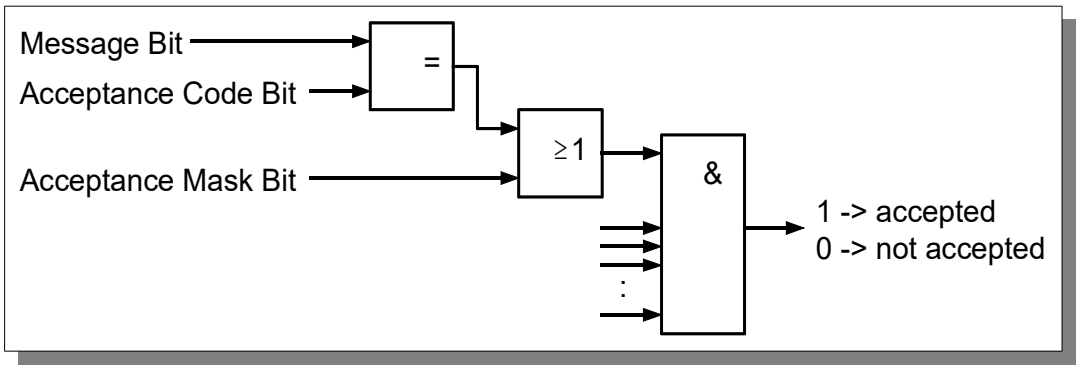


Figure 10: Mask based acceptance filtering of CAN-IDs

The acceptance code is the last 29-bit CAN-ID enabled for this handle with ***canIdAdd()*** and the acceptance mask is defined with the command `NTCAN_IOCTL_SET_20B_HND_FILTER` for ***canIoctl().***

Figure 10 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 the resulting filter behaviour.

ACR	AMR	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**

Table 7: Acceptance filter examples for 29-bit CAN-IDs

After handle creation with ***canOpen()*** the acceptance mask is configured as shown in the last row of the table above.

3.8.2.3 Flow Chart

The figure below gives a complete overview on all stages of acceptance filtering with the BIF for a received CAN frame according to its message type and its ID.

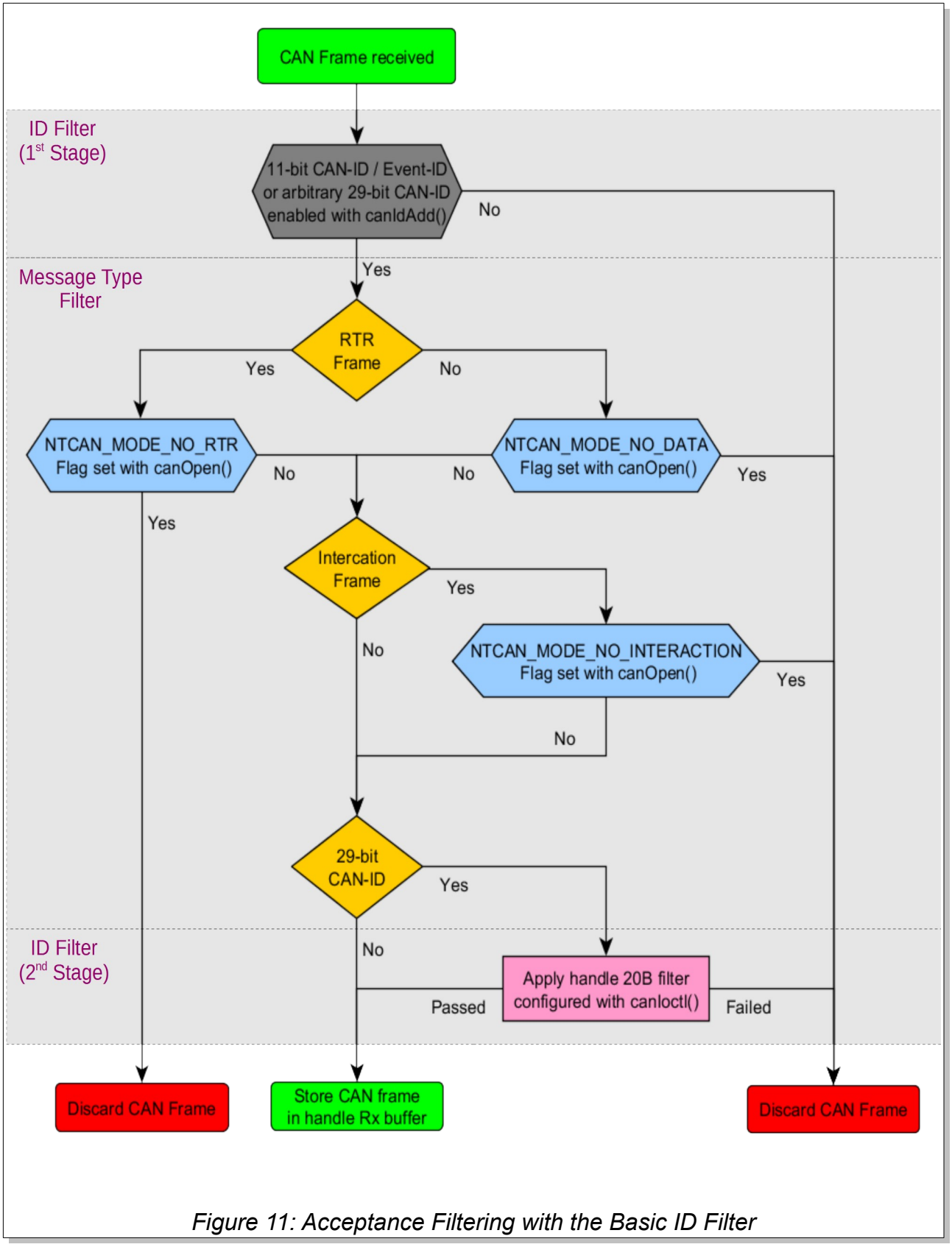


Figure 11: Acceptance Filtering with the Basic ID Filter

3.8.3 Smart ID Filter

The *Smart ID Filter* (SIF) is similar to the *Basic ID Filter* (BIF), described in the previous chapter, a two-stage filter. But it is a completely revised implementation with two major improvements:

- The first filter stage is no longer limited to 11-bit CAN-Ids and NTCAN Events but also supports 29-Bit CAN-IDs.
- The second filter stage is no longer limited to 29-Bit CAN-IDs but implements three independent filter for 29-Bit CAN-Ids, 11-bit CAN-IDs and Event-IDs.

The *Smart ID Filter* is not supported by CAN device drivers before V3.9.x. An application should check for the feature flag `NTCAN_FEATURE_SMART_ID_FILTER` returned with ***canStatus()***.

Due to the sophisticated implementation as a multi-level tree-structured table the filter is still applied with constant time independent from it's configuration and can be changed at runtime.

Drivers with the *Smart ID Filter* are fully binary backward compatible to drivers with the *Basic ID Filter* with respect to the API as well as the filter behaviour.

3.8.3.1 First Filter Stage

The CAN-IDs and Event-IDs for the first filter stage are enabled or disabled with ***canIdRegionAdd()*** and ***canIdRegionDelete().*** The legacy API calls ***canIdAdd()*** and ***canIdDelete()*** (which are internally mapped to the new API calls) can still be used in parallel.

3.8.3.2 Second Filter Stage

The three masks of the second filter stage are configured with the command `NTCAN_IOCTL_SET_HND_FILTER` for ***canIoctl()*** and an initialized `NTCAN_FILTER_MASK` structure as argument. Figure 10 shows how an active bit of the Acceptance Mask Register (AMR) results in a don't care condition for the result of the comparison between the bit in the received ID and the related bit in the Acceptance Code Register (ACR).

The resource (memory) requirement of a SIF configuration depends on the implementation of the first and second stage. As a rule of the thumb it is often worthwhile to filter as much as possible with the second stage and to keep the consecutive areas of the first stage as wide as possible.

Example:

Definition of a SIF for all odd 29-bit CAN-IDs in the range from 0x20001111 to 0x200FFFFFF (which can not be configured with the BIF).

Solution 1 (with just the first filter stage):

```
int32_t id, count;
for(id=0x20001111; id <= 0x200FFFFF; id += 2) {
   count = 1;
   canIdRegionAdd(hnd, id, &count);
}
```

Solution 2 (with a combination of first and second filter stage):

```
int32_t count = FEEEF;
canIdRegionAdd( hnd, 0x20001111, &count);
filter.acr = 0x00000001;
filter.amr = 0xFFFFFFFE;
filter.idArea = NTCAN_IDS_REGION_20B;
canIoctl(hnd, NTCAN_IOCTL_SET_HND_FILTER, &filter)
```

→ The resource usage of solution 2 is far smaller than that of solution 1.

3.8.3.3 Flow Chart

The figure below gives a complete overview on all stages of acceptance filtering with the BIF for a received CAN frame according to its message type and its ID.

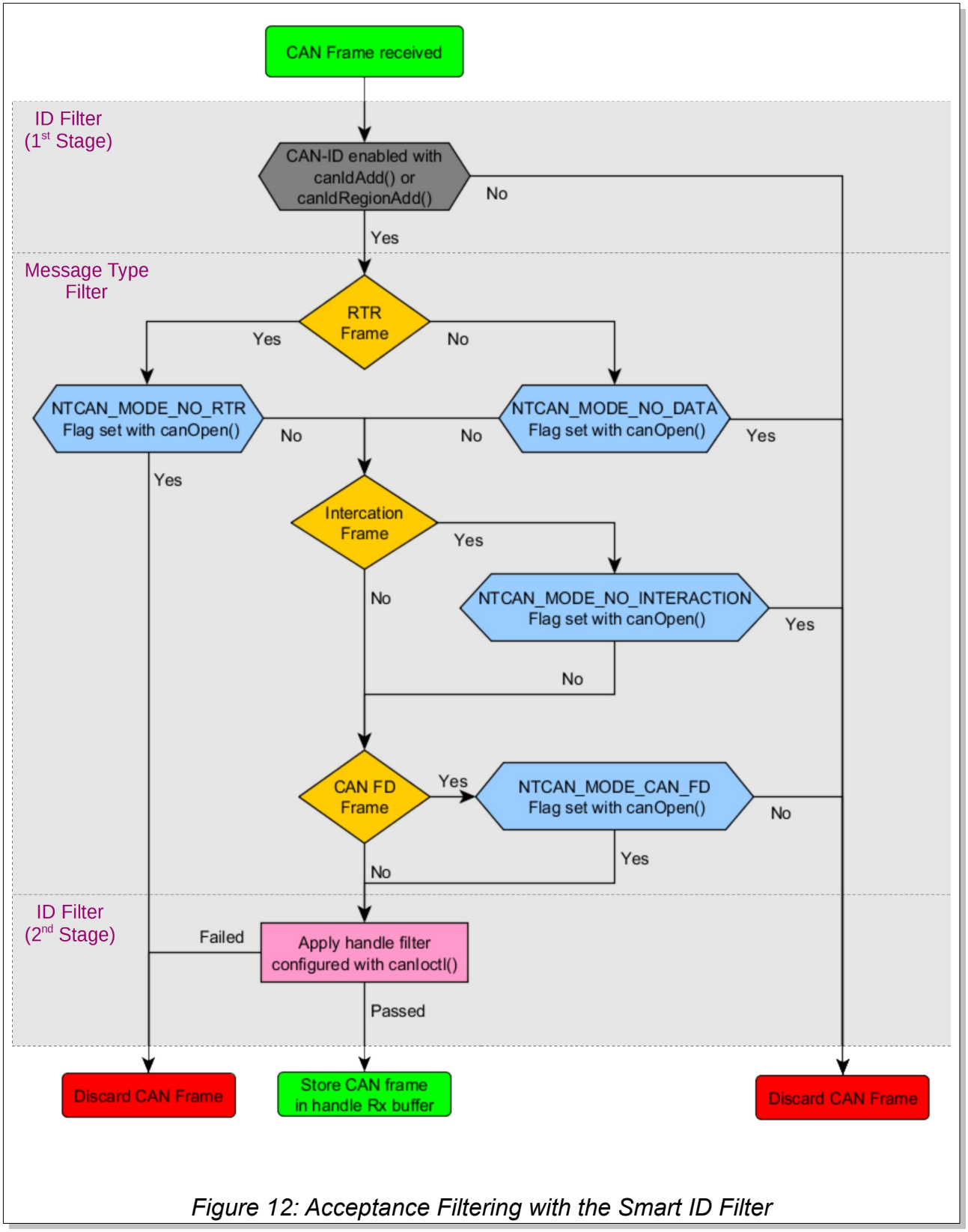


Figure 12: Acceptance Filtering with the Smart ID Filter

3.9 Timestamps

Most **esd electronics** CAN boards support capturing the time stamp of the moment a CAN message was received or a CAN event occurred. Depending on the device capabilities the time stamping is performed either in hardware by the CAN board or in software by the driver's interrupt handler using a high resolution counter of the host CPU.

Hardware timestamps are supported by most of the active **esd electronics** CAN boards and the FPGA based *Advanced CAN Core*. Hardware timestamps usually result in a higher accuracy compared to software timestamps as the jitter does not depend on the real-time capabilities or the CPU load of the host system.

3.9.1 Implementation

Timestamped CAN messages are received in *FIFO Mode* as well as in *Object Mode* using ***canReadT()*** / ***canTakeT()*** or ***canReadX()*** / ***canTakeX()*** in the same way as described in the chapters 3.10 and 3.11 for ***canRead()*** and ***canTake()***. The argument to store the CAN messages are of the type `CMSG_T` respectively `CMSG_X` to apply the additional timestamp. The latter structures are also used for Timestamp Tx mode described in chapter 3.14 using ***canSendT()*** / ***canSendX()*** or ***canWriteT()*** / ***canWriteX().*** The same timestamp is used in the `CSCHED` structure to define the start time and interval for the scheduling of CAN messages described in chapter 3.12.1.

An application can check if timestamps are supported with the feature flag `NTCAN_FEATURE_TIMESTAMP` returned with ***canStatus()***.

The current timestamp can be requested at any time calling ***canIoctl()*** with the `NTCAN_IOCTL_GET_TIMESTAMP` command. As especially reading a small amount of data directly from hardware is a performance bottleneck the timestamp returned for this command for ESDACC based devices is a virtual software timestamp which is synchronized with the hardware timestamp of the ESDACC. If the accuracy of this timestamp is not sufficient the ESDACC hardware timestamp can be requested at any time calling ***canIoctl()*** with the `NTCAN_IOCTL_GET_HW_TIMESTAMP` or `NTCAN_IOCTL_GET_HW_TIMESTAMP_EX` command.

A timestamp has 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 in order to scale the timestamps online or offline and can query the current timestamp to link them to the absolute system time.

3.9.2 Usage

In this chapter the typical steps to use of the timestamp interface (in FIFO Mode) are summarized:

1. Open CAN handle with ***canOpen()*** in *FIFO Mode*.

2. Check if timestamps are supported via the feature flag `NTCAN_FEATURE_TIMESTAMP` returned with ***canStatus()***.

3. Somewhere at the beginning of your application request once the following information:
   - The frequency of the timestamp counter (which is specific for any **esd electronics** CAN board and/or host OS). This can be accomplished by calling ***canIoctl()*** with the `NTCAN_IOCTL_GET_TIMESTAMP_FREQ` command.
   - The current value of the timestamp counter, in order to correlate received timestamps with your system time. This can be accomplished by calling ***canIoctl()*** with the `NTCAN_IOCTL_GET_TIMESTAMP` command.

4. Set the bit rate with ***canSetBaudrate()*** for the physical CAN port.

5. Configure the message filter using ***canIdAdd()***.

6. Use ***canReadT()*** or ***canTakeT()*** instead of ***canRead()*** or ***canTake()*** to receive CAN frames and scale the timestamp with the help of the timestamp frequency determined in step 3 to your application specific time base.

3.10 FIFO Mode

The CAN communication with the NTCAN-API is based on message queues which are implemented as First-In-First-Out (FIFO) buffers. They contain one or more CAN messages and can be used for event driven I/O as well as polled I/O. For the event driven CAN communication this is the only supported mode of operation.

3.10.1 Overview

Each handle is assigned a separate receive and transmit FIFO whose size is defined independently from each other when the handle is created with ***canOpen()*** and can not be changed later on.

In the Rx-FIFO CAN messages are stored in the chronological order of their reception. By calling a read operation one or more CAN messages are copied from the handle Rx-FIFO into the application buffer. By calling a write operation one or more CAN messages are copied from application buffer into the handle Tx-FIFO and the CAN messages are transmitted on the CAN bus in their chronological order.

The blocking ***canRead()/canReadT()*** call returns with new data until the receive FIFO is empty. In this case the calling thread blocks and returns immediately as soon as a new CAN message is available or returns with an error if the configured receive timeout is expired or the request is aborted by the application. If the Rx-FIFO gets overrun by the driver because the application does not process the CAN messages fast enough, the oldest CAN message is overwritten and this error is indicated to the application. The non-blocking ***canTake()/canTakeT()*** operates identical to ***canRead()/canReadT()*** but always returns immediately independent of the availability new CAN messages.

The blocking ***canWrite()/canWriteT()*** 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, in case of an I/O error or the request is aborted by the application. If the number of CAN messages to be transmitted exceeds the current Tx-FIFO capacity only the number of messages which fit into the FIFO are transmitted. The application has to verify the return values to handle this situation. The non-blocking ***canSend()/canSendT()*** call operates identical to ***canWrite()/canWriteT()*** but always returns immediately and the CAN messages are transmitted asynchronously to the calling thread.

If the application does not want to make use of the Rx-FIFO, the FIFO size can be configured to 1. This means that always the most recent CAN message it stored in the Rx-FIFO.

3.10.2 Reception and Transmission of CAN-Frames

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

   Configure individual timeouts and FIFO sizes for reception and transmission of CAN messages. On success a handle which is linked to a physical CAN port is returned.

   The configured timeouts can be modified later on by using the appropriate ***canIoctl()*** command but the FIFO sizes are immutable.

2. Set the baud rate with ***canSetBaudrate()*** for the physical CAN port.

   This might already be done by another thread or process. Thus you are advised to check the baud rate of the CAN-bus in advance with ***canGetBaudrate()***.

3. Configure the message filter using ***canIdAdd()***.

   If you want to receive CAN messages, you need to add at least one CAN-ID to the handle message filter, for transmission this step is not necessary.

4.1. Reception of CAN frames:

   Use ***canRead()/canReadT()*** or ***canTake()/canTakeT()*** to process CAN messages, which were received from the CAN-bus and passed your configured message filter.

   Either call is able to retrieve several CAN messages at once, depending on the configured Rx-FIFO size and the size of the buffer provided by the application. Depending on the current available number of received CAN messages on return the application buffer might not be entirely filled or even empty. The parameter *len* contains the number of CAN messages copied to the application buffer.

4.2. Transmission of CAN frames:

   Use ***canWrite()/canWriteT()*** or ***canSend()/canSendT()*** to transmit CAN frames on the CAN bus.

   Either call is able to send multiple messages. But only ***canWrite()*** or ***canWriteT()*** return status information in case of communication errors.

Chapter 8 conatins several self-contained examples which demonstrate transmitting and receiving CAN CC messages.

3.11 Rx Object Mode

In addition to the FIFO mode described in the previous chapter most CAN driver also support an object mode for polled I/O which is described in this chapter.

3.11.1 Overview

If an application is only interested in the most recent data of a CAN message the handle can be initialized using the object mode instead of the default FIFO mode. The operation mode for receiving CAN messages does not influence the transmission of CAN messages with this handle and the possibilities of configuring a receive filter.

CAN driver versions before 3.x are limited to CAN messages in the *Base Frame Format* (11-bit CAN-IDs) later versions also support CAN messages in the *Extended Frame Format* (29-bit CAN-IDs).

If the object mode is configured for a handle, only the non-blocking API calls to receive CAN messages ***canTake()/canTakeT()/canTakeX()*** are supported. Using a blocking call ***canRead()/canReadT()*** with a handle configured in object mode will return an error.

In contrast to calling ***canTake()/canTakeT()/canTakeX()*** in FIFO mode, the CAN identifiers have to be initialized in the application buffer before the call, because the device driver uses this information to determine the CAN messages which are of interest to the application. The amount and order of messages can be adapted by the application with every call to ***canTake()/canTakeT()/canTakeX()***, but it has to correspond to the configuration of the message filter.

As not all driver on all platforms support the object mode the application should check for the feature flag `NTCAN_FEATURE_RX_OBJECT_MODE` returned with ***canStatus()***.

To distinguish between a driver which just supports CAN messages with 11-bit CAN-IDs from a CAN driver which also supports CAN messages with 29-bit CAN-IDs the application should check for the feature flag `NTCAN_FEATURE_SMART_ID_FILTER` returned with ***canStatus()***.

To check if the object was updated between consecutive non-blocking receive operations an application can compare the time-stamp of the CAN messages returened with ***canTakeT()*** or ***canTakeX()*** or, starting with driver version V4.1.x, the *msg_lost* counter of the returned CAN messages (see chapter 6.2.3).

3.11.2 Reception of CAN Frames

1. Open a CAN-handle with ***canOpen()*** and mode flag `NTCAN_MODE_OBJECT`.

   The parameter *rxtimeout* will be ignored. The *rxqueuesize* should be set to the maximum number of different CAN messages (messages with different CAN-IDs) the application wants to receive.

2. Set the baud rate as described in chapter 3.3.1 for the physical CAN port.

The bit rate might be already configured by another application. Thus you are advised to check the current configuration in advance as described in chapter 3.3.1.

3a. Configure the message filter.

   All CAN-IDs, the application is interested in, have to be enabled in the acceptance filter with e.g. ***canIdAdd()*** as described in chapter 3.8.

3b. Create the object internally.

   The application must create the objects for the CAN messages of interest with an initial call of ***canTake()/canTakeT()/canTakeX()*** for the respective CAN-IDs.

Because of the different internal driver architecture the step 3a) is only required for V2.x device driver and the step 3b) only for V3.x/V4.x device driver. If the application does not want to implement a different behavior based on the driver version it can perform both steps independent of the driver version.

4. Reception of CAN frames.

   Use ***canTake()/canTakeT()/canTakeX()*** (obviously blocking receive calls like ***canRead()*** makes no sense in this mode) to retrieve the most recent data of a certain CAN message. To indicate the IDs of the message the application is interested in, the identifier (*id)* of the respective `CMSG/CMSG_T/CMSG_X` structure(s) have to be initialized before calling ***canTake()/canTakeT()/canTakeX()***.

Please refer to chapter 8.3 for a self-contained example using the Rx Object Mode to receive CAN CC messages.

3.12 Tx Object Mode

The *Tx Object Mode* implements an application configurable dynamic set of objects which can be transmitted autonomously by the driver. This feature is used for 2 purposes:

- *Scheduling Mode* for CAN messages
- *Autoanswer Mode* for CAN Remote Request (RTR) messages.

As the driver or the (active) CAN board is transmitting the messages autonomously without the need for application support they are transmitted with a very low jitter with respect to the cycle time (*Scheduling Mode*) or a very fast response time (*Autoanswer Mode*).

The Tx Object Mode requires CAN driver V3.x or later.

3.12.1 Scheduling Mode

Scheduling enables an application to schedule the transmission of CAN frames at a certain point of time in the future and it is possible to define 'jobs' which do this cyclically. For this reason the scheduling mode is ideally suited for (cyclic) CAN message transmission with a very low jitter.

There are no dedicated functions in the NTCAN-API which create, configure and destroy the Tx-objects used for scheduling, instead configuration and update is performed with ***canIoctl()*** and the `NTCAN_IOCTL_TX_OBJ_XXX` group of commands. A TX Object might either represent a `CMSG` or a `CMSG_X`.

Different individual scheduling sets can be defined using different CAN handles and a scheduling set is always related to the CAN handle. Scheduling sets can only be defined or modified while scheduling is stopped for this CAN port. If the handle related to a scheduling set is closed while scheduling is active the scheduling is implicitly stopped and all related resources are released. If a message can not be transmitted within it's interval it is silently discarded.

!! Only **ONE** scheduling set per CAN port is allowed to be active !!

The following steps are necessary to setup a scheduling set for CAN CC communication. For CAN FD similar steps with the CAN FD enabled API calls and objects have to be performed.

1. Create a NTCAN_HANDLE with ***canOpen().***

2. Set the bit rate with ***canSetBaudrate()*** for the physical CAN port.

This might be already done by another thread or process. Thus you are advised to check the baud rate of the CAN bus in advance with ***canGetBaudrate()***.

3. Create TX Objects with NTCAN_IOCTL_TX_OBJ_CREATE / NTCAN_IOCTL_TX_OBJ_CREATE_X commands with ***canIoctl().*** The TX Objects are defined based on their CAN-ID and the physical CAN port referenced by the CAN handle.

Even for different scheduling sets the driver implementation guarantees that there can always be one TX Object per CAN-ID and CAN port.

4. Initialize the TX Objects with NTCAN_IOCTL_TX_OBJ_SCHEDULE commands via ***canIoctl().*** Every TX Object can have an individual I/O configuration with respect to transmission time, transmission type, etc. (refer to description of CSCHED for all configuration options).

5. Start the scheduling for this set with the NTCAN_IOCTL_TX_OBJ_SCHEDULE_START command for ***canIoctl().*** If the start time for this scheduling set is not absolute it is considered to be relative to the point of time of this API call. Once the scheduling for this set is active a change of the configuration described in the previous two steps is no longer possible until the scheduling for this set is stopped with the NTCAN_IOCTL_TX_OBJ_SCHEDULE_STOP command for ***canIoctl()*** in order to guarantee a deterministic transmission.

6. Updating the Tx Objects's data of a scheduling set is possible at any time using the commands NTCAN_IOCTL_TX_OBJ_UPDATE / NTCAN_IOCTL_TX_OBJ_UPDATE_X for ***canIoctl().*** An update in combination with an out-of-order transmission is triggered by using the respective standard blocking or non blocking transmit API calls (see chapter 4.4).

7. Disabling or (re-)enabling a scheduled Tx Object is possible at any time with NTCAN_IOCTL_TX_OBJ_SCHEDULE commands and the respective flags via ***canIoctl().*** Disabling means that the Tx Object remains schedulded in the temporal grid defined by the scheduling set but the transmission is suspended for the time being.

8. Removing a Tx Objects from a scheduling set is possible using the commands NTCAN_IOCTL_TX_OBJ_DESTROY / NTCAN_IOCTL_TX_OBJ_DESTRY_X for ***canIoctl().*** The latter is only possible if scheduling is stopped.

Chapter 8.8 shows a complete example how to setup a scheduled Tx object.

3.12.2 Autoanswer Mode

The *Autoanswer Mode* enables an application to define TX Objects which are sent automatically (by the device driver) on reply to a CAN Remote Request message on the related CAN-ID. This mechanism is much faster than answering requests in the application and the data of the object can be updated asynchronously without a relation to the request frequency.

3.12.2.1 Use case

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 the CAN bus on request. Using the *Autoanswer Mode* this is quite simple. All you have to do, is to generate an *Autoanswer* TX Object and afterward update the data of this object every time you're A/D-converter has generated a new value.

3.12.2.2 Configuration

The following steps are necessary to setup an autoanswer CAN CC TX Object. The CAN FD standard does not support the RTR mechanism.

1. Create a `NTCAN_HANDLE` with ***canOpen().***
2. Set the bit rate with ***canSetBaudrate()*** for the physical CAN port.

This might be already done by another thread or process. Thus you are advised to check the baud rate of the CAN bus in advance with ***canGetBaudrate()***.

3. Configure the message filter using ***canIdAdd()***. You have to enable at least the one CAN-ID you want to be answered automatically on RTR.
4. Create an *Autoanswer* TX Object calling ***canIoctl()*** 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 *Autoanswer* object as often and anytime you like.

It is possible to use the *Autoanswer Mode* in combination with e.g. normal FIFO-modes so the data is also updated implicitly by using a respective standard blocking or non blocking transmit API call (see chapter 4.4).

3.13 Error Injection

The *Error Injection Module* is an extension to an FPGA based CAN controller board. Via several trigger modes it is possible to generate all kinds of errors on CAN networks. This implies the possibility to test existing systems with reproducible CAN errors. Depending on the configuration, a CAN controller can have several units, so it is possible to create complex error scenarios by combining these units. The concepts and some use cases of error injection are described in /6/.

The *Error Injection Module* is not available on every CAN controller board, the application should check for the feature flag `NTCAN_FEATURE_ERROR_INJECTION` returned with ***canStatus()***.

3.13.1 Overview

The *Error Injection* is an additional module on FPGA based CAN boards. The *Error Injection Module* is divided into several *Error Injection Units*. These units can be assigned to the different esdACC Controllers. Figure 13 shows a stripped-down overview of the FPGA structure. The *Error Injection* is configured in the same way as the esdACC CAN controllers. CAN signals and states are available by the esdACC Controller. The outputs of both are combined to one output signal. So there are no changes in the wire connection necessary.

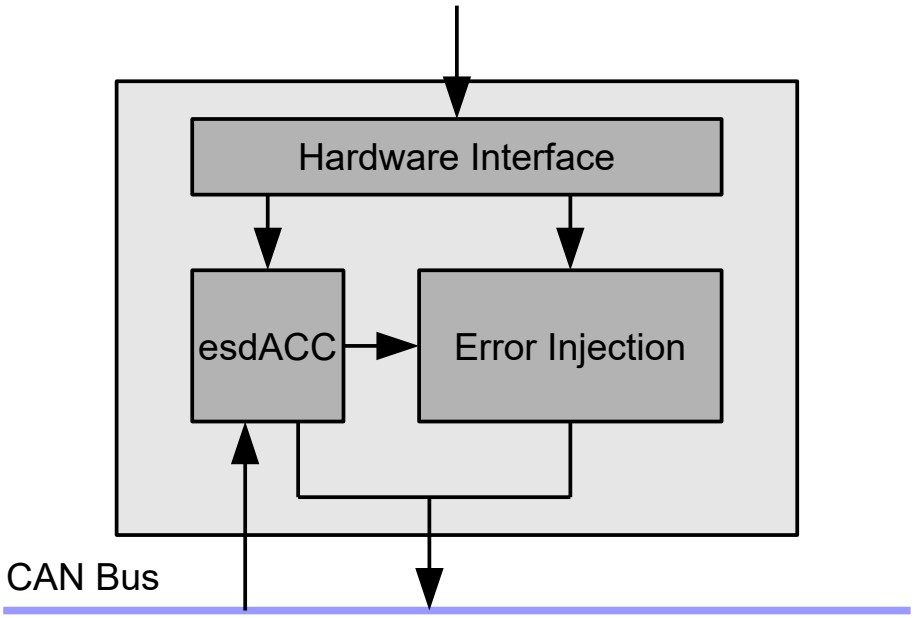


Figure 13: Overview of esdACC Error Injection

As already mentioned, the *Error Injection Module* consists of several units. An *Error Injection Unit* (shown in figure 14) is composed of a sending module (CAN TX) and several Trigger Units (Trigger ...). With the CAN TX module is it possible to send a user defined bit stream without CRC calculation or bit stuffing. There is no CAN bus feedback, so the transmission will not be terminated, if CAN error Frames are sent. This sending module can be triggered by five Trigger Units.

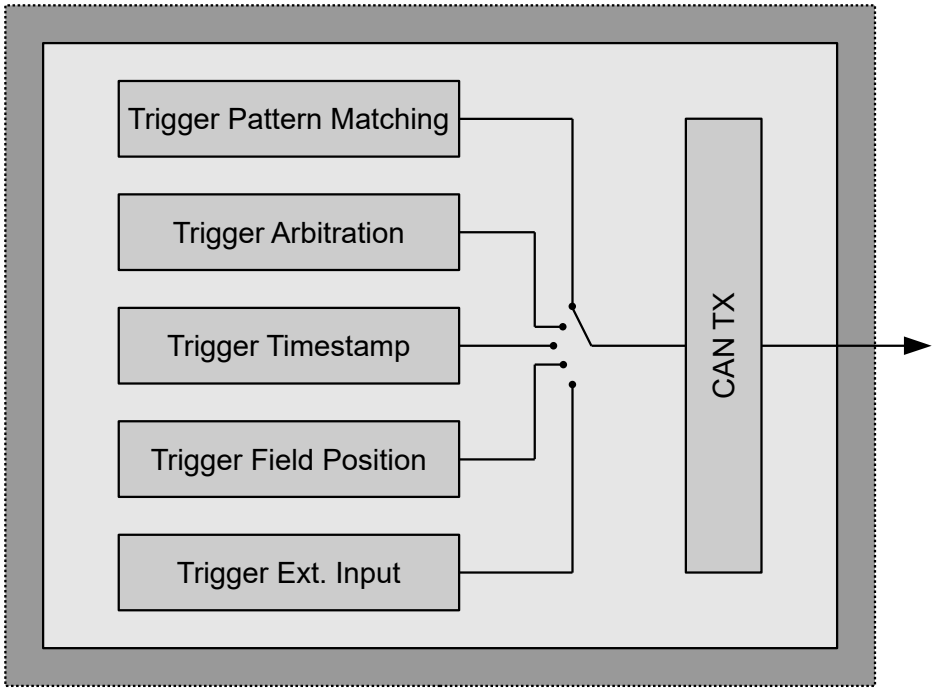


Figure 14: Error Injection Unit

Trigger Pattern Matching

This module can be searched for a user defined bit stream. If the bit stream matched on the sampled CAN bit stream the CAN TX is triggered.

Trigger Arbitration

This module sends via the CAN TX module a bit stream under the rules of arbitration. It is possible to send correct CAN frames or CAN frames with errors.

Trigger Timestamp

Trigger the CAN TX Module via a timestamp .

Trigger Field Position

It can be defined a position in a CAN frame in the coding of the ECC register (SJA1000)

Trigger external Input

Each Error Injection Unit has a trigger out signal, this Signal can be an external Trigger source for this Trigger module. Via a bitmask can be chosen, which Trigger Unit is the trigger source. Additionally it is possible to trigger via a REAR IO pin.

3.13.2 Usage

In this chapter the typical steps to use the *Error Injection* are summarized:

1. Create a `NTCAN_HANDLE` with ***canOpen().***
2. Set the bit rate with ***canSetBaudrate()*** for the physical CAN port.

This might be already done by another thread or process. Thus you are advised to check the baud rate of the CAN bus in advance with ***canGetBaudrate()***.

3. Create an *Error Injection Unit* with `NTCAN_IOCTL_EEI_CREATE` command with ***canIoctl().*** The *Error Injection Units* are defined based on their CAN-ID and the physical CAN port referenced by the CAN handle. The call will return a handle to the newly created *Error Injection Unit*.
4. Configure the *Error Injection Unit* with `NTCAN_IOCTL_EEI_CONFIGURE` command with ***canIoctl().*** The argument is a `NTCAN_EEI_UNIT` structure which is passed through a pointer.
5. With `NTCAN_IOCTL_EEI_STATUS` command as argument for ***canIoctl()*** the current status of the *Error Injection Unit* will be queried. The argument is a `NTCAN_EEI_STATUS` structure which is passed as a pointer.
6. If the *Error Injection Unit* completed its tasks, it can be reconfigured or released with the `NTCAN_IOCTL_EEI_DESTROY` command.

3.14 Timestamped TX

3.14.1 Overview

At a first glance *Timestamped TX* enables you to transmit CAN frames at a certain time using ***canSendT()*** and ***canWriteT()*** functions. But this feature goes deeper:

- Internally the driver will use this mechanism for the scheduling in Scheduling Mode
- On CAN hardware supporting this feature, as for example esd electronics's CAN/400 and CAN/402 family, *Timestamped TX* elevates the precision of your TX jobs up to plus/minus one Bit-time (assuming conditions on CAN bus allow such transmissions)
- Additional high priority TX FIFO (only mutually exclusive to timestamped transmission)

The *Timestamped TX mode* is available on certain CAN controller boards only, the application needs to check for feature flag `NTCAN_FEATURE_TIMESTAMPED_TX` returned with ***canStatus()***

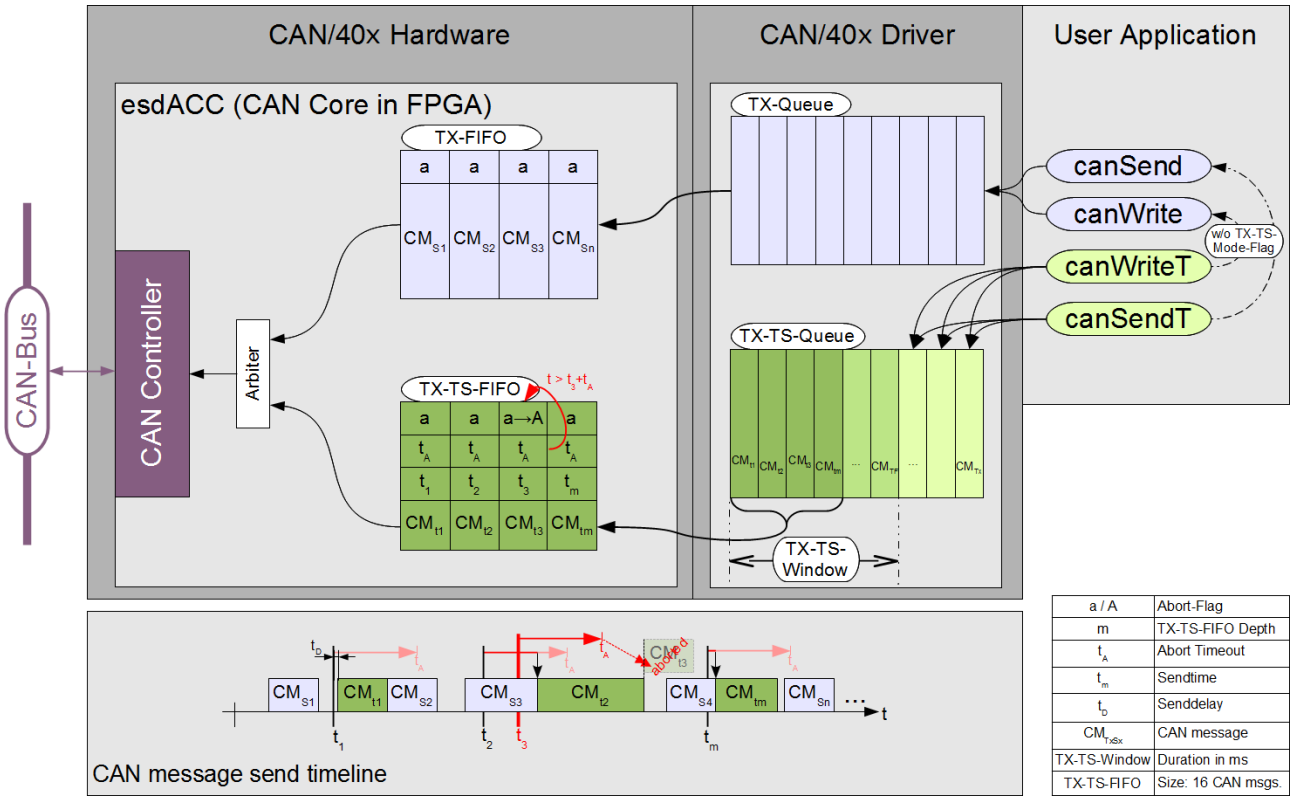


Figure 15: Timestamped TX with canSendT() and canWriteT() on CAN/40x family

3.14.2 General rules and behaviour

- *Timestamped TX* won't break any existing code nor any of your applications. If you use ***canOpen()*** without `NTCAN_MODE_TIMESTAMPED_TX` flag, ***canSendT()*** and ***canWriteT()*** will work exactly the same way as they used prior to *Timestamped TX* and exactly as ***canSend()*** and ***canWrite()*** do, except that they will accept `CMSG_T` structures
- Time of transmission is specified as an absolute time in timestamp format
- Frames with timestamp zero will be transmitted through the normal TX-queue and TX-FIFO, regardless of the function used or flags set on ***canOpen()***
- CAN frames scheduled for transmission in *Timestamped TX* mode will be enqueued in chronological order into the TX-TS-Queue
- Frames scheduled on the same point in time within one TX job (one ***canSendT()*** or ***canWriteT()*** call) will keep their given order
- Frames scheduled on the same point in time from different TX jobs (multiple different ***canSendT()*** or ***canWriteT()*** calls) will be transmitted in the order of the posting of these jobs
- TX-TS-Window (see Figure 15):
  - A user configurable time (default is hardware dependent, usually a “few” milliseconds) before the actual planned time of transmission the driver will move frames into the so called TX-TS-Window
  - CAN Frames within the TX-TS-Window will not be reordered
  - New frames can be appended to the TX-TS-Window, only
  - New frames will never be interleaved with frames already residing in the TX-TS-Window
  - From the TX-TS-Window the driver will provide the CAN hardware with CAN frames for transmission
  - This mechanism allows to accommodate different data busses (e.g. PCIe, USB,...), operating system latencies and hardware capabilities
  - CAN hardware providing hardware FIFOs or multiple transmission objects will be fed with multiple CAN frames from the driver
- An arbiter will prioritize CAN frames from the TX-TS path which are ready to send over frames from the normal TX path ($t >= t_x$ with x=[1..m] in Figure 15)
- CAN hardware supporting this mechanism in hardware (like the CAN/400-family) reaches an accuracy of plus/minus one Bit-time, when sending onto a free CAN bus (t_D in Figure 15)
- The arbiter won't take a frame prior to the programmed time of transmission, even if the time would be reached, while a frame from the normal TX-FIFO is still in the process of transmission (see CM_{t2} in Figure 15)
- Frame timeouts (see below) are available on certain hardware only (e.g. CAN/400 and CAN/402 family)
- All known abort mechanisms (TX timeout, `NTCAN_IOCTL_ABORT_TX`, …) work with *Timestamped TX* as well

3.14.3 Timestamped TX via canSendT() and canWriteT()

1. Open a NTCAN handle via ***canOpen()*** with `NTCAN_MODE_TIMESTAMPED_TX` flag set.
2. Write the desired time of transmission into the timestamp field of your `CMSG_T` structures before feeding them into ***canSendT()*** or ***canWriteT()***.

Using ***canWriteT()*** in this mode will behave exactly as one should expect. It will return only after the last frame of the job has been transmitted (depending on your chosen points of transmission this might be well in the future...).

canWriteT() will still return the number of successfully transmitted frames, but as the order of frames depends on the given timestamps, the returned value can't be mapped directly to your given CMSG_T array, if the messages were not in chronological order.

Timestamped TX is fully compliant with /2/ and does not break any CAN-rules of transmission. A scheduled frame will only be transmitted on time, if the bus is idle and the CAN priority (CAN-ID) qualifies for transmission at that certain moment. Otherwise it's transmission is delayed until correct conditions are given or the transmission is aborted (e.g. by timeout).

3.14.4 High priority TX FIFO

If you use *Timestamped TX* in the same way as described before, but instead of using timestamps in the future you use timestamps in the past (everything except zero is perfectly fine, one might be a good timestamp), the frames will be transmitted as soon as possible with the advantage of having a higher priority than the frames send with ***canSend()*** or ***canWrite()***.

3.14.5 TX Object mode scheduling

No additional efforts need to be taken. The TX object mode will automatically profit from the increased scheduling precision, if your CAN hardware supports *Timestamped TX*.

3.14.6 Frame timeout

On certain hardware it is possible to configure a “per frame timeout” via ***canIoctl()***. The following rules apply:

- The frame is aborted, if the timeout (t_A in Figure 15) is reached at the moment the frame would be taken from the TX-TS-FIFO
- The frame will not be aborted, when the timeout will only be reached, while it is already in the process of transmission
- Frame timeouts are specified in timestamp units (see `NTCAN_IOCTL_GET_TIMESTAMP_FREQ` in ***canIoctl()***)

3.15 Transmitter Delay Compensation (TDC)

3.15.1 Overview

The basic idea to reach higher data rates with CAN FD compared to the CAN CC is the introduction of a high speed data phase (after an arbitration phase with the nominal bit rate) with only one CAN node transmitting data while all other nodes receiving. During this time the propagation delay of CAN CC communication does not limit the maximum data rate for the receiving nodes as they do not drive any bits in this phase.

For the transmitting node, however, the basic operating principle of CAN must be adhered to, that a transmitter receives its own data and indicates an error if a difference is detected at the sampling point of a bit. This contradicts the data phase performance improvement of CAN FD described above, as the CAN bit time in this phase might even become smaller than the *Transmitter (Loop) Delay (TD)*, which makes a comparison within the current bit time interval impossible. The TD consists of the following individual (possibly asymmetrical Rx and Tx) delays:

- The CAN FD controller internal delay
- The CAN transceiver delay
- The galvanic isolation delay (see /8/ for further details).

To overcome this problem for the transmitting node in the data phase /2/ introduces a ***Transmitter Delay Compensation*** (TDC) mechanism which defines a ***Secondary Sample Point*** (SSP) which is delayed in a way that the transmitted bit can be correctly compared with the received one. The position of the SSP is determined, as shown in the picture below, as an offset from the start of the bit time in a multiple of *minimum time quanta* (mtq) which are usually CAN clock periods:

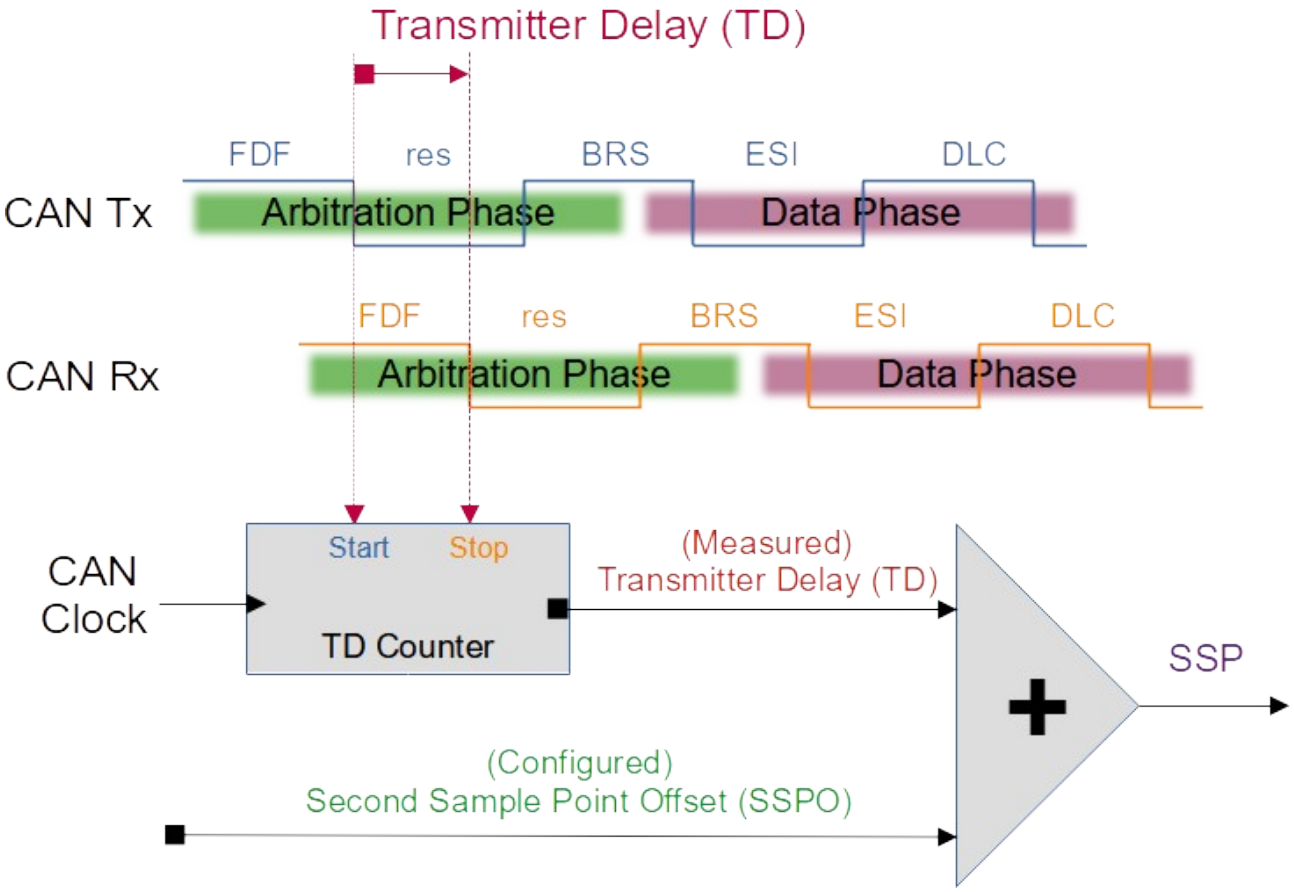


Figure 16: Transmitter Delay Compensation (TDC) and Second Sample Point (SSP)

In the data phase the transmitting node keeps the value of the current transmitted bit internally until its SSP is reached to compare it with the actually received bit value. For the SSP position within the transmitted bit, dynamic effects like transceiver asymmetries, temperature drift as well as ringing on the bus have to be considered. According to /8/ the SSP position can be set to an arbitrary position within the received bit after the CAN bus signal is stable.

The TD can either be calculated based on the individual hardware components data sheets (which just covers static effects for a delay) or it is measured for each CAN FD frame before the start of the data phase to compensate dynamic effects, too, as shown in Figure 16. The measurement mechanism resets the TD Counter in the arbitration phase when the node starts to drive the (dominant) *res* bit after the (recessive) *FDF* bit on the CAN FD controller TX pin. The TD counter is stopped as soon as the dominant level is received at the CAN FD controller RX pin. The resulting counter value is the measured TD.

The measurement mechanism described above moves the SSP to the start of the time window the CAN bus signal of the received bit becomes stable. In order to move it to a more optimal position within this time window a S*econd Sample Point Offset* (SSPO) can be added, so the effective *Second Sample Point* (SSP) is set to :

SSP := TD + SSPO

According to /1/ the TDC mechanism can be enabled or disabled. It must be enabled for all data phase bit rates with bit times smaller than the transmitter loop delay (which is usually above 1 Mbit/s). With active TDC the data phase bit rate pre-scaler must be configured to one or two (see /2/).

Some CAN FD controllers support a manual mode which ignore the measured TD and just use a configurable offset to define the SSP position.

Some CAN FD controllers support a filter to prevent that a dominant glitch inside the FDF bit causes a premature end of the TD delay measurement which lead to a wrong SSP position and in consequence may lead to the indication of a bit error within the data phase. The filter is implemented by considering a Transmitter Delay Compensation Filter (TDCF) value as a minimum counter value which has to be exceeded before the TD measurement is stopped.

3.15.2 SSP Configuration

If a CAN FD bit rate is configured with the NTCAN-API (see chapter 3.3.1), the TDC mechanism is enabled in the *TDC Automatic Mode* and the SSP used in the transmitting mode is automatically configured to the SPO which is the configured data phase SP in the receiving mode. Usually it is not required to adapt this SSP configuration in this default mode.

The table below contains a description of the timing parameters which are relevant for the NTCAN TDC configuration mechanism. The basic unit for all parameters is the Minimum Time Quanta (mtq) and not the Time Quanta (tq) which is used for the CAN bitrate configuration (see appendix Bus Timing).

Parameter	Description
SPO	The *Sample Point Offset* is the configured data phase Sample Point for a receiving node in mtq.
SSP	The Second Sample Point (SSP) is the number of mtq between the start of the transmitted bit and the secondary sample point. It is the sum of TD and SSPO.
SSPO	The Second Sample Point Offset is added to the measured TD to place the Secondary Sample Point (SSP) at an appropriate position.
SSPS	The Second Sample Point Shift is an application defined shift value in mtq to move the NTCAN defined default SSP (derived from TD and SPO in automatic mode).
TD	The Transmitter Delay is the measured physical delay in mtq of a transmitting node receiving its own bit stream.

3.15.2.1 TDC Automatic Mode

The configuration of the SSP in the NTCAN TDC Automatic Mode is shown in the picture below as a data phase with a configured data rate that leads to a CAN bit time smaller than the transmitter delay.

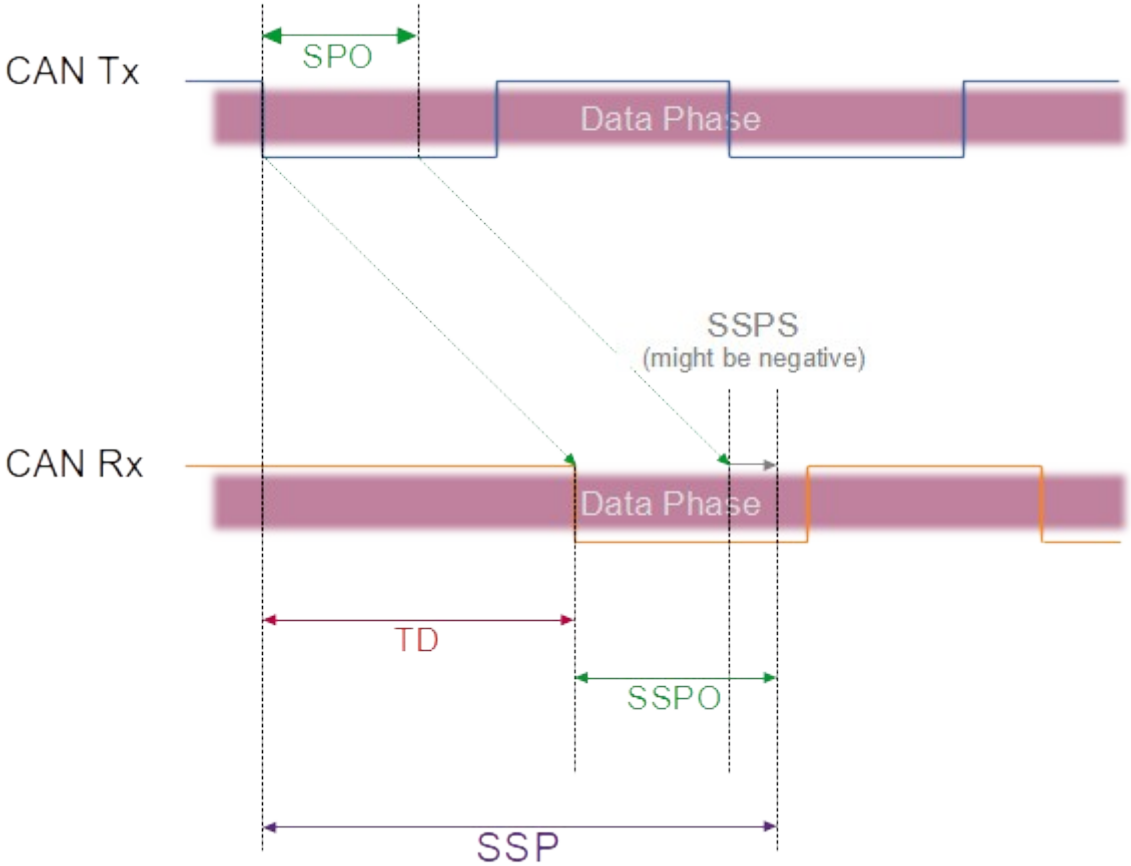


Figure 17: SSP configuration in NTCAN TDC Automatic Mode.

The CAN FD Controller measures constantly the dynamic Transmitter Delay (TD) in the arbitration phase as described in chapter 3.15.2.1. The NTCAN driver or firmware will automatically configure a Second Sample Point Offset (SSPO) identical to the Sample Point Offset (SPO).

SSP = TD + SSPO = TD + (SPO + SSPS)

The application can adapt this default SSP via the configurable Second Sample Point Shift (SSPS) value (default: 0). As the latter is defined as a signed value in TDC Automatic Mode the SSP can be shifted in a positive or negative direction.

3.15.2.2 TDC Manual Mode

The configuration of the SSP in the NTCAN TDC Manual Mode is shown in the picture below as a data phase with a configured data rate that leads to a CAN bit time smaller than the transmitter delay.

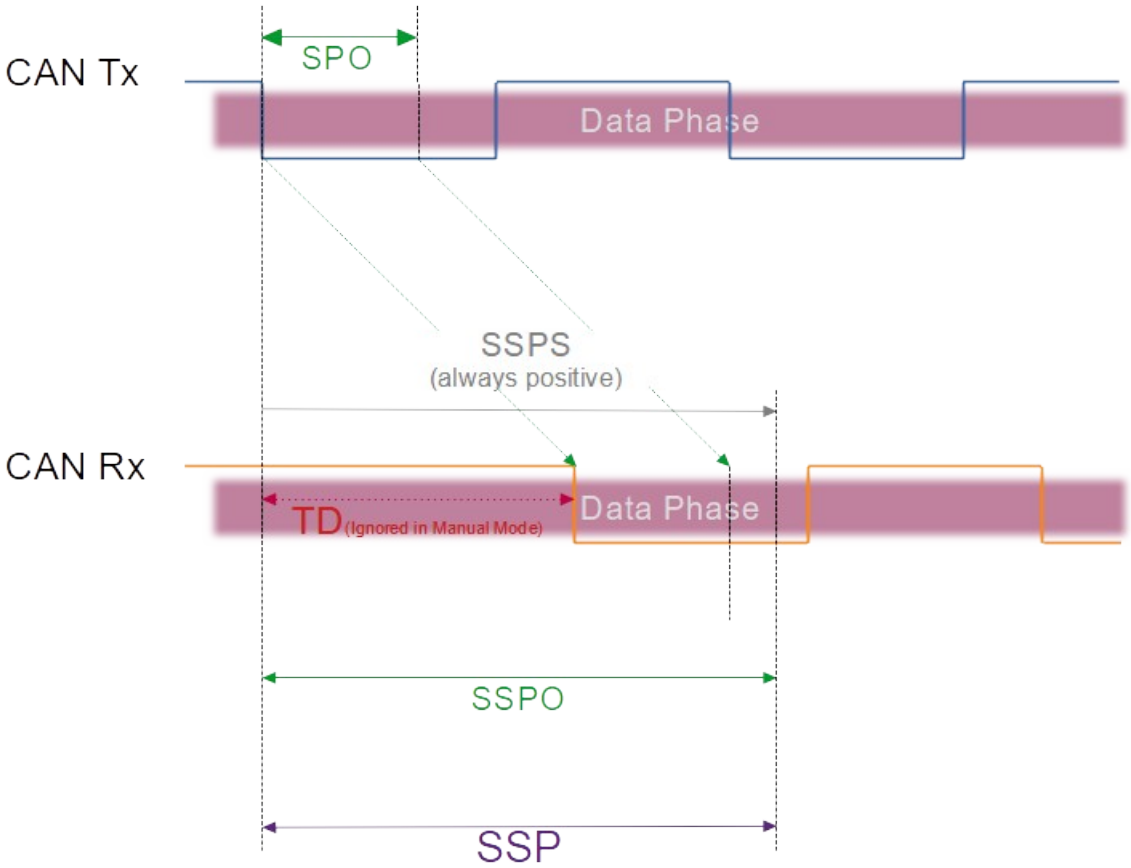


Figure 18: SSP configuration in NTCAN TDC Manual Mode.

The CAN FD Controller will ignore the internally measured the Transmitter Delay (TD) in the arbitration phase and the NTCAN driver or firmware will not automatically configure a Second Sample Point Offset (SSPO) identical to the Sample Point Offset (SPO). The effective default SSP without any further configuration is 0 mtq.

SSP = SSPS

The application must define the effective SSP via the Second Sample Point Shift (SSPS). The application must consider the TD given in the data sheets and an SSPO. In the TDC Manual Mode the SSPS is defined as an unsigned value as a shift in negative direction makes no sense.

3.15.2.3 TDC Mode Parameters

The NTCAN implements an intelligent automatic mode which defines an optimal SSP based on the continuously measured TD and the data phase bit rate configuration parameters. This default SSP can be optionally adapted or the automatic can be disabled at all with ***canSetBaudrateX()*** which is usually not required.

The current (configuration) parameters of the TDC mechanism can be requested via ***canIoctl()*** with `NTCAN_IOCTL_GET_FD_TDC`.

Bit	Read	Name	Description
31..30	RW	TDCM	The **Transmitter Delay Compensation Mode** (TDCM) can have one of the following values: ➢ **`NTCAN_TDC_MODE_AUTO`**: See chapter 3.15.2.1. ➢ **`NTCAN_TDC_MODE_MANUAL`**: See chapter 3.15.2.2. ➢ **`NTCAN_TDC_MODE_OFF`**: TDC disabled. In this mode the bit time must not be smaller than the TD. Default: **`NTCAN_TDC_MODE_AUTO`**
29..23	RO	SSPO	Second Sample Point Offset in mtq 0: Invalid 1..63/127 (for TDCEXT = 0/1)
22..16	RW	SSPS	Second Sample Point Shift in mtq **`NTCAN_TDC_MODE_AUTO:`** -32..31 (TDCEXT = 0) / -64..63 for (TDCEXT = 1) **`NTCAN_TDC_MODE_MANUAL:`** 0..63 (TDCEXT = 0) / 0..127 for (TDCEXT = 1) Note: In NTCAN_SET_TDC the values for the two different TDC modes are mapped to separate variables with appropriate data type signedness.
15	RO	TDCI	TDC Mechanism Inactive (0 = Active / 1 = Inactive)
14..8	RO	TDCF	Transmitter Delay Compensation Filter in mtq 0..63 (for TDCEXT = 0) / 0..127 for (for TDCEXT = 1)
7	RO	TDCEXT	0 = Register sizes are 6 bit (According to /1/) 1 = Register sizes are 7 bit (According to /8/)
6..0	RO	TD	Transmitter Delay in mtq 0..63 (for TDCEXT = 0) / 0..127 for (for TDCEXT = 1)

Table 8: Structure of the NTCAN_IOCTL_GET_FD_TDC argument.

The macros NTCAN_GET_TDC_FILTER, NTCAN_GET_TDC_SSPS, NTCAN_GET_TDC_TD ease evaluating a returned value.

Note: The legacy command NTCAN_IOCTL_SET_FD_TDC to configure TDCM and SSPS is still supported for backward compatibility. An application which adapts the TDC mechanism that way must set all fields with RO access in the table above to 0.

3.16 Switchable Bus Termination

3.16.1 Overview

According to /2/ the physical twisted-pair wires of a CAN network requires a specific impedance, typically provided by a 120 Ohm resistors on each end of the bus. Some *esd electronics* CAN interfaces allow to activate/deactivate a bus termination resistor for each physical CAN port with the NTCAN API.

The *Switchable Bus Termination* support is only availabe for a subset of the *esd electronics* CAN boards. An application should check for the feature flag `NTCAN_FEATURE_PROG_TERM` returned with ***canStatus()***.

To prevent an erroneous bus termination please refer to the hardware manual of your CAN board for the following implementation details of the *Switchable Bus Termination*:

- Can the software configuration be overridden by a manual configuration ?
- Is the default state an activated or deactivated termination resistor ?
- Remains a software activated resistor active while the CAN hardware is unpowered ?

3.16.2 Usage

Do the following steps to activate/deactivate the CAN bus termination:

1. Create a `NTCAN_HANDLE` with ***canOpen()*** for the physical CAN port.

2. Activate or deactivate the CAN bust termination resistor using the arguments `NTCAN_TERM_ENABLE` respectively `NTCAN_TERM_DISABLE` for the he command `NTCAN_IOCTL_SET_TERM_CFG` passed via ***canIoctl().***

3. Set the bit rate with ***canSetBaudrate()*** or ***canSetBaudrateX()*** with this handle and start transmitting or receiving messages

3.17 GPIO Support

3.17.1 Overview

Some esd electronics CAN interfaces allow to control additional General Purpose Input/Output (GPIO) ports with the NTCAN API.

➢The GPIO hardware support is only availabe for a subset of the *esd electronics* CAN boards. An application should check for the feature flag `NTCAN_FEATURE_GPIO` returned with ***canIoctl(***`NTCAN_IOCTL_GET_INFO`**)** in `NTCAN_INFO::features`.

The NTCAN API supports the control of up to 32 I/O channels. The configuration of the I/O channels is set or can be requested on a per channel basis via ***canIoctl()*** based on the `NTCAN_GPIO_CFG` structure.

The state of the GPIO channels is updated or requested based on the NTCAN Events mechanism. This allows a polled as well as event based handling of the GPIOs. There are dedicated *NTCAN Events* to:

- Set the state of the channels configured as digital outputs (`NTCAN_EV_GPIO_SET_DO`)
- Get the state of the channels configured as digital outputs (`NTCAN_EV_GPIO_GET_DO`)
- Get the state of the channels configured as digital inputs (`NTCAN_EV_GPIO_GET_DI`)
- Change the direction of an I/O channel (`NTCAN_EV_GPIO_SET_DIR`)

The common state of all up to 32 I/Os is set or returned in the EV_GPIO_DATA structure which allows to define which channels are included/excluded by the command with a bitmask.

GPIO related I/O operations are only supported on a NTCAN handle openend for the logical *base net* of the CAN hardware and will fail on all other logical nets.

3.17.2 Polling mode

Do the following steps to control the GPIO ports in a polling mode:

1. Create a `NTCAN_HANDLE` with ***canOpen()*** in FIFO mode with the logical net number of the base net of the CAN board.

2. Configure each I/O channel with ***canIoctl()*** and the `NTCAN_IOCTL_SET_GPIO_CFG` command. Set the member *irq_mode* in the `NTCAN_GPIO_CFG` structure to `NTCAN_GPIO_CFG_IRQ_NONE` to enforce the polling operation mode.

3. Enable the NTCAN Event `NTCAN_EV_GPIO_GET_DI` in the handle acceptance filter with ***canIdAdd()*** together with other NTCAN events and CAN messages you want to receive with this handle.

4. Set the state of the digital outputs with a transmit operation like ***canSend()*** sending the `NTCAN_EV_GPIO_SET_DO` event with a configured `EV_GPIO_DATA` structure.

5. Get the state of the digital inputs with a transmit operation like ***canSend()*** sending the `NTCAN_EV_GPIO_GET_DI` event which triggers the device driver to store the current state as `NTCAN_EV_GPIO_GET_DI` event in the handle Rx FIFO which can be polled with ***canTake().***

3.17.3 Event based mode

Do the following steps to control the GPIO ports in an event based mode:

1. Create a `NTCAN_HANDLE` with ***canOpen()*** in FIFO mode with the logical net number of the base net of the CAN board.

2. Configure each I/O channel with ***canIoctl()*** and the `NTCAN_IOCTL_SET_GPIO_CFG` command. Set the member *irq_mode* in the `NTCAN_GPIO_CFG` structure to a value different from `NTCAN_GPIO_CFG_IRQ_NONE` to enforce an event based operation mode.

3. Enable the NTCAN Event `NTCAN_EV_GPIO_GET_DI` in the handle acceptance filter with ***canIdAdd()*** together with other NTCAN events and CAN messages you want to receive with this handle.

4. Set the state of the digital outputs with a transmit operation like ***canSend()*** sending the `NTCAN_EV_GPIO_SET_DO` event with a configured `EV_GPIO_DATA` structure.

5. The state of the digital inputs is stored by the device driver on change as `NTCAN_EV_GPIO_GET_DI` event in the handle Rx FIFO which can be received with a receive operation as ***canTake().*** Polling the `NTCAN_EV_GPIO_GET_DI` event concurrently, as described in the previous section, is still possible.

3.18 Operating System Support

This chapter gives an overview on which **esd electronics** CAN board is supported by which operating system (OS).

In the headings of the tables 10 to 12 in this chapter actively supported OS (versions) are written with **bold** faces and legacy OS (versions) are written with grey faces.

Attention!

All further developments for OS (or OS versions) categorised as legacy by esd electronics have been discontinued. In most cases the latest version of the device driver files for the respective OS (version) can still be made available but without any technical support from **esd electronics**.

As described in chapter 2.1 every NTCAN implementation exports the same set of API functions to do basic CAN-I/O. The individual implementation for a certain OS (version) and CAN board might provide additional capabilities (e.g. hardware timestamps, error injection, ...) which are not supported by other implementation because of hardware and/or OS limitations.

The differences between individual driver implementations with respect to their capabilities is covered with feature IDs which refer to table 9 on the next page. A simple '+' means that the platform is supported but no feature of table 9 applies and a '-' means that the hardware is not supported on this platform.

The background colour of the cells indicates if the hardware is supported with a driver version 1.x, 2.x, 3.x or 4.x (see chapter 2.2) according to the following schema.

	Driver V 4.x		Driver V 3.x		Driver V 2.x		Driver V 1.x

Feature ID	Description
1	CAN hardware is only supported by a legacy Windows NT driver on Win 2000/XP.
2	The CAN driver is distributed as source and has to be build for the customer's system
3	No driver support for EFF (29-bit CAN-IDs). Hardware with the CAN controller Philips 82C200 (manufactured until December 1999) does not support 29-bit-IDs.
4	Support for asynchronous I/O on Windows (Overlapped I/O)
5	Support for *RxObject Mode* (see chapter 3.11)
6	Support for *Listen Only Mode* (see chapter 3.3.2)
7	Support for *Frame Scheduling* (see chapter 3.12)
8	Support for Timestamps (see chapter 3.9)
9	Support for 29-bit ID filter mask (see chapter 3.8.2.2)
10	Support for a firmware update (see /1/).
11	Support for *Smart Disconnect* from CAN bus (see chapter 3.3.8)
12	Support for *Baud Rate Change Event* (see chapter 3.7)
13	Support for *Automatic Baud Rate Detection* (see chapter 3.3.7)
14	Support for *Extended CAN Bus Diagnostic* (see chapter 3.6.2)
15	Support for *Error Injection* (see chapter 3.13)
16	Support for *Timestamped Tx (*see chapter 3.14*).*
17	The CAN hardware is supported by a Linux-CAN (aka SocketCAN) driver which is part of Linux since kernel 2.6.25. If the device is not additionally supported by a NTCAN driver a wrapper library is available which maps from NTCAN to Linux-CAN API.
18	Support for CAN FD (see chapter 1.4).
19	LIN support for enabled hardware (see chapter 1.5). **Note:** Even if the device driver supports LIN the CAN hardware must also support the LIN add-on !
20	Support for the *Disable Automatic Retransmission* (DAR) mode (see chapter 3.3.6)
21	Support for the *Transmit Pause* mode (see chapter 3.3.5)

Table 9: Driver Features

The table below shows the CAN driver capabilities for **Windows Operating Systems**.

CAN Module	Order no.	Non Real-time Windows Operating Systems					Real-time	
		Win 9x / ME	Win NT	Win XP / Vista	Win 7/8	Win 10 / 11	RTX	RTX64
EtherCAN	C.2050.xx	-	-	9, 10, 13	9, 10, 13	9, 10, 13	-	-
EtherCAN/2	C.2051.xx	-	-	6, 8, 10, 12, 13, 14	6, 8, 10, 12, 13, 14	6, 8, 10, 12, 13, 14	-	-
CAN-USB/Mini	C.2064.xx	4, 5, 6, 8, 9, 10, 11, 12	-	4, 5, 6, 8, 9, 10, 11, 12	4, 5, 6, 8, 9, 10, 11, 12	-	-	-
CAN-USB/2	C.2066.xx	-	-	4,5,6,8,9,10, 11,12,13,14	4,5,6,8,9,10, 11,12,13,14	4,5,6,8,9,10, 11,12,13,14	-	-
CAN-USB/Micro	C.2068.xx	-	-	4, 5, 6, 8, 9, 10, 11, 12, 13	4, 5, 6, 8, 9, 10, 11, 12, 13	4, 5, 6, 8, 9, 10, 11, 12, 13	-	-
CAN-USB/400	C.2069.xx	-	-	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,16	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,16	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	-	-
CAN-USB/3-FD	C.2076.62	-	-	-	-	4,5,6,8,9,10, 11,12,13,14, 18,20,21	-	-
CAN-PCC	C.2422.xx	3, 4, 5	3, 4, 5	1, 3, 4, 5	-	-	-	-
CAN-ISA/200 CAN-PC104/200	C.2011.xx C.2013.xx	4, 5, 9	3, 4, 5, 9	1, 3, 4, 5, 9	-	-	-	-
CAN-ISA/331 CAN-PC104/331	C.2010.xx C.2012.xx	4, 5, 9, 10	4, 5, 10, 16	1, 4, 5, 9, 10	-	-	-	-
CAN-PCI/200 CPCI-CAN/200	C.2021.xx C.2035.xx	4, 5, 16	4, 5, 16	4, 5, 6, 8, 9, 11, 12, 13, 14	4, 5, 6, 8, 9, 11, 12, 13, 14	4, 5, 6, 8, 9, 11, 12, 13, 14	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6, 7, 8, 9, 11, 12, 13, 14
CAN-PCIe/200 CAN-PCI104/200	C.2042.xx C.2046.xx	-	-	4, 5, 6, 8, 9, 11, 12, 13, 14	4, 5, 6, 8, 9, 11, 12, 13, 14	4, 5, 6, 8, 9, 11, 12, 13, 14	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6, 7, 8, 9, 11,12,13,14
CAN-PCI/266 PMC-CAN/266	C.2036.xx C.2040.xx	-	4, 5, 16	4, 5, 6, 8, 9, 11, 12, 13, 14	4, 5, 6, 8, 9, 11, 12, 13, 14	4, 5, 6, 8, 9, 11, 12, 13, 14	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6,7,8,9, 11,12,13,14
CAN-PCI/331 CPCI-CAN/331 PMC-CAN/331	C.2020.xx C.2027.xx C.2025.xx	4, 5, 9, 10	4, 5, 9, 10	4, 5, 8, 9, 10, 12	4, 5, 8, 9, 10, 12	4, 5, 8, 9, 10, 12	5, 6, 7, 8, 9, 11, 12	-
CAN-PCI/360 CPCI-CAN/360	C.2022.xx C.2026.xx	4, 5, 9, 10	4, 5, 9, 10	4, 5, 8, 9, 10	4, 5, 8, 9, 10	4, 5, 8, 9, 10	5, 7, 8, 9	-
CAN-PCI/400 CAN-PCIe/400 CPCI-CAN/400 PMC-CAN/400	C.2048.xx C.2043.xx C.2033.xx C.2047.xx	-	-	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15	5, 6, 7, 8, 9, 10, 11, 12, 13, 14
CAN-PCIe/402 CAN-PCI/402 CAN-PCIeMini/402 CPCI-CAN/402 CPCIserial-CAN/402	C.2045.xx C.2049.xx C.2044.0x I.2332.xx I.3001.xx	-	-	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16,	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 20	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
CAN-PCI/402-FD CAN-PCIe/402-FD CAN-PCIeMini/402-FD CAN-PCIeMiniHS/402 CAN-M.2/402-2-FD CPCIserial-CAN/402-FD PMC-CAN/402-FD XMC-CAN/402-FD	C.2049.xx C.2045.6x C.2044.6x C.2054.6x C.2074.6x I.3001.6x C.2028.xx C.2018.xx	-	-	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18,	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 19, 20, 21	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19, 20,21
CPCIserial-CAN/402-FD PMC-CAN/402-FD XMC-CAN/402-FD	C.2045.9x C.2028.xx C.2018.xx	-	-	-	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18,	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 20, 21	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 20,21
CAN-PCI/405	C.2023.xx	-	-	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	-

Table 10: CAN driver capabilities (Windows operating systems)

The table below shows the CAN driver capabilities for **UNIX Operating Systems**.

CAN Module	Order no.	Unix Operating Systems					
		Linux 2.4.x / 2.6.x / 3.x/4.x/5.x/6.x (32-/64 Bit)	**LynxOS**	**PowerMAX OS**	**Solaris**	**SGI-IRIX6.5**	**AIX**
EtherCAN	C.2050.xx	9, 10, 13	-	-	-	-	-
EtherCAN/2	C.2051.xx	6, 8, 10, 12, 13, 14	-	-	-	-	-
CAN-USB/Mini	C.2064.xx	-	-	-	-	-	-
CAN-USB/Micro CAN-USB/2	C.2068.xx C.2066.xx	17	-	-	-	-	-
CAN-USB/400	C.2069.xx	-	-	-	-	-	-
CAN-USB/3-FD	C.2076.62	-	-	-	-	-	-
CAN-ISA/200 CAN-PC104/200 (SJA1000 version)	C.2011.xx C.2013.xx	5, 6, 7, 8, 9, 11, 12, 13, 14	-	-	-	-	-
CAN-ISA/331 CAN-PC104/331	C.2010.xx C.2012.xx	5, 7, 8, 9, 10, 12,	+	-	3	-	-
CAN-PC104/200 (82527 version)	C.2013.xx	5, 7, 8, 11, 12, 16	-	-	-	-	-
CAN-PCI/200 CAN-PCIe/200 CPCI-CAN/200 CAN-PCI104/200 CAN-PCI/266 PMC-CAN/266	C.2021.xx C.2042.xx C.2035.xx C.2064.xx C.2036.xx C.2040.xx	5, 6, 7, 8, 9, 11, 12, 13, 14, 17	-	-	-	-	-
CAN-PCI/331 CPCI-CAN/331 PMC-CAN/331	C.2020.xx C.2027.xx C.2025.xx	5, 7, 8, 9, 10, 12	+	-	3	2,3	3
CAN-PCI/360 CPCI-CAN/360	C.2022.xx C.2026.xx	5, 7, 8, 9, 10, 12	-	-	-	-	-
CAN-PCI/400 CAN-PCIe/400 CPCI-CAN/400 PMC-CAN/400	C.2048.xx C.2043.xx C.2033.xx C.2047.xx	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15	-	-
CAN-PCIe/402 CAN-PCI/402 CAN-PCIeMini/402 CPCI-CAN/402 CPCIserial-CAN/402	C.2045.xx C.2049.xx C.2044.0x I.2332.xx I.3001.xx	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 20, 21	-	-	-	-	-
CAN-PCI/402-FD CAN-PCIe/402-FD CAN-PCIeMini/402-FD CAN-PCIeMiniHS/402 CAN-M.2/402-2-FD CPCIserial-CAN/402-FD PMC-CAN/402-FD XMC-CAN/402-FD	C.2049.xx C.2045.6x C.2044.6x C.2054.6x C.2074.6x I.3001.6x C.2028.xx C.2018.xx	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 19, 20, 21	-	-	-	-	-
CAN-PCI/405	C.2023.xx	5, 7, 8, 9, 10, 12, 13, 14	-	-	-	5, 8, 10	-
VME-CAN2	V.1405.xx	+	+	+	-	-	-
VME-CAN4	V.1408.xx	+	+	+	+	-	-

Table 11: CAN driver capabilities (UNIX operating systems)

The table below shows the CAN driver capabilities for **Real-Time Operating Systems**.

CAN Module	Order no.	Real-Time Operating Systems							
		INtime	OnTime RTOS-32	QNX 4	QNX 6	QNX 7 (32-/64-Bit)	RTOS-UH	VxWorks 5.4/5.5/6.x	VxWorks 7
EtherCAN	C.2050.xx		-	-	-	-	-	-	-
EtherCAN/2	C.2051.xx		-	-	-	-	-	-	-
CAN-USB/Mini	C.2064.xx		-	-	-	-	-	-	-
CAN-USB/Micro CAN-USB/2	C.2068.xx C.2066.xx		5, 6, 7, 8, 9, 11, 12, 13, 14	-	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16	-	-	-
CAN-USB/400	C.2069.xx		-	-	-	-	-	-	-
CAN-USB/3-FD	C.2076.62		-	-	-	-	-	-	-
CAN-ISA/200 CAN-PC104/200 (SJA1000 version)	C.2011.xx C.2013.xx		-	+	5, 6, 7, 8, 9, 11, 12, 13, 14	-	-	5, 6 ,8 9, 11, 12, 13, 14,	-
CAN-ISA/331 CAN-PC104/331	C.2010.xx C.2012.xx		-	10	5, 6, 7, 8, 9, 10, 12,	-		5, 6 ,8 9, 10	-
CAN-PC104/200 (82527 version)	C.2013.xx		-	+	5, 6, 7, 8, 9, 11, 12	-	-	5, 6, 8, 9, 11, 12, 14	-
CAN-PCI/200 CAN-PCIe/200 CPCI-CAN/200 CAN-PCI104/200 CAN-PCI/266 PMC-CAN/266	C.2021.xx C.2042.xx C.2035.xx C.2064.xx C.2036.xx C.2040.xx		5, 6, 7, 8, 9, 11, 12, 13, 14, 16	-	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6, 7, 8, 9, 11, 12, 13, 14	-	5, 6, 8, 9, 11, 12, 13, 14	-
CAN-PCI/331 CPCI-CAN/331 PMC-CAN/331	C.2020.xx C.2027.xx C.2025.xx		-	10	5, 6, 7, 8, 9, 10, 12,	5, 6, 7, 8, 9, 10,12,	9, 10	5, 8, 9, 10, 12	5, 6, 7, 8, 9, 10,12,
CAN-PCI/360 CPCI-CAN/360	C.2022.xx C.2026.xx		-	-	5, 6, 7, 8, 10, 11, 12, 13, 14, 16	5, 6, 7, 8, 10, 11, 12, 13, 14, 16	-	-	-
CAN-PCI/400 CAN-PCIe/400 CPCI-CAN/400 PMC-CAN/400	C.2048.xx C.2043.xx C.2033.xx C.2047.xx		5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	-
CAN-PCIe/402 CAN-PCI/402 CAN-PCIeMini/402 CPCI-CAN/402 CPCIserial-CAN/402	C.2045.xx C.2049.xx C.2044.0x I.2332.xx I.3001.xx	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 20, 21	-	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 20, 21	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 20, 21	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
CAN-PCI/402-FD CAN-PCIe/402-FD CAN-PCIeMini/402-FD CAN-PCIeMiniHS/402 CAN-M.2/402-2-FD CPCIserial-CAN/402-FD PMC-CAN/402-FD XMC-CAN/402-FD	C.2049.xx C.2045.6x C.2044.6x C.2054.6x C.2074.6x I.3001.6x C.2028.xx C.2018.xx	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19, 20, 21	-	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19, 20, 21	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19, 20, 21	-	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19
CAN-PCI/405	C.2023.xx		-	-	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	-	-	-
VME-CAN2	V.1405.xx		-	-	-	-	+	-	-
VME-CAN4	V.1408.xx		-	-	-	-	-	5, 8, 9, 10, 12	-

Table 12: CAN driver capabilities (Real-time operating systems)

The NTCAN-API is also supported on many ***esd electronics*** embedded CPU boards with local CAN controllers. The table below gives an overview for the different supported operating systems.

Embedded CPU	Order no.	Local Operating Systems			
		Linux	**VxWorks**	**QNX6**	**RTOS-UH**
CAN-CBX-CPU5202 CAN-CBX-CPU5201	C.307x.xx	5, 6, 7, 8, 9, 11, 12, 13, 14,16	-	5, 6, 7, 8, 9, 11, 12, 13, 14,16	-
PMC-CPU/440	V.2027.xx	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	-
PMC-CPU/405	V.2020.xx	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6 ,8 9, 11, 12, 13, 14	5, 6, 7, 8, 9, 11, 12, 13, 14	9
CPCI-405	I.2306.04	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6 ,8 9, 11, 12, 13, 14,	5, 6, 7, 8, 9, 11, 12, 13, 14	9
CPCI-CPU/750	I.2402.xx	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6 ,8 9, 11, 12, 13, 14,	-	9
EPPC-405	I.2001.xx	5, 6, 7, 8, 9, 11, 12, 13, 14	5, 6 ,8 9, 11, 12, 13, 14,	5, 6, 7, 8, 9, 11, 12, 13, 14	9
EPPC-405-HR	I.2006.xx	5, 6, 7, 8, 9, 11, 12, 13, 14	-	5, 6, 7, 8, 9, 11, 12, 13, 14	9
EPPC-405-UC	I.2005.xx	5, 6, 7, 8, 9, 11, 12, 13, 14	-	5, 6, 7, 8, 9, 11, 12, 13, 14	-

Table 13: CAN driver capabilities (esd electronics embedded CPU boards)

4. API Reference

This chapter describes each NTCAN API function logically grouped into the sections

- Initialization and Cleanup.
- Configuration
- Receiving Data
- Transmitting Data
- Miscellaneous

Each API function documentation is structured identically into a description of

- Syntax
- Functionality
- Arguments
- Return Values
- Usage
- Requirements
- Further References

Arguments
In each function description the arguments are described in a tabular format. A usage type (see table below) in squared brackets is followed by the description of the argument usage in the specific function.

Usage Type	Meaning
[in]	Indicates the parameter is input. The function reads from the buffer. The caller provides the buffer and initializes it.
[out]	Indicates the parameter is output. The function writes to the buffer. The caller provides the buffer and the function initializes it.
[in/out]	Indicates the parameter is input and output. The caller provides the buffer and initializes it. The function both reads from and writes to the buffer.

Table 14: Parameter Usage Types

4.1 Initialization and Cleanup

This section describes the functions available to establish and release a logical link to a physical CAN port.

4.1.1 canOpen

The function establishes a logical link to a physical CAN port, defines the operation mode, the message type filter, the handle queue sizes and I/O timeouts.

Syntax:

```
NTCAN_RESULT canOpen
(
        int           net,          /* Net number                      */
        uint32_t      flags,        /* Mode flags                      */
        int32_t       txqueuesize,  /* # of entries in message queue   */
        int32_t       rxqueuesize,  /* # of entries in message queue   */
        int32_t       txtimeout,    /* tx-timeout in milliseconds      */
        int32_t       rxtimeout,    /* rx-timeout in milliseconds      */
        NTCAN_HANDLE *handle        /* CAN handle                      */
);
```

Description:

The function establishes a logical link to a physical CAN port by returning a CAN handle on success which is an input parameter for nearly all NTCAN-API functions described in this chapter. Every CAN handle represents a virtual CAN controller with an individual I/O configuration which is independent from other handles apart from common characteristics of the referenced physical CAN port (e.g. bit-rate and error counters).

The maximum number of available handles is limited by the driver and operating system specific global or per process limits.

An application can open several handle to the same physical CAN port with different modes of operation or configuration as well as to different physical CAN ports. Since driver revision 2.x every handle can by used for a full-duplex communication so it is possible to read from this handle in one thread while it is used for transmission in another thread at the same time.

Arguments:

net

[in] The logical net number which is assigned to the physical CAN port in the range from 0 to `NTCAN_MAX_NETS`.

flags

[in] This parameter is a bit mask which defines the basic operation mode and the message type filter. This configuration can not be reconfigured at runtime without closing and reopening the handle.

Flag	Description
NTCAN_MODE_OBJECT	If this flag is set, the handle is configured to receive data in the *Rx Object Mode* instead of the default *FIFO Mode*. The transmission of CAN data is not affected by this flag. In *Object Mode* all message type filter flags are without any effect.
NTCAN_MODE_NO_RTR	This flag configures the message type filter of the handle to discard all received *Remote Request* (RTR) CAN frames.
NTCAN_MODE_NO_DATA	This flag configures the message type filter of the handle to discard all received CAN data frames.
NTCAN_MODE_NO_INTERACTION	This flag configures the message type filter of the handle to discard all CAN messages received via the interaction mechanism.
NTCAN_MODE_MARK_INTERACTION	This flag configures the message type filter of the handle to mark all CAN messages received in FIFO mode via the interaction mechanism in the *mode* bit of the member *len* of CMSG or CMSG_T structure.
NTCAN_MODE_LOCAL_ECHO	This flag configures the message type filter to store frames in the receive queue of the handle via the interaction mechanism even if they have been sent on this handle independent from the configuration of the CAN-ID or Message Type Filter. The messages are marked as described above for the mode flag NTCAN_MODE_MARK_INTERACTION.
NTCAN_MODE_TIMESTAMPED_TX	If this flag is set the timestamp of a CMSG_T structure defines the point of time this message is sent with ***canSendT()***. If the flag is not set or the driver does not support the *Timestamped Tx* mode (see chapter 3.14) all messages are transmitted immediately.
NTCAN_MODE_FD	This flag is required to operate the CAN handle in the CAN FD mode (if supported by the CAN controller). If this flag is not set received CAN FD messages will be discarded on this handle and FD messages to be transmit will be turned into CAN CC frames without further notice for backward compatibility.
NTCAN_MODE_LIN	This flag is required to open a handle in LIN mode. The feature requires LIN physics and the usage is reserved by esd electronics. It is documented here just for completeness. LIN application developer have to use the dedicated LIN API.
NTCAN_MODE_OVERLAPPED	**Only Windows OS (unsupported by RTX / INtime):** This flag opens the handle for asynchronous (also called 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.

Table 15: Mode flags of canOpen()

txqueuesize

[in] Size of the transmit queue in number of CAN messages. The maximum size is limited to the platform specific `NTCAN_MAX_TX_QUEUESIZE`. If no queue is required the value can be set to `NTCAN_NO_QUEUE`. Passing `NTCAN_NO_QUEUE` assigns the device driver specific minimum queue size so transmitting data is possible even than. The queue size can not be reconfigured at runtime without closing and reopening the handle.

rxqueuesize

[in] Size of the receive queue in number of CAN messages. The maximum size is limited to the platform specific `NTCAN_MAX_RX_QUEUESIZE`. If no queue is required the value can be set to `NTCAN_NO_QUEUE`. Passing `NTCAN_NO_QUEUE` assigns the device driver specific minimum queue size so receiving data is possible even than The queue size can not be reconfigured at runtime without closing and reopening the handle.

txtimeout

[in] Timeout in milliseconds for a blocking transmit request with ***canWrite()*** / ***canWriteT()*** / ***canWriteX()***. If a transmit request can not be completed within this configured timeout the request is aborted by the driver and the call will return with a timeout error. If the timeout is set to 0 a transmit request on this handle will be started without timeout and will consequently return with an error code only if aborted by the application or if a bus error occurs. The handle transmit timeout might be changed at runtime before the next transmit request without closing the handle using ***canIoctl()***.

If the timeout value is below a driver or operating system specific minimum value the timeout is set without notice to this minimum. An application can use ***canIoctl()*** with `NTCAN_IOCTL_GET_TX_TIMEOUT` to read the configured value.

rxtimeout

[in] Timeout in milliseconds for a blocking receive request with ***canRead()*** / ***canReadT()*** / ***canReadX().*** If no data is received within this configured timeout the request will return with a timeout error. If the timeout is set to 0 a receive request on this handle will be started without timeout and will consequently return with an error code only if aborted by the application. The handle receive timeout might be changed at runtime before the next receive request without closing the handle using ***canIoctl()***.

If the timeout value is below a driver or operating system specific minimum value the timeout is set without notice to this minimum. An application can use ***canIoctl()*** with `NTCAN_IOCTL_GET_RX_TIMEOUT` to read the configured value.

handle

[out] Pointer to a memory location where the CAN driver will store the CAN handle on success.

Return Values:

On success, the function returns `NTCAN_SUCCESS`. On error, one of the error codes described in chapter 7.

Usage:

The function has to be called before any other function described in this chapter because the returned CAN handle is the input argument for nearly all NTCAN-API functions.

Remark:

If the board contains LIN ports the device driver will also assign them a logical net number. To prevent that a CAN application uses inadvertently a LIN port opening it will return with the error code `NTCAN_NO_CAN_CAPABILITY` instead of `NTCAN_NET_NOT_FOUND` to indicate that there is a port available (a logical net number in use) which can not be used for CAN communication.

Requirements:

N/A.

See also:

Further information on the handle returned by this function can be found in the description of the data structure `NTCAN_HANDLE`.

4.1.2 canClose

Close the link to the physical CAN port.

Syntax:

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

Description:

The function closes the link to the physical CAN port. As a consequence all handle specific resources are released. For CAN-IDs which are still enabled in the handle filter mask ***canIdDelete()*** is called implicitly.

Non-blocking pending transmit requests with ***canSend()*** / ***canSendT()*** are not guaranteed to be completed when closing the handle.

Blocking receive requests on this handle will return with an error in case of a kernel mode CAN driver. For a user mode CAN driver this behavior can not be guaranteed and the application has to take precautions for this situation.

Arguments:

handle

[in] CAN handle.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

N/A.

Requirements:

A valid CAN handle.

See also:

Description of ***canOpen()***.

4.2 Configuration

This section describes the functions available to configure the CAN communication in the **CAN CC** mode.

4.2.1 canSetBaudrate

The function initializes the nominal bit rate of a physical CAN port for the **CAN CC** operation mode.

Syntax:

```
NTCAN_RESULT CALLTYPE canSetBaudrate
(
        NTCAN_HANDLE    handle,      /* CAN handle                  */
        uint32_t        baud         /* Bit rate to be set          */
);
```

Description:

This function configures the nominal bit rate as described in detail in chapter 3.3. A CAN port is passive on the CAN bus until the bit rate is set. A change of the bit rate affects all CAN handle which refer to this physical port.

The system integrator has to verify that all CAN nodes on the bus are set to the same bit rate. Configuring different bit rates will result in CAN communication errors even if the device is not sending any data itself.

Arguments:

handle

[in] CAN handle.

baud

[in] This parameter defines the bit rate. The bits 28..31 of this 32-bit argument are used as configuration flags. The combination of the *User Bit Rate* (UBR) bit 31 and the *User Bit Rate Numerical* (UBRN) bit 29 define the meaning of the value given as bit rate in bit 0..27 of this argument.

If supported by the CAN controller hardware the *Listen Only Mode* (LOM) bit 30 and/or the *Self Test Mode* (STM) bit 28 can be optionally set in addition to the bit rate to enable the respective operation mode.

UBR	LOM	UBRN	STM	27... ...24	23... ...16	15... ...8	7... ...0
0	0/1	0	0/1	Reserved (0)		Bit rate table index	
0	0/1	1	0/1	Reserved (0)	Numerical value in Bit/s		
1	0/1	0	0/1	CAN Controller specific baud timing register values			
1	0/1	1	0/1	Unsupported combination of UBR/UBRN			

The header **<ntcan.h>** defines NTCAN_USER_BAUDRATE for the UBR bit, NTCAN_USER_BAUDRATE_NUM for the UBRN bit, NTCAN_LISTEN_ONLY_MODE for the LOM bit and NTCAN_SELF_TEST_MODE for the STM bit. In addition the special values NTCAN_NO_BAUDRATE to indicate an unconfigured bit rate and NTCAN_AUTOBAUD to initialize an automatic bit rate detection are defined for the parameter *baud*.

Bit rate table index (UBR = 0, UBRN = 0)

The bits 0..15 of baud contain the index of the **esd electronics** bit rate table defined below which is CAN controller and operating system independent. It follows the recommendations of the CiA for the standard bit rates but also contains some intermediate as well as common higher layer CAN protocol bit rates. For the CiA recommended bit rates the header **<ntcan.h>** defines the constants NTCAN_BAUD_XXX where XXX is the bit rate in Kbit/s.

Table index [hex]	Bit Rate [kBit/s]	NTCAN-API Constant
0	1000	NTCAN_BAUD_1000
E*	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*	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

Table 16: **esd electronics** Nominal Bit Rate Table

* This bit rate is not available for all esd electronics CAN boards because of hardware/firmware limitations. For the CAN/405 family the bit rate will have large deviations and for the CAN/331 family in combination with a V2.x driver the bit rate table index is unsupported but the bit rate can be configured as a numerical value.

Numerical bit rate (UBR = 0, UBRN = 1)

The bits 0..23 of baud contain a numerical value in Bit/s and the driver will configure the CAN controller register based on an internal algorithm which first of all tries to minimize the deviation from the given bit rate and chooses and optimized result with respect to several other criteria if more than one CAN controller register configuration is possible.

Due to the algorithm based approach in rare cases the resulting CAN controller configuration register values may differ from the values configured using the **esd electronics** bit rate table for this bit rate.

CAN controller bit rate register (UBR = 1, UBRN = 0)

In this configuration the bits 0..27 of baud contain the values which are programmed directly into the related register of the CAN controller to configure the bit rate.

The values are hardware specific and together with the knowledge of the CAN controller clock frequency you have to refer to the data sheet of the respective CAN controller for details to calculate the resulting bit rate. The required information for this is returned in `NTCAN_INFO` or `NTCAN_BITRATE`.

The table below contains the relation between the bits 0..27 of *baud* and the Bus Timing Registers (BTR) of the CAN controller as they are documented in its user manual.

For CAN FD enabled controller the aggregated controller register sizes of the bit-rate determining values may exceed the overall maximum of 28 bits so some values must be cropped and are set to 0. If direct configuration of these bits is required you must use ***canSetBaudrateX()***.

CAN Controller	Baud (Bit 0..27)	Controller BTR
NXP SJA1000	Bit 0..7 Bit 8..15 Bit 16..27 (Reserved)	BTR1 (Bit 0..7) BTR0 (Bit 0..7)
Intel I82527	Bit 0..7 Bit 8..15 Bit 16..27 (Reserved)	BTR1 (Bit 0..7) BTR0 (Bit 0..7)
Fujitsu MBxxxxx MCU	Bit 0..15 Bit 16..27 (Reserved)	BTR (Bit 0..15)
NXP LPC2xxx	Bit 0..23 Bit 24..27 (Reserved)	BTRxCAN (Bit 0..23)
Freescale MCU (MSCAN)	Bit 0..7 Bit 8..15 Bit 16..27 (Reserved)	CANBTR0 (Bit 0..7) CANBTR1 (Bit 0..7)
Atmel ARM	Bit 0..24 Bit 25..27 (Reserved)	CAN_BR (0..24)
esd Advanced CAN Core (ESDACC)	Bit 0..27	BTR (0..27) (see Annex A:)
ST STM32Fxxx MCU (bxCAN)	Bit 0..25 Bit 26..27 (Reserved)	CAN_BTR (0..25)
Bosch CC770	Bit 0..7 Bit 8..15 Bit 16..27 (Reserved)	BTR1 (Bit 0..7) BTR0 (Bit 0..7)
ST SPEAr320 (C_CAN)	Bit 0..15 Bit 16..27 (Reserved)	BTR (Bit 0..15)
Freescale iMX MCU (FlexCAN)	Bit 0..15 Bit 16..27 (Reserved)	CTRL (16..31)
TI AM335x (Sitara) MCU (D_CAN)	Bit 0..19 Bit 20..27 (Reserved)	BTR (0..19)
Microchip MCP2515	Bit 0..7 Bit 8..15 Bit 16..24 Bit 25..27 (Reserved)	CNF1 (Bit 0..7) CNF2 (Bit 0..7) CNF3 (Bit 0..7)
CAST IP Core	Bit 0..7 Bit 8..13 (Reserved) Bit 14..15 Bit 16 (Reserved) Bit 17..23 Bit 24..27 (Reserved)	Prescaler Register (Bit 0..7) Additional Register (Bit 6..7) Bit Timing Register (Bit 1..7)
Microchip SAM E70/S70/V70/V71 (M_CAN)	Bit 0..5 Bit 6..13 Bit 14..19 Bit 20..27	NBTP (Bit 0..5) NBTP (Bit 8..15) NBTP (Bit 25..30) NBTP (Bit 16..23)

Table 17: CAN Controller Specific Bus Timing Register

Listen-Only Mode (LOM = 1)

Any bit rate configuration above can be combined with the listen-only mode flag (see chapter 3.3.2 for details and requirements) which allows to receive CAN messages without the danger to disturb the CAN bus in case of the wrong bit rate.

Self Test Mode (STM = 1)
Any bit rate configuration above can be combined with the self test mode flag (see chapter 3.3.3 for details and requirements) which allows the CAN controller to receive transmitted messages by itself just as if they were coming from another node without the requirement for an acknowledge.

Automatic bit rate detection
The automatic bit rate detection process is started if *baud* is set to `NTCAN_AUTOBAUD` (see chapter 3.3.7 for details and requirements). An application can follow the automatic bit rate detection process by polling the current bit rate with ***canGetBaudrate()*** or by waiting for the `EV_BAUD_CHANGE` event. The bit rate detection process is cancelled as soon a bit rate is detected or can be aborted by the application calling ***canSetBaudrate()*** again with a valid parameter for *baud* other than `NTCAN_AUTOBAUD`.

Remove from bus
In order to force the CAN hardware to leave the CAN bus *baud* has to be set to `NTCAN_NO_BAUDRATE`. The support of this feature is hardware/firmware dependent (see chapter 3.18).

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

Examples:

```
/* Set bit rate to 500 Kbit/s in listen-only mode with bit rate table */
canSetBaudrate(NTCAN_LISTEN_ONLY_MODE | NTCAN_BAUD_500);

/* Set CAN controller register to 1 Mbit/s directly on SJA1000 (16 MHz) */
canSetBaudrate(NTCAN_USER_BAUDRATE | 0x0014);

/* Set bit rate to 125 Kbit/s as numerical bit rate */
canSetBaudrate(NTCAN_USER_BAUDRATE_NUM | 125000);
```

Requirements:

N/A.

See also:

Use ***canSetBaudrateX()*** to configure the nominal bit rate of a CAN port in the CAN FD mode.

4.2.2 canGetBaudrate

The function returns the configured nominal bit rate of the a CAN port.

Syntax:

```
NTCAN_RESULT canGetBaudrate(
        NTCAN_HANDLE   handle,      /* CAN Handle                          */
        uint32_t       *baud        /* Pointer to store current bit rate */
);
```

Description:

The function returns the nominal bit rate configured for the physical CAN port referenced by the CAN handle.

Arguments:

handle

[in] CAN handle.

baud

[out] Pointer to a memory location where the CAN driver will store the current bit rate on success. The possible values for *baud* are the same as described in the argument list of ***canSetBaudrate()***.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

The function can be used by the application to check which bit rate for the physical CAN port is configured or to poll for the result of the automatic bit rate detection process.

Remark:

If the operation mode of the CAN port was originally configured with ***canSetBaudrateX()*** to CAN CC the device driver returns the appropriate representation for *baud* or returns `NTCAN_BAUD_FD` if configured to **CAN FD**.

Requirements:

N/A.

See also:

Description of ***canSetBaudrate()***. More detailed information about the configured bit rate is returned in `NTCAN_BITRATE` with ***canIoctl()***.

4.2.3 canSetBaudrateX

The function initializes the nominal bit rate and/or data bit rate of a physical CAN port for the CAN FD mode or the CAN CC mode.

Syntax:

```
NTCAN_RESULT CALLTYPE canSetBaudrateX
(
        NTCAN_HANDLE      handle,      /* CAN handle                   */
        NTCAN_BAUDRATE_X  *baud        /* Bit rate configuration       */
);
```

Description:

This function configures the bit rate as described in detail in chapter 3.3. A CAN port is passive on the CAN bus until the bit rate is set. A change of the bit rate affects all CAN handle which refer to this physical port.

The system integrator has to verify that all CAN nodes on the bus are set to the same bit rate. Configuring different bit rates will result in CAN communication errors even if the device is not sending any data itself.

Arguments:

handle

[in] CAN handle.

baud

[in] Reference to a configured `NTCAN_BAUDRATE_X` structure.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

Examples:

```
/*
 * Set nominal bit rate to 500 Kbit/s and the data phase bit rate to 2MBit/s
 * using the esd electronics bit rate table.
 */

NTCAN_RESULT         err;
NTCAN_HANDLE         hnd;
NTCAN_BAUDRATE_X     baud;

   /* Open a CAN handle on net 0 with support for CAN FD operation mode */
err = canOpen(0, NTCAN_MODE_FD, 100, 100, 1000, 1000, &hnd);


baud.mode       = NTCAN_BAUDRATE_MODE_INDEX;
baud.flags      = NTCAN_BAUDRATE_FLAG_FD;
baud.reserved   = 0;
baud.arb.u.idx  = NTCAN_BAUD_500;
baud.data.u.idx = NTCAN_BAUD_2000;

/* Set the bit rate for CAN FD operation mode */
err = canSetBaudrateX(hnd, &baud);
```

Requirements:

A CAN FD enabled CAN controller to configure the CAN FD operation mode.

See also:

canSetBaudrate() and `NTCAN_BAUDRATE_X`.

4.2.4 canGetBaudrateX

The function returns the configured bit rate of the a CAN port.

Syntax:

```
NTCAN_RESULT canGetBaudrateX(
        NTCAN_HANDLE      handle,   /* CAN Handle                        */
        NTCAN_BAUDRATE_X  *baud     /* Bit rate configuration            */
);
```

Description:

The function returns the bit rate configured for the physical CAN port referenced by the CAN handle.

Arguments:

handle

[in] CAN handle.

baud

[out] Pointer to a memory location where the CAN driver will store the current bit rate on success.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

The function can be used by the application to check which operation mode (CAN CC / CAN FD) and nominal bit rate/data bit rate for the physical CAN port.

Remark:

If the operation mode of the CAN port was originally configured with ***canSetBaudrate()*** the device driver creates the appropriate `NTCAN_BAUDRATE_X` representation internally**.**

Requirements:

N/A.

See also:

Description of ***canSetBaudrateX()*** and `NTCAN_BAUDRATE_X`.More detailed information about the configured bit rate is returned in `NTCAN_BITRATE` with ***canIoctl()***.

4.2.5 canIdAdd

The function enables a CAN-ID or Event-ID in the handle message filter.

Syntax:

```
NTCAN_RESULT canIdAdd(
        NTCAN_HANDLE  handle,    /* Handle                          */
        int32_t       id         /* CAN-ID to add to filter         */
);
```

Description:

After a CAN handle is created with ***canOpen()*** the ID filter is cleared (no CAN messages will be received). To receive a CAN message with a certain CAN identifier or an event with a certain Event-ID it is required to enable this ID in the handle filter as otherwise a received message or event is discarded by the driver for this handle.

Because of the CAN message filter implementation for 29-bit CAN-IDs it is sufficient to enable an arbitrary (29-bit) CAN-ID to receive all messages with 29-bit CAN-IDs with this handle.
If the application configures the 29-bit filter as described in chapter 3.8.2.2 for this handle *id* is the acceptance code of this filter.

Configuration and I/O requests for a CAN handle are usually serialized by the device driver. One exception of this rule is the device driver for the CAN-USB/2, CAN-USB/Micro and CAN-AIR/2 where configuration and I/O requests are handled with different USB endpoints to improve I/O performance. To make sure that an I/O request like ***canRead()*** does not overtakes the configuration of the acceptance mask you can e.g. requesting the actual timestamp from the CAN device with ***canIoctl()*** after the last ***canIdAdd()*** request. The timestamp will not be returned before all pending requests are completely processed.

Arguments:

handle

[in] CAN handle.

id

[in] CAN-ID or Event-ID of message to enable for reception on this handle. Valid ranges are:

Range [hex]	Description
0x00..0x7FF	11-bit CAN identifier
0x20000000..0x3FFFFFFF	29-bit CAN Identifier
0x40000000..0x400000FF	Event-ID

Return Values:

Upon success, NTCAN_SUCCESS is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

Example:

```
NTCAN_RESULT result;
int32_t id;

/*
 * Receive all 2.0A CAN-IDs with the handle hnd which was previously
 * opened with canOpen() (not part of this code excerpt)
 */
for(id = 0; id < 0x7FF; id++) {
   result = canIdAdd(hnd, id);
   if(result != NTCAN_SUCCESS) {
         printf("canIdAdd() returned with error %x\n", result);
   }
}
```

Remark:
If the driver implements the *Smart ID Filter* this call is mapped internally to ***canIdRegionAdd().***

Requirements:
N/A.

See also:
Description of ***canIdDelete()***.

4.2.6 canIdRegionAdd

This function enables a range of CAN-IDs (11- or 29-bit) or Event-IDs in the handle message filter (only for the SIF).

Syntax:

```
NTCAN_RESULT canIdRegionAdd(
        NTCAN_HANDLE  handle,    /* Read Handle                          */
        int32_t       idStart,   /* First CAN-ID or Event-ID             */
        int32_t       *idCount   /* IN:  Count of requested ID's         */
                                 /* OUT: Successful selected ID's.       */
);
```

Description:

After a CAN handle is created with ***canOpen()*** the CAN-ID filter is cleared (no CAN messages will pass the filter). To receive a CAN message with a certain CAN-ID or an NTCAN-Event with a certain Event-ID it is required to enable this ID in the handle filter as otherwise a received message or event is discarded by the driver for this handle.

This function enables a consecutive range of IDs which will pass the filter if received. For one physical CAN node the same ID can be selected for any number of handles.

Arguments:

handle

[in] CAN handle.

idStart

[in] First CAN-ID or Event-ID of the consecutive range.

idCount

[in] Number of consecutive IDs to enable.
[out] Number of successful enabled consecutive IDs

The ID range is defined by *idStart* and *idCount*. It must be within the range specified in one of the three valid ID areas:

Range [hex]	Description
0x00..0x7FF	11-bit CAN identifier
0x20000000..0x3FFFFFFF	29-bit CAN Identifier
0x40000000..0x400000FF	NTCAN Event Identifier

Return Values:

Upon success, NTCAN_SUCCESS is returned or one of the error codes described in chapter 7 in case of a failure. As the required memory for the filter depends on the filter configuration NTCAN_INSUFFICIENT_RESOURCES is returned if system limits are exceeded.

Requirements:

CAN driver V 3.9.x or later.

Remark:

The runtime of this function and the resulting resource (memory) requirement is not constant but depends on the current and desired filter configuration.

See also:

Description of ***canIdRegionDelete()***.

4.2.7 canIdDelete

The function disables a CAN-ID or Event-ID in the handle message filter.

Syntax:

```
NTCAN_RESULT canIdDelete(
        NTCAN_HANDLE  handle,    /* Handle                          */
        int32_t       id         /* CAN-ID to add to filter         */
);
```

Description:

This function disables receiving messages with the given CAN-ID or Event-ID on this handle.

Because of the CAN message filter implementation for 29-bit CAN-IDs it is sufficient to disable an arbitrary (29-bit) CAN-ID to stop receiving any messages with a 29-bit CAN-ID on this handle.

Arguments:

handle

[in] CAN handle.

id

[in] CAN-ID or Event-ID of message to disable for reception on this handle. Valid ranges are:

Range [hex]	Description
0x0..0x7FF	11-bit CAN identifier
0x20000000..0x3FFFFFFF	29-bit CAN Identifier
0x40000000..0x400000FF	Event-ID

Return Values:

Upon success, NTCAN_SUCCESS is returned or one of the error codes described in chapter 7 in case of a failure

Usage:

N/A.

Remark:

If the driver implements the *Smart ID Filter* it is is mapped internally to ***canIdRegionDelete()***.

Requirements:

Previously *id* had to be enabled with with ***canIdAdd().***

See also:

Description of ***canIdAdd()*** and ***canClose()***.

4.2.8 canIdRegionDelete

This function disables a range of CAN-IDs (11- or 29-bit) or Event-IDs in the handle message filter (only for the SIF).

Syntax:

```
NTCAN_RESULT canIdRegionDelete(
        NTCAN_HANDLE  handle,    /* Read Handle                            */
        int32_t       idStart,   /* First Rx-CAN-Identifier or Event-ID     */
        int32_t       *idCount   /* IN:  Count of requested ID's           */
                                 /* OUT: Successful selected ID's.         */
);
```

Description:

This function disables a consecutive range of Ids.

Arguments:

handle

[in] CAN handle.

idStart

[in] First CAN-ID or Event-ID of the consecutive range.

idCount

[in] Number of consecutive IDs to disable.
[out] Number of successful disabled consecutive IDs

The ID range is defined by *idStart* and *idCount*. It must be within the range specified in one of the three valid ID areas:

Range [hex]	Description
0x00..0x7FF	11-bit CAN identifier
0x20000000..0x3FFFFFFF	29-bit CAN Identifier
0x40000000..0x400000FF	NTCAN Event Identifier

Return Values:

Upon success, NTCAN_SUCCESS is returned or one of the error codes described in chapter 7 in case of a failure. As the required memory for the filter depends on the filter configuration NTCAN_INSUFFICIENT_RESOURCES is returned if system limits are exceeded.

Requirements:

CAN driver V 3.9.x or later.

Remark:

The runtime of this function and the resulting resource (memory) requirement is not constant but depends on the current and desired filter configuration.

See also:

Description of ***canIdRegionAdd()*** and ***canClose()***.

4.2.9 canIoctl

The function performs a variety of control functions on CAN devices.

Syntax:

```
NTCAN_RESULT canIoctl(
        NTCAN_HANDLE  handle,    /* Handle                          */
        uint32_t      ulCmd,     /* Command specifier               */
        void *        pArg       /* Ptr to command specific argument */
);
```

Description:

This function is an universal entry to configure or request additional CAN I/O configuration. The data type of the input or output data referenced by *pArg*, depends on the control command *ulCmd*. The Usage section contains a list of all supported commands together with their input or output data type.

Arguments:

handle

[in] CAN handle.

ulCmd

[in] Command.

pArg

[in/out] Pointer to *ulCmd* dependent input or output data.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure

Usage:

List of supported commands with their command specific arguments. If a command does not require an argument, *pArg* has to be set to NULL. This list is divided into several sections covering the following functionality:

- CAN communication related I/O controls.
- CAN message filter related I/O controls.
- CAN bus diagnostic related I/O controls.
- CAN message scheduling related I/O controls.
- CAN Error Injection related I/O controls.
- CAN FD TDC and SSP related I/O controls.
- DAR related I/O controls
- Timestamped Tx related I/O controls.
- Miscellaneous I/O controls.

Communication related I/O controls:

This group of commands change or request handle specific CAN-I/O related configuration parameter.

NTCAN_IOCTL_FLUSH_RX_FIFO	Argument: -	N/A
Purges all CAN messages of the handle's receive queue.		

NTCAN_IOCTL_GET_RX_MSG_COUNT	Argument: uint32_t	Out
The number of available CAN messages in the handle's receive queue is stored at the memory location referenced by *pArg.*		

NTCAN_IOCTL_GET_TX_MSG_COUNT	Argument: uint32_t	Out
The number of available CAN messages in the handle's transmit queue is stored at the memory location referenced by *pArg.*		

NTCAN_IOCTL_GET_RX_TIMEOUT	Argument: uint32_t	Out
The receive timeout (in ms) defined for this handle is stored in at the memory location referenced by *pArg.* This value may differ from the value defined in ***canOpen()*** due to OS specific rounding or minimums.		

NTCAN_IOCTL_GET_TX_TIMEOUT	Argument: uint32_t	Out
The transmit timeout (in ms) defined for this handle is stored in at the memory location referenced by *pArg.* This value may differ from the value defined in ***canOpen()*** due to OS specific rounding or minimums.		

NTCAN_IOCTL_SET_RX_TIMEOUT	Argument: uint32_t	In
Re-configure the receive timeout of this handle. The argument is a reference to the previously initialized memory location with the new timeout value in ms. The new value will be used for the next blocking receive function. A pending receive request is not affected by this change.		

NTCAN_IOCTL_SET_TX_TIMEOUT	Argument: uint32_t	In
Re-configure the transmit timeout of this handle. The argument is a reference to the previously initialized memory location with the new timeout value in ms. The new value will be used for the next blocking transmit function. A pending transmit request is not affected by this change.		

NTCAN_IOCTL_ABORT_RX	Argument: -	N/A
A pending blocked receive operation on this handle is aborted but the handle is not closed.		

NTCAN_IOCTL_ABORT_TX	Argument: -	N/A
A pending blocked transmit operation on this handle is aborted but the handle is not closed.		

CAN message filter related I/O controls:

This group of commands configure the second stage of the ID acceptance filter (see chapter 3.8 for details).

NTCAN_IOCTL_SET_HND_FILTER	Argument: NTCAN_FILTER_MASK	In
Configure the handle specific AMR and ACR for the second filter stage of the SIF.		

NTCAN_IOCTL_SET_20B_HND_FILTER	Argument: uint32_t	In
Configure the handle specific AMR for the second filter stage handle of the BIF (Only for 29-bit CAN-IDs).		

CAN bus diagnostics related I/O controls:

This group of commands change or request CAN bus diagnostic related parameters for a physical CAN port (see chapter 3.6.2 for details).

NTCAN_IOCTL_SET_BUSLOAD_INTERVAL	Argument: uint32_t	In
Configure the system wide interval time for the bus load event NTCAN_EV_BUSLOAD in ms for the physical CAN port referenced by *handle*. Without any configuration the default interval is 1000 ms. As the timer handler within the driver might work with a fixed interval, the given value might be rounded to the next integral multiple of the timer handler's integral time. An application can check the really configured value with the command NTCAN_IOCTL_GET_BUSLOAD_INTERVAL.		

NTCAN_IOCTL_GET_BUSLOAD_INTERVAL	Argument: uint32_t	Out
The value of the interval time for the bus load event NTCAN_EV_BUSLOAD in ms for the physical CAN port referenced by *handle* is stored at the memory location referenced by *pArg*, if this feature is supported by CAN hardware and/or device driver.		

NTCAN_IOCTL_GET_BUS_STATISTIC	Argument: NTCAN_BUS_STATISTIC	Out
The current CAN bus statistic and diagnostic data for the physical CAN port referenced by *handle* is stored at the memory location referenced by *pArg.* If *pArg* is set to NULL all statistical counter are reset.		

NTCAN_IOCTL_GET_CTRL_STATUS	Argument: NTCAN_CTRL_STATE	Out
The current CAN controller state for the physical CAN port referenced by *handle* is stored at the memory location referenced by *pArg.*		

NTCAN_IOCTL_GET_BITRATE_DETAILS	Argument: NTCAN_BITRATE	Out
Detailed information about the configured bit rate of the physical CAN port referenced by *handle* is stored at the memory location referenced by *pArg.*		

CAN board information related I/O controls:

This group of commands request CAN port specific information.

NTCAN_IOCTL_GET_SERIAL	Argument: uint32_t	Out
The hardware serial number of the CAN board referenced by *handle* is stored at the memory location referenced by *pArg* (if supported by the CAN board). As a CAN board can have several physical CAN ports the same serial number is returned for all logical CAN networks related to this board. The serial number is returned in an encoded format. Each of the two upper nibbles of the value represents one of the leading letters of the production lot number (0x0 => 'A' , 0x1 => 'B', …,0xF => 'P'). The remaining 24 bits are the numerical part. Example: The value 0x1D012345 is the serial number BN074565. If reading the serial number is not supported by the device a 0 is returned which results in the serial number AA000000 according to the encoding described above.		

NTCAN_IOCTL_GET_TIMESTAMP_FREQ	Argument: uint64_t	Out
The resolution of the timestamp counter in Hz of the CAN port referenced by *handle* is stored at the memory location referenced by *pArg*, if timestamps are supported by CAN hardware, device driver and operating system.		

NTCAN_IOCTL_GET_TIMESTAMP	Argument: uint64_t	Out
The value of the timestamp counter related to the CAN port referenced by *handle* is stored at the memory location referenced by *pArg* , if timestamps are supported by CAN hardware, device driver and operating system.		

NTCAN_IOCTL_GET_HW_TIMESTAMP	Argument: uint64_t[3]	Out
!! This command is only supported on ESDACC based devices !! The current value of the OS specific local high resolution counter is stored at array element 0, followed by the hardware timestamp related to the CAN port referenced by *handle* stored at array element 1, followed by current value of the local high resolution counter stored at array element 2. The frequency of the OS specific local high resolution counter is returned with NTCAN_INFO.		

NTCAN_IOCTL_GET_HW_TIMESTAMP_EX	Argument: uint64_t[5]	Out
!! This command is only supported on ESDACC based devices !! The first three array elements are identical to the values stored by the command NTCAN_IOCTL_GET_HW_TIMESTAMP followed by the value returned with the command NTCAN_IOCTL_GET_TIMESTAMP stored at array element 3, followed by current value of the local high resolution counter stored at array element 4.		

NTCAN_IOCTL_GET_INFO	Argument: NTCAN_INFO	Out
A comprehensive information about the device and driver environment stored in the NTCAN_INFO structure related to the CAN port referenced by *handle* stored at the memory location referenced by *pArg.*		

Miscellaneous I/O controls:

This group of commands covers requests which do not fit into any of the other sections.

NTCAN_IOCTL_GET_NATIVE_HANDLE	Argument: Native OS handle type	Out

The OS specific handle or file descriptor of the CAN device referenced by *handle* is stored at the memory location referenced by *pArg* (see description of NTCAN_HANDLE for details). The table below contains the native handle type which is returned.

Operating System	Native OS handle type
Windows	HANDLE
Linux	int
QNX	int
LynxOS	int

TX Object Mode related I/O controls:

This group of commands covers all commands which are used to configure the TX Object Mode.

NTCAN_IOCTL_TX_OBJ_CREATE	Argument: CMSG	In
Creates an object with the CAN-ID defined in the CMSG structure referenced by *pArg*, which can be used for scheduling. You will be able to create exactly **one** object per CAN-ID (11-bit and 29-bit) per physical CAN port. For 29-bit CAN-IDs you might be restricted by the available resources, as usually the host system has not enough memory to handle 2^{29} objects.		

NTCAN_IOCTL_TX_OBJ_UPDATE	Argument: CMSG	In
Update CAN data and length for an existing object. The object is referenced by the CAN-ID defined in the CMSG structure referenced by *pArg.* **Note:** In order to update **and** transmit one or more previously created TX Objects in a single step you have to use ***canSend ()*** / ***canSendT ()*** or ***canWrite ()*** / ***canWriteT ()***.		

NTCAN_IOCTL_TX_OBJ_DESTROY	Argument: CMSG	In
Destroys a formerly created object (and therefore stops its scheduling). The object is referenced by the CAN-ID defined in the CMSG structure referenced by *pArg.*		

NTCAN_IOCTL_TX_OBJ_CREATE_X	Argument: CMSG_X	In
Creates an object with the CAN-ID defined in the CMSG_X structure referenced by *pArg*, which can be used for scheduling. You will be able to create exactly **one** object per CAN-ID (11-bit and 29-bit) per physical CAN port. For 29-bit CAN-IDs you might be restricted by the available resources, as usually the host system has not enough memory to handle 2^{29} objects.		

NTCAN_IOCTL_TX_OBJ_UPDATE_X	Argument: CMSG_X	In
Update CAN data and length for an existing object. The object is referenced by the CAN-ID defined in the CMSG_X structure referenced by *pArg.* **Note:** In order to update **and** transmit one or more previously created TX Objects in a single step you have to use ***canSendX()*** / ***canWriteX()***.		

NTCAN_IOCTL_TX_OBJ_DESTROY_X	Argument: CMSG_X	In
Destroys a formerly created object (and therefore stops its scheduling). The object is referenced by the CAN-ID defined in the CMSG_X structure referenced by *pArg.*		

NTCAN_IOCTL_TX_OBJ_SCHEDULE	Argument: CSCHED	In
Configures the scheduling for an existing object. If there are several objects scheduled for the same time, the order of these NTCAN_IOCTL_TX_OBJ_SCHEDULE commands will define the order of transmission. This command is only processed as long as scheduling is NOT running to prevent race conditions with transmission order.		

NTCAN_IOCTL_TX_OBJ_SCHED_START	Argument: -	N/A
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 periodically 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	Argument: -	N/A
Disables the scheduling. In order to prevent a non-deterministic transmission order caused by configuration changes, this includes deletion of all schedules of this CAN-handle. The existing TX Objects once created are persistent keeping their configuration, but before calling NTCAN_IOCTL_TX_OBJ_SCHED_START again, a new schedule configuration has to be assigned with NTCAN_IOCTL_TX_OBJ_SCHEDULE.		

NTCAN_IOCTL_TX_OBJ_AUTOANSWER_ON	Argument: CMSG	In
Enable the autoanswer mode for an existing object. In this mode the initialized CAN message will be sent automatically by the driver every time a related RTR is received. Autoanswer can be enabled independent from scheduling for each object. The object is referenced by the CAN-ID defined in the CMSG structure referenced by *pArg.* **Note:** As the RTR concept is only supported in CAN CC and not in CAN FD there is no CMSG_X support.		

NTCAN_IOCTL_TX_OBJ_AUTOANSWER_OFF	Argument: CMSG	In
Disable the autoanswer mode for an existing object. Autoanswer can be disabled independent from scheduling for each object. The object is referenced by the CAN-ID defined in the CMSG structure referenced by *pArg.* **Note:** As the RTR concept is only supported in CAN CC and not in CAN FD there is no CMSG_X support.		

NTCAN_IOCTL_TX_OBJ_AUTOANSWER_ONCE	Argument: CMSG	In
Identical behavior as described for NTCAN_IOCTL_TX_OBJ_AUTOANSWER_ON with the difference that the reply is sent exactly once after an object update. **Note:** As the RTR concept is only supported in CAN CC and not in CAN FD there is no CMSG_X support.		

Error Injection related I/O controls:

This group of commands covers all commands which are used to configure the Error Injection Module.

NTCAN_IOCTL_EEI_CREATE	Argument: uint32_t	Out
Allocate an Error Injection Unit and bind it to an esdACC CAN Controller. A Handle to this Unit will be returned by the argument.		

NTCAN_IOCTL_EEI_DESTROY	Argument: uint32_t	In
Free the Error Injection Unit. The handle defined for this Error Injection Unit is stored in at the memory location referenced by *pArg.*		

NTCAN_IOCTL_EEI_STATUS	Argument: NTCAN_EEI_STATUS	In
Checks the Status of an Error Injection Unit. If there is no valid handle in NTCAN_EEI_STATUS, the ioctl will return units_total and units_free without an error.		

NTCAN_IOCTL_EEI_CONFIGURE	Argument: NTCAN_EEI_UNIT	In
Configures the Error Injection Unit.		

NTCAN_IOCTL_EEI_START	Argument: uint32_t	In
Enables the Error Injection Unit. The handle defined for this Error Injection Unit is stored in at the memory location referenced by *pArg.*		

NTCAN_IOCTL_EEI_STOP	Argument: uint32_t	In
Disables the Error Injection Unit. The handle defined for this Error Injection Unit is stored in at the memory location referenced by *pArg.*		

NTCAN_IOCTL_EEI_TRIGGER_NOW	Argument: uint32_t	In
Sets trigger now and the CAN TX Module will send the TX Pattern with the next TX Point. The handle defined for this Error Injection Unit is stored in at the memory location referenced by *pArg.*		

Timestamped TX related I/O controls:

This group of commands covers all commands which are used in conjunction with *Timestamped TX*. These should be used only, if the feature is available and the respective feature bit is set (`FEATURE_TIMESTAMPED_TX`, s. 179).

`NTCAN_IOCTL_SET_TX_TS_WIN`	Argument: `uint32_t`	In
Set the size of the planning window for *Timestamped TX* in ms.		

`NTCAN_IOCTL_GET_TX_TS_WIN`	Argument: `uint32_t`	Out
Returns the size of the *Timestamped TX* planning window in ms.		

`NTCAN_IOCTL_SET_TX_TS_TIMEOUT`	Argument: `uint32_t`	In
This I/O control is available on special CAN hardware, only. Configure a frame timeout for *Timestamped TX* in timestamp ticks. Please note, the maximum configurable timeout varies depending on timestamp frequency of the CAN hardware in use. 0: No timeout n: Timeout, if the frame has not been transmitted after n timestamp ticks (measured from the scheduled transmission point). See chapter 3.9 for details about the timestamp implementation.		

`NTCAN_IOCTL_GET_TX_TS_TIMEOUT`	Argument: `uint32_t`	Out
Returns the configured frame timeout, which is used for *Timestamped TX*. See chapter 3.9 for details about the timestamp implementation.		

CAN FD Transmitter Delay Compensation (TDC) related I/O controls:

This group of commands covers all commands which are used in conjunction with the Transmitter Delay Compensation (TDC) and Second Sample Point (SSP). They are CAN FD specific and so only available for CAN FD capable hardware (Feature bit `FEATURE_CAN_FD`, s. 179).

`NTCAN_IOCTL_GET_FD_TDC`	Argument: `uint32_t`	Out
Returns the current TDC configuration (see 3.15.2.3 for details on the argument of the call).		

`NTCAN_IOCTL_SET_FD_TDC`	Argument: `uint32_t`	In
Deprecated command to configure the TDC mechanism (see 3.15.2.3 for details on the argument of the call). An application should use ***canSetBaudrateX()*** instead.		

Disable Automatic Retransmission (DAR) related I/O controls:

This group of commands covers all commands which are used in conjunction with the *Disable Automatic Retransmission* (DAR) capability of the CAN hardware. (Feature bit `NTCAN_FEATURE_DAR` and `NTCAN_FEATURE_DAR_FRAME`, s. 179).

`NTCAN_IOCTL_SET_DAR_MODE`	Argument: `uint32_t`	In
Sets the DAR mode configuration mask. • If the NTCAN_DAR_DISABLE_ON_ARB_LOST bit is set, the CAN message will be retransmitted although the arbitration was lost and DAR is enabled. • If NTCAN_DAR_DISABLE_ON_TX_ERROR bit is set, the CAN message will be retransmitted although the transmission failed wit an error and DAR is enabled. The default value for this configuration option is that each bit is reset. Any change is applied immediately to the next failed transmission with enabled DAR mode. **Note:** This IOCTL command option is currently only supported by the ESDACC controller.		

`NTCAN_IOCTL_SET_DAR_MODE`	Argument: `uint32_t`	Out
Returns the current value of the DAR mode configuration mask. Refer to `NTCAN_IOCTL_SET_DAR_MODE` above for supported options. **Note:** This IOCTL command option is currently only supported by the ESDACC controller.		

I/O Configuration related I/O controls:

This group of commands change or request configuration parameter of physical I/Os.

NTCAN_IOCTL_SET_TERM_CFG	Argument: uint32_t	In
If the CAN hardware supports a programmatically switchable CAN bus termination option (indicated with NTCAN_FEATURE_PROG_TERM), this command configures the state of termination resistor of the physical port referenced by *handle.* The value NTCAN_TERM_DISABLE deactivates the CAN bus termination and the value NTCAN_TERM_ENABLE activates it.		

NTCAN_IOCTL_GET_TERM_CFG	Argument: uint32_t	Out
If the CAN hardware supports a programmatically switchable CAN bus termination option (indicated with NTCAN_FEATURE_PROG_TERM), this command returns the state of the physical port referenced by *handle.* The returned values to indicate an activated/deactivated bus termination resistor are described in NTCAN_IOCTL_SET_TERM_CFG.		

NTCAN_IOCTL_SET_GPIO_CFG	Argument: NTCAN_GPIO_CFG	In
If the CAN hardware supports optional GPIO ports (indicated with NTCAN_FEATURE_GPIO), this command allows the confguration of each I/O channel with an initialized NTCAN_GPIO_CFG structure. **Note:** This IOCTL command is only supported with a handle opened for the base net of the respective CAN hardware which logical net number is returned in the NTCAN_INFO structure.		

NTCAN_IOCTL_GET_GPIO_CFG	Argument: NTCAN_GPIO_CFG	Out
If the CAN hardware supports optional GPIO ports (indicated with NTCAN_FEATURE_GPIO), this command returns the current confguration of each I/O channel in a NTCAN_GPIO_CFG structure. **Note:** This IOCTL command is only supported with a handle opened for the base net of the respective CAN hardware which logical net number is returned in the NTCAN_INFO structure.		

Reserved I/O controls:

This group of commands is reserved for internal use by esd electronics and just documented for completeness.

NTCAN_IOCTL_LIN_MASTER_SEL	Argument: N/A	N/A
N/A		

Requirements:
N/A.

See also:
N/A.

4.3 Receiving CAN messages

This section describes the functions available to receive CAN / CAN FD messages and CAN events. The API offers services to receive data in a blocking (event based) and non-blocking (polling) way with or without timestamps.

4.3.1 canTake

Non-blocking reception of CAN messages and CAN events without a timestamp.

Syntax:

```
NTCAN_RESULT canTake(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CMSG          *cmsg,     /* Ptr to application buffer       */
             int32_t       *len );    /* OUT: Size of CMSG-Buffer        */
                                      /* IN:  Number of received messages */
```

Description:

The function returns available CAN messages or CAN events for this handle in a non-blocking way (polling). The behaviour of the function is different for the *FIFO Mode* and *Object Mode*.

FIFO-Mode

For a handle configured with ***canOpen()*** to the (default) *FIFO-Mode*, received CAN messages or CAN events stored in the Rx FIFO of this handle are copied into the application buffer referenced by *cmsg* in the sequential order of their reception. Every copied message is removed from the handle Rx FIFO afterwards. The maximum size of the application buffer has to be stored in *len* as multiple of `CMSG` objects before calling ***canTake()***. Upon return, *len* contains the number of `CMSG` objects copied into the application buffer.

Object-Mode

For a handle configured with ***canOpen()*** to the *Object-Mode*, the caller has to initialize the member *id* of all `CMSG` objects in the application buffer referenced by *cmsg* with the CAN message identifiers of interest. The size of the application buffer has to be stored in *len* as multiple of `CMSG` objects before calling ***canTake()***. Upon return, the application buffer is filled with the most recently received CAN messages for the requested CAN-IDs. To indicate that no data has yet been received for a requested CAN identifier, the `NTCAN_NO_DATA` bit is set in the member *len* of the `CMSG` object in the application buffer.

In *Object-Mode* the call is limited to return CAN messages with standard CAN identifiers (11-bit) for driver version V2.x. Extended CAN identifier (29-bit) support was introduced with driver V3.x (see chapter 3.11.2. for the driver version specific differences to initialize the object mode). CAN events in object mode are unsupported by all driver versions.

Arguments:

handle

[in] CAN handle.

cmsg

[in/out] Pointer to the application buffer as array of `CMSG` objects to store the received CAN messages or CAN events. In *Object-Mode* the *id* member of each single `CMSG` object of this array has to be initialized to the requested (11-bit) CAN-ID before the call.

If the `NTCAN_EV_BASE` bit (Bit 30) is set in the member *id* of a received `CMSG` object to indicate that this is a CAN event the application can cast the data to an `EVMSG` object for further processing.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG` objects which can be stored in the buffer referenced by *cmsg*. Upon return, the driver has stored the number of messages copied into the application buffer into this parameter.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

As a non-blocking call, the return value is `NTCAN_SUCCESS` even if no CAN messages are received (copied into the application buffer). For this reason upon return an application should always check the parameter *len*.

Usage:

The call is intended for applications which poll (cyclically) for new CAN messages or CAN events without the need for timestamps.

Requirements:

To receive CAN messages the controller has to be initialized with ***canSetBaudrate()*** and an appropriate filter has to be configured with ***canIdAdd().***

See also:

Description of ***canIdAdd()*** and ***canOpen()***.

4.3.2 canTakeT

Non-blocking reception of CAN messages and CAN events with timestamp.

Syntax:

```
NTCAN_RESULT canTakeT(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CMSG_T        *cmsg_t,   /* Ptr to application buffer       */
             int32_t       *len);     /* OUT: Size of CMSG_T buffer      */
                                      /* IN:  Number of received messages */
```

Description:

The function returns available timestamped CAN messages or CAN events for this handle in a non-blocking way (polling). The behaviour of the function is different for the *FIFO Mode* and *Object Mode*.

FIFO-Mode

For a handle configured with ***canOpen()*** to the (default) *FIFO-Mode*, received timestamped CAN messages or CAN events stored in the Rx FIFO of this handle are copied into the application buffer referenced by *cmsg_t* in the sequential order of their reception. Every copied message is removed from the handle Rx FIFO afterwards. The maximum size of the application buffer has to be stored in *len* as multiple of `CMSG_T` objects before calling ***canTakeT()***. Upon return, *len* contains the number of `CMSG_T` objects copied into the application buffer.

Object-Mode

For a handle configured with ***canOpen()*** to the *Object-Mode*, the caller has to initialize the member *id* of all `CMSG_T` objects in the application buffer referenced by *cmsg_t* with the CAN message identifiers of interest. The size of the application buffer has to be stored in *len* as multiple of `CMSG_T` objects before calling ***canTakeT()***. Upon return, the application buffer is filled with the most recently received CAN messages for the requested CAN-IDs. To indicate that no data has yet been received for a requested CAN identifier, the `NTCAN_NO_DATA` bit is set in the member *len* of the `CMSG` object in the application buffer.

In *Object-Mode* the call is limited to return CAN messages with standard CAN identifiers (11-bit) for driver version V2.x. Extended CAN identifier (29-bit) support was introduced with driver V3.x (see chapter 3.11.2. for the driver version specific differences to initialize the object mode). CAN events in object mode are unsupported by all driver versions.

Arguments:

handle

[in] CAN handle.

cmsg_t

[in/out] Pointer to the application buffer as array of `CMSG_T` objects to store the received timestamped CAN messages and CAN events. In *Object-Mode* the *id* member of each single `CMSG_T` object of this array has to be initialized to the requested (11-bit) CAN-ID before the call.

If the `NTCAN_EV_BASE` bit (Bit 30) is set in the member *id* of a received `CMSG_T` object to indicate that this is a CAN event the application can cast the data to an `EVMSG_T` object for further processing.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_T` objects which can be stored in the buffer referenced by ***cmsg_t***. Upon return, the driver has stored the number of messages copied into the application buffer into this parameter.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

As a non-blocking call, the return value is `NTCAN_SUCCESS` even if no CAN messages are received (copied into the application buffer). For this reason upon return an application should always check the parameter *len*.

Usage:

The call is intended for applications which poll (cyclically) for new timestamped CAN messages or CAN events.

Requirements:

The CAN driver or CAN hardware has to support time stamping which is indicated with the `NTCAN_FEATURE_TIMESTAMP` flag in `CAN_IF_STATUS`. To receive timestamped data the CAN controller has to be initialized with ***canSetBaudrate()*** and an appropriate filter has to be configured with ***canIdAdd().***

See also:

Description of ***canIdAdd()*** and ***canOpen()***.

4.3.3 canTakeX

Non-blocking reception of CAN / CAN FD messages and CAN events with timestamp.

Syntax:

```
NTCAN_RESULT canTakeX(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CMSG_X        *cmsg_x,   /* Ptr to application buffer       */
             int32_t       *len);     /* OUT: Size of CMSG_X buffer      */
                                      /* IN:  Number of received messages */
```

Description:

The function returns available timestamped CAN / CAN FD messages or CAN events for this handle in a non-blocking way (polling). The behaviour of the function is different for the *FIFO Mode* and *Object Mode*. The handle has to be opened in CAN FD Mode.

FIFO-Mode

For a handle configured with ***canOpen()*** to the (default) *FIFO-Mode*, received timestamped CAN / CAN FD messages or CAN events, stored in the Rx FIFO of this handle, are copied into the application buffer referenced by *cmsg_x* in the sequential order of their reception. Every copied message is removed from the handle Rx FIFO afterwards. The maximum size of the application buffer has to be stored in *len* as multiple of `CMSG_X` objects before calling ***canTakeX()***. Upon return, *len* contains the number of `CMSG_X` objects copied into the application buffer.

Object-Mode

For a handle configured with ***canOpen()*** to the *Object-Mode*, the caller has to initialize the member *id* of all `CMSG_X` objects in the application buffer referenced by *cmsg_x* with the CAN message identifiers of interest. The size of the application buffer has to be stored in *len* as multiple of `CMSG_X` objects before calling ***canTakeX()***. Upon return, the application buffer is filled with the most recently received CAN messages for the requested CAN-IDs. To indicate that no data has yet been received for a requested CAN identifier, the `NTCAN_NO_DATA` bit is set in the member *len* of the `CMSG_X` object in the application buffer.

In *Object-Mode* the call is limited to return CAN messages with standard CAN identifiers (11-bit) for driver version V2.x. Extended CAN identifier (29-bit) support was introduced with driver V3.x (see chapter 3.11.2. for the driver version specific differences to initialize the object mode). CAN events in object mode are unsupported by all driver versions.

Arguments:

handle

[in] CAN handle opened in CAN FD mode.

cmsg_x

[in/out] Pointer to the application buffer as array of `CMSG_X` objects to store the received timestamped CAN messages and CAN events. In *Object-Mode* the *id* member of each single `CMSG_X` object of this array has to be initialized to the requested (11-bit) CAN-ID before the call.

If the `NTCAN_EV_BASE` bit (Bit 30) is set in the member *id* of a received `CMSG_X` object to indicate that this is a CAN event the application can cast the data to an `EVMSG_T` object for further processing.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_X` objects which can be stored in the buffer referenced by *cmsg_x*. Upon return, the driver has stored the number of messages copied into the application buffer into this parameter**.**

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

As a non-blocking call, the return value is `NTCAN_SUCCESS` even if no CAN messages are received (copied into the application buffer). For this reason upon return an application should always check the parameter *len*.

Usage:

The call is intended for applications which poll (cyclically) for new timestamped CAN messages or CAN events.

Requirements:

The CAN driver or CAN hardware has to support CAN FD which is indicated with the `NTCAN_FEATURE_FD` flag in `CAN_IF_STATUS`. To receive CAN / CAN FD messages the CAN FD controller has to be initialized with ***canSetBaudrate()*** and an appropriate filter has to be configured with ***canIdAdd().***

See also:

Description of ***canIdAdd()*** and ***canOpen()***.

4.3.4 canRead

Blocking reception of CAN messages and CAN events without timestamp.

Syntax:

```
NTCAN_RESULT canRead(
             NTCAN_HANDLE  handle,    /* Handle                              */
             CMSG          *cmsg,     /* Ptr to application buffer           */
             int32_t       *len,      /* OUT: Size of CMSG-Buffer            */
                                      /* IN:  Number of received messages */
             OVERLAPPED    *ovrlppd); /* NULL or overlapped-structure        */
```

Description:

The function returns available data for this handle immediately or blocks until

- New CAN data or CAN events are received.
- The configured receive timeout is exceeded.
- The I/O operation is aborted with ***canIoctl()*** and the command `NTCAN_IOCTL_ABORT_RX`.
- The handle is closed with ***canClose()***.
- Change of system or device state like e.g. a power state change.

Received CAN messages or CAN events stored in the Rx FIFO of the this handle are copied into the application buffer referenced by *cmsg* in the sequential order of their reception. Every copied message is removed from the handle Rx FIFO afterwards. The maximum size of the application buffer has to be stored in *len* as multiple of `CMSG` objects before calling ***canRead()***. Upon return, *len* contains the number of `CMSG` objects copied into the application buffer.

Win32/64

On Windows the CAN driver supports the asynchronous (overlapped) I/O extension which was introduced with Windows NT.
If the flag `NTCAN_MODE_OVERLAPPED` is **not** set in ***canOpen()*** for this handle the driver performs a synchronous I/O operation which is described above and the parameter *ovrlppd should* be set to 0.
If the flag `NTCAN_MODE_OVERLAPPED` is set in ***canOpen()*** for this handle the driver performs an asynchronous I/O operation and the driver returns immediately even if data is available. The status of the I/O operation has to be retrieved with ***canGetOverlappedResult()***.

Arguments:

handle

[in] CAN handle.

cmsg

[out] Pointer to the application buffer as array of `CMSG` objects to store the received CAN messages or CAN events.

If the `NTCAN_EV_BASE` bit (Bit 30) is set in the member *id* of a received `CMSG` object to indicate that this is a CAN event the application can cast the data to an `EVMSG` object for further processing.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG` objects which can be stored in the buffer referenced by *cmsg*. Upon return, the driver has stored the number of messages copied into the application buffer into this parameter.

ovrlppd

[in] This parameter is used to support Windows asynchronous I/O operations and is ignored by all other supported operating systems. For Windows the pointer has to be set to a valid and unique `OVERLAPPED` structure if the handle is opened with the flag `NTCAN_MODE_OVERLAPPED`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

As a blocking call does not support the *Object-Mode* a handle opened in this mode will return from this call with `NTCAN_INVALID_PARAMETER`.

Usage:

The call is intended for applications which want to receive CAN messages or CAN events without the need for timestamps in an event based way. For this reason it is ideally suited for a multithreaded implementation where receiving CAN messages can be handled in one or more independent threads.

Example:

```
/*
 * 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;  /* Initialize maximum size of application buffer */
 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);
   }
.
.
```

Requirements:

To receive CAN messages the controller has to be initialized with ***canSetBaudrate()*** and an appropriate filter has to be configured with ***canIdAdd().***

See also:

Description of ***canIdAdd()*** and ***canOpen()***.

4.3.5 canReadT

Blocking reception of timestamped CAN messages and CAN events.

Syntax:

```
NTCAN_RESULT canReadT(
             NTCAN_HANDLE  handle,    /* Handle                              */
             CMSG_T        *cmsg_t,   /* Ptr to application buffer           */
             int32_t       *len,      /* OUT: Size of CMSG_T-Buffer          */
                                      /* IN:  Number of received messages */
             OVERLAPPED    *ovrlppd); /* NULL or overlapped-structure        */
```

Description:

The function returns available data for this handle immediately or blocks until

- New CAN data or CAN events are received.
- The configured receive timeout is exceeded.
- The I/O operation is aborted with ***canIoctl()*** and the command `NTCAN_IOCTL_ABORT_RX`.
- The handle is closed with ***canClose()***.
- Change of system or device state like e.g. a power state change.

Received timestamped CAN messages or CAN events stored in the Rx FIFO of the this handle are copied into the application buffer referenced by *cmsg_t* in the sequential order of their reception. Every copied message is removed from the handle Rx FIFO afterwards. The maximum size of the application buffer has to be stored in *len* as multiple of `CMSG_T` objects before calling ***canReadT()***. Upon return, *len* contains the number of `CMSG_T` objects copied into the application buffer.

Win32/64

On Windows the CAN driver supports the asynchronous (overlapped) I/O extension which was introduced with Windows NT.
If the flag `NTCAN_MODE_OVERLAPPED` is **not** set in ***canOpen()*** for this handle the driver performs a synchronous I/O operation which is described above and the parameter *ovrlppd should* be set to 0.
If the flag `NTCAN_MODE_OVERLAPPED` is set in ***canOpen()*** for this handle the driver performs an asynchronous I/O operation and the driver returns immediately even if data is available. The status of the I/O operation has to be retrieved with ***canGetOverlappedResultT()***.

Arguments:

handle

[in] CAN handle.

cmsg_t

[out] Pointer to the application buffer as array of `CMSG_T` objects to store the received CAN messages or CAN events.

If the `NTCAN_EV_BASE` bit (Bit 30) is set in the member *id* of a received `CMSG_T` object to indicate that this is a CAN event the application can cast the data to an `EVMSG_T` object for further processing.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_T` objects which can be stored in the buffer referenced by *cmsg_t*. Upon return, the driver has stored the number of messages copied into the application buffer into this parameter.

ovrlppd

[in] This parameter is used to support Windows asynchronous I/O operations and is ignored by all other supported operating systems. For Windows the pointer has to be set to a valid and unique `OVERLAPPED` structure if the handle is opened with the flag `NTCAN_MODE_OVERLAPPED`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

As a blocking call does not support the *Object-Mode* a handle opened in this mode will return from this call with `NTCAN_INVALID_PARAMETER`.

Usage:

The call is intended for applications which want to receive timestamped CAN messages or CAN events in an event based way. For this reason it is ideally suited for a multithreaded implementation where receiving CAN messages can be handled in one or more independent threads.

Requirements:

To receive CAN messages the controller has to be initialized with ***canSetBaudrate()*** and an appropriate filter has to be configured with ***canIdAdd().***

See also:

Description of ***canIdAdd()*** and ***canOpen()***.

4.3.6 canReadX

Blocking reception of timestamped CAN / CAN FD messages and CAN events.

Syntax:

```
NTCAN_RESULT canReadX(
             NTCAN_HANDLE  handle,    /* Handle                              */
             CMSG_X        *cmsg_x,   /* Ptr to application buffer           */
             int32_t       *len,      /* OUT: Size of CMSG_X-Buffer          */
                                      /* IN:  Number of received messages */
             OVERLAPPED    *ovrlppd); /* NULL or overlapped-structure        */
```

Description:

The function returns available data for this handle immediately or blocks until

- New CAN /CAN FD messages or CAN events are received.
- The configured receive timeout is exceeded.
- The I/O operation is aborted with ***canIoctl()*** and the command `NTCAN_IOCTL_ABORT_RX`.
- The handle is closed with ***canClose()***.
- Change of system or device state like e.g. a power state change.

Received timestamped CAN / CAN FD messages or CAN events stored in the Rx FIFO of the this handle are copied into the application buffer referenced by *cmsg_x* in the sequential order of their reception. Every copied message is removed from the handle Rx FIFO afterwards. The maximum size of the application buffer has to be stored in *len* as multiple of `CMSG_X` objects before calling ***canReadX()***. Upon return, *len* contains the number of `CMSG_X` objects copied into the application buffer.

Win32/64

On Windows the CAN driver supports the asynchronous (overlapped) I/O extension which was introduced with Windows NT.
If the flag `NTCAN_MODE_OVERLAPPED` is **not** set in ***canOpen()*** for this handle the driver performs a synchronous I/O operation which is described above and the parameter *ovrlppd should* be set to 0.
If the flag `NTCAN_MODE_OVERLAPPED` is set in ***canOpen()*** for this handle the driver performs an asynchronous I/O operation and the driver returns immediately even if data is available. The status of the I/O operation has to be retrieved with ***canGetOverlappedResultX()***.

Arguments:

handle

[in] CAN handle.

cmsg_x

[out] Pointer to the application buffer as array of `CMSG_X` objects to store the received CAN / CAN FD messages or CAN events.

If the `NTCAN_EV_BASE` bit (Bit 30) is set in the member *id* of a received `CMSG_X` object to indicate that this is a CAN event the application can cast the data to an `EVMSG_T` object for further processing.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_X` objects which can be stored in the buffer referenced by *cmsg_x*. Upon return, the driver has stored the number of messages copied into the application buffer into this parameter**.**

ovrlppd

[in] This parameter is used to support Windows asynchronous I/O operations and is ignored by all other supported operating systems. For Windows the pointer has to be set to a valid and unique `OVERLAPPED` structure if the handle is opened with the flag `NTCAN_MODE_OVERLAPPED`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

As a blocking call does not support the *Object-Mode* a handle opened in this mode will return from this call with `NTCAN_INVALID_PARAMETER`.

Usage:

The call is intended for applications which want to receive timestamped CAN messages or CAN events in an event based way. For this reason it is ideally suited for a multithreaded implementation where receiving CAN messages can be handled in one or more independent threads.

Requirements:

To receive CAN messages the controller has to be initialized with ***canSetBaudrate()*** and an appropriate filter has to be configured with ***canIdAdd().***

See also:

Description of ***canIdAdd()*** and ***canOpen()***.

4.4 Transmitting CAN messages

This section describes the functions available to transmit CAN messages. The API offers services to transmit data asynchronously and synchronously.

4.4.1 canSend

Asynchronous transmission of CAN messages.

Syntax:

```
NTCAN_RESULT canSend(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CMSG          *cmsg,     /* Ptr to application buffer       */
             int32_t       *len );    /* OUT: # of messages to transmit */
                                      /* IN:  # of transmitted messages */
```

Description:

The function copies the CAN messages from the application buffer referenced by *cmsg* into the handle's Tx FIFO. The number of messages has to be stored in *len* as multiple of `CMSG` objects before calling ***canSend()***. The function returns immediately and the transmission of the CAN messages is performed asynchronously by the CAN device driver. Upon return, *len* contains the number of `CMSG` objects copied into the Tx FIFO. The number of messages actually copied into the handle's Tx FIFO depends on the Tx FIFO size defined with ***canOpen()*** and the current capacity of this FIFO which depends on the number of pending messages from earlier transmission requests.
The transmission of CAN messages is not affected by the configured operation mode (*FIFO-* or *Object-Mode)*.

Arguments:

handle

[in] CAN handle.

cmsg

[in] Pointer to the application buffer as an array of initialized `CMSG` objects which contain the CAN messages that should be transmitted.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG` objects in the buffer referenced by *cmsg*. Upon return, the driver has stored the number of messages copied into the handle's Tx FIFO into this parameter.

Return Values:

Upon success, NTCAN_SUCCESS is returned or one of the error codes described in chapter 7 in case of a failure.

The return value of NTCAN_SUCCESS does not mean that all CAN messages in the application buffer are copied into the handle's Tx FIFO. This has to be checked by the application on return with the parameter *len*.
As an asynchronous call the return value of NTCAN_SUCCESS does not mean that all CAN messages are transmitted successfully. If a transmission result is required one of the synchronous transmission services have to be used.

Usage:

The call is intended for applications which have to transmit CAN messages asynchronously.

Requirements:

To transmit CAN messages the controller has to be initialized with ***canSetBaudrate().***

See also:

Description of ***canOpen()***.

4.4.2 canSendT

Asynchronous transmission of CAN messages with timestamp.

Syntax:

```
NTCAN_RESULT canSendT(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CMSG_T        *cmsg_t,   /* Ptr to application buffer       */
             int32_t       *len );    /* OUT: # of messages to transmit */
                                      /* IN:  # of transmitted messages */
```

Description:

By default the timestamp information of the CMSG_T structures is not used by NTCAN. The API call is just implemented to allow the use of CMSG_T structures throughout for all CAN I/O operations.
If your CAN hardware supports *Timestamped TX* (FEATURE_TIMESTAMPED_TX, s. ***canIoctl()***) and you open a handle with MODE_TIMESTAMPED_TX (s. ***canOpen()***) you can use the timestamp information in CMSG_T structure to schedule for transmission at a certain time.

The function copies the CAN messages from the application buffer referenced by *cmsg_t* into the handle's Tx FIFO. The number of messages has to be stored in *len* as multiple of CMSG_T objects before calling ***canSendT()***. The function returns immediately and the transmission of the CAN messages is performed asynchronously by the CAN device driver. Upon return, *len* contains the number of CMSG_T objects copied into the Tx FIFO. The number of messages actually copied into the handle's Tx FIFO depends on the Tx FIFO size defined with ***canOpen()*** and the current capacity of this FIFO which depends on the number of pending messages from earlier transmission requests.
The transmission of CAN messages is not affected by the configured operation mode (*FIFO*- or *Object-Mode)*.

Default behaviour:
By default the CAN messages will be transmitted in the given order as soon as possible (exactly as ***canSend()*** would do).

Mode *Timestamped TX*:
In this mode ***canSendT()*** can be utilized in various ways:

- Normal transmission
  Setting the timestamps in the given CMSG_T structures to zero will deactivate the *Timestamped TX* feature for the given frames. The frames will be transmitted through the normal TX-FIFO.
- Scheduled transmission
  Set the timestamps in the given CMSG_T structures and the CAN messages will be transmitted as soon as the timestamp value is reached (assuming an idle CAN bus).
- Prioritized transmission
  If you set a timestamp in the past (e.g. a value of one, but past value different from zero will do), the frames will be transmitted as soon as possible and before CAN frames which have been posted by any other transmission call (***canSend()*** and ***canWrite()***).

You can use ***canIoctl()*** to further configure this mode. For example with certain CAN hardware a “per frame timeout” can be set for CAN messages, which are transmitted by this call.

Arguments:

handle

[in] CAN handle.

cmsg_t

[in] Pointer to the application buffer as an array of initialized `CMSG_T` objects which contain the CAN messages that should be transmitted.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_T` objects in the buffer referenced by *cmsg_t*. Upon return, the driver has stored the number of messages copied into the handle's Tx FIFO into this parameter.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

The return value of `NTCAN_SUCCESS` does not mean that all CAN messages in the application buffer are copied into the handle's Tx FIFO. This has to be checked by the application on return with the parameter *len*.
As an asynchronous call the return value of `NTCAN_SUCCESS` does not mean that all CAN messages are transmitted successfully. If a transmission result is required one of the synchronous transmission services have to be used.

Usage:

The call is intended for applications which have to transmit CAN messages with timestamps asynchronously.

Requirements:

To transmit CAN messages the controller has to be initialized with ***canSetBaudrate().***

See also:

Description of ***canOpen()***.

4.4.3 canSendX

Asynchronous transmission of CAN CC / CAN FD messages with timestamp.

Syntax:

```
NTCAN_RESULT canSendX(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CMSG_X        *cmsg_x,   /* Ptr to application buffer       */
             int32_t       *len );    /* OUT: # of messages to transmit */
                                      /* IN:  # of transmitted messages */
```

Description:

By default the timestamp information of the CMSG_X structures is not used by NTCAN. The API call is just implemented to allow the use of CMSG_X structures throughout for all CAN I/O operations.
If your CAN hardware supports *Timestamped TX* (FEATURE_TIMESTAMPED_TX, s. ***canIoctl()***) and you open a handle with MODE_TIMESTAMPED_TX (s. ***canOpen()***) you can use the timestamp information in CMSG_X structure to schedule for transmission at a certain time.

The function copies the CAN messages from the application buffer referenced by *cmsg_x* into the handle's Tx FIFO. The number of messages has to be stored in *len* as multiple of CMSG_X objects before calling ***canSendX()***. The function returns immediately and the transmission of the CAN messages is performed asynchronously by the CAN device driver. Upon return, *len* contains the number of CMSG_X objects copied into the Tx FIFO. The number of messages actually copied into the handle's Tx FIFO depends on the Tx FIFO size defined with ***canOpen()*** and the current capacity of this FIFO which depends on the number of pending messages from earlier transmission requests.

If the NTCAN_FD bit is set in the parameter *len* of the structure CMSG_X but the handle was opened without NTCAN_MODE_FD (see chapter 4.1.1) the NTCAN_FD bit will be implicitly reset which turns the message into a CAN CC frame limited to 8 byte of data.

If the NTCAN_FD bit is set in the parameter *len* of the structure CMSG_X but no data bit rate is configured the NTCAN_NO_BRS bit will be implicitly set which transmits the frame without a bit rate switch for the data phase.

The transmission of CAN messages is not affected by the configured operation mode (*FIFO-* or *Object-Mode)*.

Default behaviour:
By default the CAN messages will be transmitted in the given order as soon as possible (exactly as ***canSend()*** would do).

Mode *Timestamped TX*:
In this mode ***canSendX()*** can be utilized in various ways:

- Normal transmission
  Setting the timestamps in the given CMSG_X structures to zero will deactivate the *Timestamped TX* feature for the given frames. The frames will be transmitted through the normal TX-FIFO.

- Scheduled transmission
  Set the timestamps in the given `CMSG_X` structures and the CAN messages will be transmitted as soon as the timestamp value is reached (assuming an idle CAN bus).
- Prioritized transmission
  If you set a timestamp in the past (e.g. a value of one, but past value different from zero will do), the frames will be transmitted as soon as possible and before CAN frames which have been posted by any other transmission call (***canSend()*** and ***canWrite()***).

You can use ***canIoctl()*** to further configure this mode. For example with certain CAN hardware a “per frame timeout” can be set for CAN messages, which are transmitted by this call.

Arguments:

handle

[in] CAN handle.

cmsg_x

[in] Pointer to the application buffer as an array of initialized `CMSG_X` objects which contain the CAN messages that should be transmitted.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_X` objects in the buffer referenced by *cmsg_x*. Upon return, the driver has stored the number of messages copied into the handle's Tx FIFO into this parameter.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

The return value of `NTCAN_SUCCESS` does not mean that all CAN messages in the application buffer are copied into the handle's Tx FIFO. This has to be checked by the application on return with the parameter *len*.
As an asynchronous call the return value of `NTCAN_SUCCESS` does not mean that all CAN messages are transmitted successfully. If a transmission result is required one of the synchronous transmission services have to be used.

Usage:

The call is intended for applications which have to transmit CAN / CAN FD messages with timestamps asynchronously.

Requirements:

To transmit CAN / CAN FD messages the controller has to be initialized with ***canSetBaudrate().***

See also:

Description of ***canOpen()***.

4.4.4 canWrite

Synchronous transmission of CAN messages.

Syntax:

```
NTCAN_RESULT canWrite(
             NTCAN_HANDLE  handle,    /* Handle                              */
             CMSG          *cmsg,     /* Ptr to application buffer           */
             int32_t       *len,      /* OUT: # of messages to transmit      */
                                      /* IN:  # of transmitted messages      */
             OVERLAPPED    *ovrlppd); /* NULL or overlapped-structure        */
```

Description:

The function copies the CAN messages from the application buffer referenced by *cmsg* into the handle's Tx FIFO. The number of messages has to be stored in *len* as multiple of `CMSG` objects before calling ***canWrite()***. The function blocks until

- All CAN messages are transmitted.
- An I/O error has occurred.
- The configured transmit timeout is exceeded.
- The I/O operation is aborted with ***canIoctl()*** and the command `NTCAN_IOCTL_ABORT_TX`.
- The I/O operation is aborted by the CAN controller if the DAR mode is enabled (globally or in the message) and the first and only transmission attempt failed because of communication errors or a lost arbitration procedure.
- The handle is closed with ***canClose()***.
- Change of system or device state like e.g. a power state change.

Upon return, *len* contains the number of `CMSG` objects transmitted successfully.

Win32/64

On Windows the CAN driver supports the asynchronous (overlapped) I/O extension which was introduced with Windows NT.
If the flag `NTCAN_MODE_OVERLAPPED` is **not** set in ***canOpen()*** for this handle the driver performs a synchronous I/O operation which is described above and the parameter *ovrlppd should* be set to 0.
If the flag `NTCAN_MODE_OVERLAPPED` is set in ***canOpen()*** for this handle the driver performs an asynchronous I/O operation and the driver returns immediately. The status of the I/O operation has to be retrieved with ***canGetOverlappedResult()***.

Arguments:

handle

[in] CAN handle.

cmsg

[out] Pointer to the application buffer as an array of initialized `CMSG` objects which contain the CAN messages that should be transmitted.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG` objects in the buffer referenced by *cmsg*. Upon return, the driver has stored the number of messages transmitted successfully into this parameter.

ovrlppd

[in] This parameter is used to support Windows asynchronous I/O operations and is ignored by all other supported operating systems. For Windows the pointer has to be set to a valid and unique `OVERLAPPED` structure if the handle is opened with the flag `NTCAN_MODE_OVERLAPPED`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure. The behaviour and error return codes in case of bus error situation while the I/O operation is in progress is CAN device and/or operating system dependent due to technical differences.

Usage:

The call is intended for applications which have to transmit CAN messages synchronously.

Requirements:

To transmit CAN messages the controller has to be initialized with ***canSetBaudrate().***

See also:

Description of ***canOpen()***.

4.4.5 canWriteT

Synchronous transmission of CAN messages (with timestamp).

Syntax:

```
NTCAN_RESULT canWriteT(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CMSG_T        *cmsg_t,   /* Ptr to application buffer       */
             int32_t       *len,      /* OUT: # of messages to transmit  */
                                      /* IN:  # of transmitted messages  */
             OVERLAPPED    *ovrlppd); /* NULL or overlapped-structure    */
```

Description:

By default the timestamp information of the CMSG_T structures is not used by NTCAN. The API call is just implemented to allow the use of CMSG_T structures throughout for all CAN I/O operations.
If your CAN hardware supports *Timestamped TX* (FEATURE_TIMESTAMPED_TX, s. ***canIoctl()***) and you open a handle with MODE_TIMESTAMPED_TX (s. ***canOpen()***) you can use the timestamp information in CMSG_T structure to schedule for transmission at a certain time.

The function copies the CAN messages from the application buffer referenced by *cmsg_t* into the handle's Tx FIFO. The number of messages has to be stored in *len* as multiple of CMSG_T objects before calling ***canWriteT()***. The function blocks until

- All CAN messages are transmitted.
- An I/O error has occurred.
- The configured transmit timeout is exceeded.
- The I/O operation is aborted with ***canIoctl()*** and the command NTCAN_IOCTL_ABORT_TX.
- The I/O operation is aborted by the CAN controller if the DAR mode is enabled (globally or in the message) and the first and only transmission attempt failed because of communication errors or a lost arbitration procedure.
- The handle is closed with ***canClose()***.
- Change of system or device state like e.g. a power state change.

Upon return, *len* contains the number of CMSG_T objects transmitted successfully.

The transmission of CAN messages is not affected by the configured operation mode (*FIFO*- or *Object-Mode)*.

Default behaviour:
By default the CAN messages will be transmitted in the given order as soon as possible (exactly as ***canWrite()*** would do).

Mode *Timestamped TX*:
In this mode ***canWriteT()*** can be utilized in various ways:

- Normal transmission
  Setting the timestamps in the given CMSG_T structures to zero will deactivate the *Timestamped TX* feature for the given frames. The frames will be transmitted through the normal TX-FIFO.
- Scheduled transmission
  Set the timestamp in the given CMSG_T structures and the CAN messages will be transmitted as soon as the timestamp value is reached (assuming an idle CAN bus).

- Prioritized transmission
  If you set a timestamp in the past (e.g. a value of one, but past value different from zero will do), the frames will be transmitted as soon as possible and before CAN frames which have been posted by any other transmission call (***canSend()*** and ***canWrite()***).

You can use ***canIoctl()*** to further configure this mode. For example with certain CAN hardware a “per frame timeout” can be set for CAN messages, which are transmitted by this call.


Win32/64

On Windows the CAN driver supports the asynchronous (overlapped) I/O extension which was introduced with Windows NT.
If the flag `NTCAN_MODE_OVERLAPPED` is **not** set in ***canOpen()*** for this handle the driver performs a synchronous I/O operation which is described above and the parameter *ovrlppd should* be set to 0.
If the flag `NTCAN_MODE_OVERLAPPED` is set in ***canOpen()*** for this handle the driver performs an asynchronous I/O operation and the driver returns immediately. The status of the I/O operation has to be retrieved with ***canGetOverlappedResultT()***.

Using ***canWriteT()*** with *Timestamped TX* mode will behave exactly as one should expect. It will return only after the last frame of the job has been transmitted (depending on your chosen points of transmission this might be well in the future...).

Arguments:

handle
[in] CAN handle.

cmsg
[out] Pointer to the application buffer as an array of initialized `CMSG_T` objects which contain the CAN messages that should be transmitted.

len
[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_T` objects in the buffer referenced by *cmsg_t*. Upon return, the driver has stored the number of messages transmitted successfully into this parameter.

canWriteT() will still return the number of successfully transmitted frames, but as the order of frames depends on the given timestamps, the returned value can't be mapped directly to your given CMSG_T array, if the messages were not in chronological order.

ovrlppd
[in] This parameter is used to support Windows asynchronous I/O operations and is ignored by all other supported operating systems. For Windows the pointer has to be set to a valid and unique `OVERLAPPED` structure if the handle is opened with the flag `NTCAN_MODE_OVERLAPPED`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure. The behaviour and error return codes in case of bus error situation while the I/O operation is in progress is CAN device and/or operating system dependent due to technical differences.

Usage:

The call is intended for applications which have to transmit CAN messages synchronously.

Requirements:
To transmit CAN messages the controller has to be initialized with ***canSetBaudrate().***

See also:
Description of ***canOpen()***.

4.4.6 canWriteX

Synchronous transmission of CAN / CAN FD messages (with timestamp).

Syntax:

```
NTCAN_RESULT canWriteX(
             NTCAN_HANDLE  handle,    /* Handle                              */
             CMSG_X        *cmsg_x,   /* Ptr to application buffer           */
             int32_t       *len,      /* OUT: # of messages to transmit      */
                                      /* IN:  # of transmitted messages      */
             OVERLAPPED    *ovrlppd); /* NULL or overlapped-structure        */
```

Description:

By default the timestamp information of the CMSG_X structures is not used by NTCAN. The API call is just implemented to allow the use of CMSG_X structures throughout for all CAN I/O operations.
If your CAN hardware supports *Timestamped TX* (FEATURE_TIMESTAMPED_TX, s. ***canIoctl()***) and you open a handle with MODE_TIMESTAMPED_TX (s. ***canOpen()***) you can use the timestamp information in CMSG_T structure to schedule for transmission at a certain time.

The function copies the CAN / CAN FD messages from the application buffer referenced by *cmsg_x* into the handle's Tx FIFO. The number of messages has to be stored in *len* as multiple of CMSG_X objects before calling ***canWriteX()***. The function blocks until

- All CAN / CAN FD messages are transmitted.
- An I/O error has occurred.
- The configured transmit timeout is exceeded.
- The I/O operation is aborted with ***canIoctl()*** and the command NTCAN_IOCTL_ABORT_TX.
- The I/O operation is aborted by the CAN controller if the DAR mode is enabled (globally or in the message) and the first and only transmission attempt failed because of communication errors or a lost arbitration procedure.
- The handle is closed with ***canClose()***.
- Change of system or device state like e.g. a power state change.

Upon return, *len* contains the number of CMSG_X objects transmitted successfully.

If the NTCAN_FD bit is set in the parameter *len* of the structure CMSG_X but the handle was opened without NTCAN_MODE_FD (see chapter 4.1.1) the NTCAN_FD bit will be implicitly reset which turns the message into a CAN CC frame limited to 8 byte of data.

If the NTCAN_FD bit is set in the parameter *len* of the structure CMSG_X but no data bit rate is configured the NTCAN_NO_BRS bit will be implicitly set which transmits the frame without a bit rate switch for the data phase.

The transmission of CAN / CAN FD messages is not affected by the configured operation mode (*FIFO-* or *Object-Mode)*.

Default behaviour:
By default the CAN / CAN FD messages will be transmitted in the given order as soon as possible (exactly as ***canWrite()*** would do).

Mode *Timestamped TX*:
In this mode ***canWriteX()*** can be utilized in various ways:

- Normal transmission
  Setting the timestamps in the given `CMSG_X` structures to zero will deactivate the *Timestamped TX* feature for the given frames. The frames will be transmitted through the normal TX-FIFO.
- Scheduled transmission
  Set the timestamp in the given `CMSG_X` structures and the CAN / CAN FD messages will be transmitted as soon as the timestamp value is reached (assuming an idle CAN bus).
- Prioritized transmission
  If you set a timestamp in the past (e.g. a value of one, but past value different from zero will do), the frames will be transmitted as soon as possible and before CAN / CAN FD frames which have been posted by any other transmission call).

You can use ***canIoctl()*** to further configure this mode. For example with certain CAN hardware a “per frame timeout” can be set for CAN / CAN FD messages, which are transmitted by this call.


Win32/64

On Windows the CAN driver supports the asynchronous (overlapped) I/O extension which was introduced with Windows NT.
If the flag `NTCAN_MODE_OVERLAPPED` is **not** set in ***canOpen()*** for this handle the driver performs a synchronous I/O operation which is described above and the parameter *ovrlppd should* be set to 0.
If the flag `NTCAN_MODE_OVERLAPPED` is set in ***canOpen()*** for this handle the driver performs an asynchronous I/O operation and the driver returns immediately. The status of the I/O operation has to be retrieved with ***canGetOverlappedResultX()***.

Using ***canWriteX()*** with *Timestamped TX* mode will behave exactly as one should expect. It will return only after the last frame of the job has been transmitted (depending on your chosen points of transmission this might be well in the future...).

Arguments:

handle
[in] CAN handle.

cmsg
[out] Pointer to the application buffer as an array of initialized `CMSG_X` objects which contain the CAN / CAN FD messages that should be transmitted.

len

[in/out] Pointer to a memory location which has to be initialized before the call to the number of `CMSG_X` objects in the buffer referenced by *cmsg_x*. Upon return, the driver has stored the number of messages transmitted successfully into this parameter.

canWriteX() will still return the number of successfully transmitted frames, but as the order of frames depends on the given timestamps, the returned value can't be mapped directly to your given CMSG_X array, if the messages were not in chronological order.

ovrlppd

[in] This parameter is used to support Windows asynchronous I/O operations and is ignored by all other supported operating systems. For Windows the pointer has to be set to a valid and unique `OVERLAPPED` structure if the handle is opened with the flag `NTCAN_MODE_OVERLAPPED`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure. The behaviour and error return codes in case of bus error situation while the I/O operation is in progress is CAN device and/or operating system dependent due to technical differences.

Usage:

The call is intended for applications which have to transmit CAN / CAN FD messages synchronously.

Requirements:

To transmit CAN messages the controller has to be initialized with ***canSetBaudrate().***

See also:

Description of ***canOpen()***.

4.5 Miscellaneous functions

This section covers the miscellaneous functions to retrieve information about the CAN runtime environment and the functions which ease writing portable and compact code.

4.5.1 canStatus

Returns status information about the CAN hardware and software environment.

Syntax:

```
NTCAN_RESULT canStatus(
             NTCAN_HANDLE  handle,    /* Handle                          */
             CAN_IF_STATUS *cstat);   /* Ptr to status structure         */
```

Description:

The function returns status information about the CAN hardware (board type, hardware revision, firmware revision and CAN controller type), the software runtime environment (revision of device driver and NTCAN library) and the supported capabilities (features) of the CAN driver and/or hardware.

Arguments:

handle

[in] CAN handle.

cstat

[in] Pointer to the application buffer where the driver stores the retrieved status information.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

Return details about the hardware and software environment to the application which may by used to adapt the implementation dynamically.

Requirements:

N/A.

See also:

Description of `CAN_IF_STATUS`.

4.5.2 canGetOverlappedResult

Retrieves the results of an asynchronous (overlapped) operation on the specified CAN handle.

Syntax:

```
EXPORT NTCAN_RESULT CALLTYPE canGetOverlappedResult(
      NTCAN_HANDLE  handle,       /* Handle                          */
    __OVERLAPPED    *ovrlppd,     /* OUT: Win32 overlapped structure  */
                                 /* IN:  N/A                        */
      int32_t       *len,         /* OUT: N/A                         */
                                  /* IN:  # of available CMSG-Buffer  */
      BOOL          bWait );      /* FALSE =>do not wait, else wait   */
```

Description:


Win32/64

This function is required on Windows OS to support the use of asynchronous (also called overlapped) I/O operations. The call is just a wrapper for the ***GetOverlappedResult()*** function of Windows to return the number of CAN messages (without timestamps) instead of the number of transferred bytes.
On all other platforms supported by NTCAN the call is available but returns immediately with an error.

The results reported by the ***canGetOvelappedResult()*** function are those of the specified handle's last overlapped operation to which the specified `OVERLAPPED` structure was provided, and for which the operation's results were pending. A pending operation is indicated when ***canRead()*** or ***canWrite()*** returns `NTCAN_IO_PENDING`. When an I/O operation is pending, the function that started the operation resets the *hEvent* member (which should be a manual-reset event object) of the `OVERLAPPED` structure to the non-signalled state. Then when the pending operation has been completed, the system sets the event object to the signalled state.
If the *bWait* parameter is TRUE, ***canGetOvelappedResult()*** determines whether the pending operation has been completed by waiting for the event object to be in the signalled state.

Arguments:

handle

[in] CAN handle. This is the same handle that was specified when the overlapped operation was started by a call to ***canRead()*** or ***canWrite()*.**

ovrlppd

[in/out] A pointer to an `OVERLAPPED` structure structure that was specified when the overlapped operation was started.

len

[in/out] A pointer to a variable that receives the number of CAN messages that were actually transferred by the read or write operation.

bWait

[in] If this parameter is TRUE, the function does not return until the operation has been completed. If this parameter is FALSE and the operation is still pending, the function returns `NTCAN_IO_INCOMPLETE`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

The call is required for applications which transmit CAN messages asynchronously.

Requirements:

The flag `NTCAN_MODE_OVERLAPPED` has to be set in ***canOpen()*** for this handle***.***

See also:

Description of ***canRead()*** and ***canWrite().***

4.5.3 canGetOverlappedResultT

Retrieves the results of an asynchronous (overlapped) operation on the specified CAN handle.

Syntax:

```
EXPORT NTCAN_RESULT CALLTYPE canGetOverlappedResultT(
      NTCAN_HANDLE  handle,       /* Handle                              */
      OVERLAPPED    *ovrlppd,     /* OUT: Win32 overlapped structure     */
                                  /* IN:  N/A                            */
      int32_t       *len,         /* OUT: N/A                            */
                                  /* IN:  # of available CMSG_T-Buffer   */
      BOOL          bWait );      /* FALSE => do not wait, else wait     */
```

Description:


Win32/64

This function is required on Windows OS to support the use of asynchronous (also called overlapped) I/O operations. The call is just a wrapper for the ***GetOverlappedResult()*** function of Windows to return the number of CAN messages (with timestamps) instead of the number of transferred bytes.
On all other platforms supported by NTCAN the call is available but returns immediately with an error.

The results reported by the ***canGetOvelappedResultT()*** function are those of the specified handle's last overlapped operation to which the specified `OVERLAPPED` structure was provided, and for which the operation's results were pending. A pending operation is indicated when ***canReadT()*** or ***canWriteT()*** returns `NTCAN_IO_PENDING`. When an I/O operation is pending, the function that started the operation resets the *hEvent* member (which should be a manual-reset event object) of the `OVERLAPPED` structure to the nonsignalled state. Then when the pending operation has been completed, the system sets the event object to the signalled state.
If the *bWait* parameter is TRUE, ***canGetOvelappedResultT()*** determines whether the pending operation has been completed by waiting for the event object to be in the signalled state.

Arguments:

handle

[in] CAN handle. This is the same handle that was specified when the overlapped operation was started by a call to ***canReadT()*** or ***canWriteT()*.**

ovrlppd

[in/out] A pointer to an `OVERLAPPED` structure structure that was specified when the overlapped operation was started.

len

[in/out] A pointer to a variable that receives the number of CAN messages that were actually transferred by the read or write operation.

bWait

[in] If this parameter is TRUE, the function does not return until the operation has been completed. If this parameter is FALSE and the operation is still pending, the function returns `NTCAN_IO_INCOMPLETE`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

The call is required for applications which transmit CAN messages asynchronously.

Requirements:

The flag `NTCAN_MODE_OVERLAPPED` has to be set in ***canOpen()*** for this handle***.***

See also:

Description of ***canReadT()*** or ***canWriteT().***

4.5.4 canGetOverlappedResultX

Retrieves the results of an asynchronous (overlapped) operation on the specified CAN handle.

Syntax:

```
EXPORT NTCAN_RESULT CALLTYPE canGetOverlappedResultX(
      NTCAN_HANDLE  handle,       /* Handle                              */
      OVERLAPPED    *ovrlppd,     /* OUT: Win32 overlapped structure     */
                                  /* IN:  N/A                            */
      int32_t       *len,         /* OUT: N/A                            */
                                  /* IN:  # of available CMSG_X-Buffer   */
      BOOL          bWait );      /* FALSE => do not wait, else wait     */
```

Description:


Win32/64

This function is required on Windows OS to support the use of asynchronous (also called overlapped) I/O operations. The call is just a wrapper for the ***GetOverlappedResult()*** function of Windows to return the number of `CMSG_X` instead of the number of transferred bytes.
On all other platforms supported by NTCAN the call is available but returns immediately with an error.

The results reported by the ***canGetOvelappedResultX()*** function are those of the specified handle's last overlapped operation to which the specified `OVERLAPPED` structure was provided, and for which the operation's results were pending. A pending operation is indicated when ***canReadT()*** or ***canWriteT()*** returns `NTCAN_IO_PENDING`. When an I/O operation is pending, the function that started the operation resets the *hEvent* member (which should be a manual-reset event object) of the `OVERLAPPED` structure to the nonsignalled state. Then when the pending operation has been completed, the system sets the event object to the signalled state.
If the *bWait* parameter is TRUE, ***canGetOvelappedResultX()*** determines whether the pending operation has been completed by waiting for the event object to be in the signalled state.

Arguments:

handle

[in] CAN handle. This is the same handle that was specified when the overlapped operation was started by a call to ***canReadT()*** or ***canWriteT()*.**

ovrlppd

[in/out] A pointer to an `OVERLAPPED` structure structure that was specified when the overlapped operation was started.

len

[in/out] A pointer to a variable that receives the number of `CMSG_X` messages that were actually transferred by the read or write operation.

bWait

[in] If this parameter is TRUE, the function does not return until the operation has been completed. If this parameter is FALSE and the operation is still pending, the function returns `NTCAN_IO_INCOMPLETE`.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

The call is required for applications which transmit CAN / CAN FD messages asynchronously.

Requirements:

The flag `NTCAN_MODE_OVERLAPPED` has to be set in ***canOpen()*** for this handle***.***

See also:

Description of ***canReadT()*** or ***canWriteT().***

4.5.5 canFormatError

Generate a platform independent textual description of an NTCAN error or status code.

Syntax:

```
NTCAN_RESULT canFormatError(
               NTCAN_RESULT  error,     /* Error code                    */
               uint32_t      type,      /* Error message type            */
               char          *pBuf,     /* Pointer to destination buffer */
               uint32_t      bufsize);  /* Size of the buffer above      */
```

Description:

The function copies a textual description of an error or status code in the format defined by the parameter *type* for the error code *error* into the application output buffer referenced by *pBuf* which size is defined with the parameter *bufsize*. In case of an unknown error the numerical value of the error code in a hexadecimal representation is appended to the error text.

Arguments:

error

[in] Numerical error code returned by an NTCAN API function.

type

[in] This parameter specifies the format of the returned string. Supported values are:

`NTCAN_ERROR_FORMAT_LONG` – Return a textual (English) error description.
`NTCAN_ERROR_FORMAT_SHORT` – Return error constant as text.

pBuf

[out] A pointer to a buffer that receives the null-terminated error message.

bufsize

[in] This parameter specifies the size of the output buffer *pBuf* in bytes.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

An application should use this function to implement a platform independent error signalling which is capable to indicate error messages for error codes introduced with updated versions of the NTCAN library without the need to modify the application.

Example:

```
char szErrorText[60];
NTCAN_RESULT status = NTCAN_RX_TIMEOUT;  /* Set to Rx timeout error */

/* Format short error message */
canFormatError(status, NTCAN_ERROR_FORMAT_SHORT, szErrorText,
               sizeof(szErrorText));
printf("%s -", szErrorText);

/* Format long error message */
canFormatError(status, NTCAN_ERROR_FORMAT_LONG, szErrorText,
               sizeof(szErrorText));
printf("%s\n", szErrorText);

/* Expected console output:
 *     NTCAN_RX_TIMEOUT - Receive operation timed out
 */
```

Requirements:

N/A.

See also:

Chapter 7 for the list of implemented status and error return codes.

4.5.6 canFormatEvent

Generate a platform independent textual interpretation of an NTCAN event.

Syntax:

```
NTCAN_RESULT canFormatEvent(
  EVMSG                     *event,    /* Event message                */
  NTCAN_FORMATEVENT_PARAMS  *para,     /* Parameters                   */
  char                      *pBuf,     /* Pointer to destination buffer */
  uint32_t                  bufsize);  /* Size of the buffer above     */
```

Description:

The function copies a textual interpretation of an NTCAN event in the format defined by the parameter *para* for the event *event* into the application output buffer referenced by *pBuf* which size is defined with the parameter *bufsize*..

Arguments:

event

[in] Reference to an NTCAN event.

para

[in/out] Reference to an initialized `NTCAN_FORMATEVENT_PARAMS` structure which has to be at least initialized to 0 (see **Remarks** below).

pBuf

[out] A pointer to a buffer that receives the null-terminated event message.

bufsize

[in] This parameter specifies the size of the output buffer *pBuf* in bytes.

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Remark:

As the `EVMSG` does not contain the net number and can therefore not be linked to the CAN device which indicated the event to the application ***canFormatEvent()*** requires additional meta information for some event types. These additional data has to be passed by the application with the `NTCAN_FORMATEVENT_PARAMS` structure. For some event types the call even returns with meta information stored into this structure with the intention that the application keeps them persistent until the next call for this event type. The table below gives an overview which event types require meta information and/or require persistence to work properly.

Event	Input data	Persist	Remark
`NTCAN_EV_CAN_ERROR`	N/A	No	N/A
`NTCAN_EV_BAUD_CHANGE`	N/A	No	N/A
`NTCAN_EV_CAN_ERROR_EXT`	*ctrl_type*	No	Only supported for SJA1000 and esdACC CAN controller.
`NTCAN_EV_BUSLOAD`	*timestamp_freq* *num_baudrate*	Yes	Returns `NTCAN_INVALID_PARAMETER` for the first call.

The currently unused definition of `NTCAN_FORMATEVENT_SHORT` can be used at compile time to check if the ***canFormatEvent()*** function is available.

Usage:

An application should use this function to implement a platform independent CAN event handling which is capable to show textual interpretation for CAN events introduced with updated versions of the NTCAN library or without the need to modify the application.

Example:

```
char szMsg[60];
EVMSG evmsg;
NTCAN_FORMATEVENT_PARAMS para;

   /* Create a CAN event (usually returned with e.g. canRead()) */
evmsg.evid       = NTCAN_EV_CAN_ERROR;
evmsg.len        = 4;
evmsg.can_status = NTCAN_BUSSTATE_BUSOFF;

   /* Initialize parameter structure */
memset(para, 0, sizeof(para);

   /* Create a textual event description */
if(NTCAN_SUCCESS == canFormatEvent(&evmsg, &para, szMsg, sizeof(szMsg)) {
   printf("%s\n", szMsg);
}

/* Expected console output: “Controller bus-off” */
```

Requirements:

N/A.

See also:

Chapter 7 for the list of implemented status and error return codes.

4.5.7 canFormatFrame

Generate the bitstream for error injection based on a CAN message.

Syntax:

```
NTCAN_RESULT canFormatFrame(CMSG *msg,           /* CAN message            */
                            CMSG_FRAME *frame,  /* CAN Frame + Information*/
                            uint32_t    eccExt);/* ECC Errors + Features  */
```

Description:

This function simplifies generating the CAN bitstream used by the error injection units based on an initialized CAN message (`CMSG`) by calculating the CRC and the position of stuff-bits. Additionally to the bitstream are also the individual positions of the frame parts stored in the frame structure. Via the eccExt errors can be incorporated into the bitstream, such as stuff-errors or form-erros in a certain position in the frame.

Arguments:

msg

[in] Reference to the completely initialized `CMSG` structure.

frame

[out] Pointer to the memory location to store the resulting bitstream as a `CMSG_FRAME` structure.

eccExt

[in] Specifies the position of an error in the CAN-Frame in the coding of the ECC register form NXP SJA1000

Return Values:

Upon success, `NTCAN_SUCCESS` is returned or one of the error codes described in chapter 7 in case of a failure.

Usage:

Fill CMSG with valid data an call ***canFormatFrame()***.

Example:

```
CMSG canMsg;
CMSG_FRAME canFrame;

canMsg.id      = 42;
canMsg.len     =  2;
canMsg.data[0] = 12;
canMsg.data[1] = 34;

memset(&canFrame, 0, sizeof(canFrame));
ret = canFormatFrame(&canMsg, &canFrame, 0);
if (ret != NTCAN_SUCCESS) {
    printf("canFormatFrame returned: %d\n", ret);
    return 0;
}
printf("Generated CAN bitstream:\n Length: %d\n", canFrame.length);
printf("\0x%x 0x%x 0x%x 0x%x 0x%x\n",
        canFrame.can_frame.l[0], canFrame.can_frame.l[1],
        canFrame.can_frame.l[2], canFrame.can_frame.l[3],
        canFrame.can_frame.l[4]);
```

Requirements:

N/A.

See also:

N/A.

5. Macros

This chapter describes the macros in the header **<ntcan.h>** which simplify writing applications and make the code more readable.

5.1 NTCAN_DATASIZE_TO_DLC

Convert the payload size of a **CAN FD** message into a DLC value.

Syntax:

```
#define NTCAN_DATASIZE_TO_DLC(dataSize)
```

Description:

This macro returns the DLC value of a CAN FD message for a given payload size in bytes.

Remark:

Compliant with /2/ payloads with more than 8 bytes (for CAN FD frames) can only be mapped to certain discrete DLC values. The mapping is performed by this macro rounding up to the next DLC value which payload can hold the given number of bytes. The argument is internally limited to 64.

Arguments:

dataSize

Payload in bytes in the range from 0..64.

5.2 NTCAN_DLC

Return the Data Length Code (DLC)

Syntax:

```
#define NTCAN_DLC(len)
```

Description:

This macro returns the DLC code of a CAN message without additional meta information which may be coded in the parameter *len* of the CMSG, CMSG_T or CMSG_X structure.

Arguments:

len

Member *len* of `CMSG`, `CMSG_T` or `CMSG_X`.

5.3 NTCAN_DLC_AND_TYPE

Return the data length code and the message type.

Syntax:

```
#define NTCAN_DLC_AND_TYPE(len)
```

Description:

This macro returns the data length code of a CAN message together with the message type information (RTR message or data message) but without additional optional meta information which may be coded in the parameter *len* of the CMSG or CMSG_T structure.

Arguments:

len

Member *len* of `CMSG` or `CMSG_T`.

Note:

This macro is still available for backward compatibility reasons but deprecated as the specification /2/ removes the RTR bit for CAN FD messages and so the meta information (bit 4) in the parameter *len* to mark a message as RTR is only valid for CAN CC messages and is used with a different meaning for CAN FD messages by the NTCAN API. To check if a received message is a RTR message the CAN FD aware macro `NTCAN_IS_RTR` should be used.

5.4 NTCAN_GET_BOARD_STATUS

Return the hardware status.

Syntax:

```
#define NTCAN_GET_BOARD_STATUS(boardstatus)
```

Description:

This macro returns the CAN controller HW status.

Arguments:

boardstatus

Member *boardstatus* of `CAN_IF_STATUS`.

5.5 NTCAN_GET_CTRL_TYPE

Return the CAN controller type.

Syntax:

```
#define NTCAN_GET_CTRL_TYPE(boardstatus)
```

Description:

This macro returns the CAN controller type according to table 21.

Arguments:

boardstatus

Member *boardstatus* of `CAN_IF_STATUS`.

5.6 NTCAN_GET_TDC_FILTER

Return the configured TDC filter.

Syntax:

```
#define NTCAN_GET_TDC_FILTER(val)
```

Description:

This macro returns the configured TDC filter (see chapter 3.15.2) from the 32 bit value returned with `NTCAN_IOCTL_GET_FD_TDC`.

Remark:

Supported only by CAN FD controllers. A CAN controller without TDC filter support will return 0.

Arguments:

val
Value returned with `NTCAN_IOCTL_GET_FD_TDC`.

5.7 NTCAN_GET_TDC_MODE

This macro returns the configured TDC mode.

Syntax:

```
#define NTCAN_GET_TDC_MODE(val)
```

Description:

This macro returns the configured TDC mode (see chapter 3.15.2) from the 32 bit value returned with `NTCAN_IOCTL_GET_FD_TDC`. Supported modes are:

- **`NTCAN_TDC_MODE_AUTO`**: TDC automatic mode (See chapter 3.15.2.1).
- **`NTCAN_TDC_MODE_MANUAL`**: TDC Manual Mode (See chapter 3.15.2.2).
- **`NTCAN_TDC_MODE_OFF`**: TDC is disabled.

Remark:

Supported only by CAN FD controllers.

Arguments:

val
Value returned with `NTCAN_IOCTL_GET_FD_TDC`.

5.8 NTCAN_GET_TDC_SSPO

Return the current SSP Offset as a multiple of CAN clock cycles.

Syntax:

```
#define NTCAN_GET_TDC_SSPO(val)
```

Description:

This macro returns the current SSP Offset (SSPO) (see chapter 3.15.2) from the 32 bit value returned with `NTCAN_IOCTL_GET_FD_TDC`.

Remark:

Supported only by CAN FD controllers.

Arguments:

val
Value returned with `NTCAN_IOCTL_GET_FD_TDC`.

5.9 NTCAN_GET_TDC_SSPS

Return the configured TDC shift as a multiple of CAN clock cycles.

Syntax:

```
#define NTCAN_GET_TDC_SSPS(val)
```

Description:

This macro returns the configured TDC shift value (see chapter 3.15.2) from the 32 bit value returned with `NTCAN_IOCTL_GET_FD_TDC`.

Remark:

Supported only by CAN FD controllers. For backward compatibility this macro is also available with the legacy name `NTCAN_GET_TDC_OFFSET`.

Arguments:

val
Value returned with `NTCAN_IOCTL_GET_FD_TDC`.

5.10 NTCAN_GET_TDC_TD

Get the current measured TD value as multiple of CAN clock cycles.

Syntax:

```
#define NTCAN_GET_TDC_TD(val)
```

Description:

This macro returns the current measured TD value (see chapter 3.15.2) from the 32 bit value returned with `NTCAN_IOCTL_GET_FD_TDC`.

Remark:

Supported only by CAN FD controllers. For backward compatibility this macro is also available with the legacy name `NTCAN_GET_TDC_VALUE`.

Arguments:

val
Value returned with `NTCAN_IOCTL_GET_FD_TDC`.

5.11 NTCAN_IS_FD

Check for FD message.

Syntax:

```
#define NTCAN_IS_FD(len)
```

Description:
This macro returns a value different from 0 if the message is a CAN FD frame and 0 in case of a CAN CC frame.

Arguments:
len
Member *len* of `CMSG_X`.

5.12 NTCAN_IS_FD_WITHOUT_BRS

Check if a FD frame is transmitted without bit rate change during the data phase.

Syntax:

```
#define NTCAN_IS_FD_WITHOUT_BRS(len)
```

Description:

This macro returns a value different from 0 if the CAN controller which transmitted the received CAN FD message did this without changing the bit rate during the data phase but keeping the configured nominal bit rate.

Remark:

Supported only by CAN FD controllers transmitting CAN FD messages.

Arguments:

len

Member *len* of `CMSG_X`

5.13 NTCAN_IS_RTR

Check for a RTR message.

Syntax:

```
#define NTCAN_IS_RTR(len)
```

Description:

This macro returns a value different from 0 if the message is a RTR frame and 0 in case of a data frame.

Arguments:

len

Member *len* of `CMSG`, `CMSG_T` or `CMSG_X`.

5.14 NTCAN_IS_INTERACTION

Check for an interaction message.

Syntax:

```
#define NTCAN_IS_INTERACTION(len)
```

Description:

This macro returns a value different from 0 if the message is receive via the interaction mechanism and 0 if not.

Remark:

The interaction indication has to be enabled with ***canOpen()*** otherwise the macro will always return 0 because of the disabled indication.

Arguments:

len

Member *len* of `CMSG`, `CMSG_T` or `CMSG_X`.

5.15 NTCAN_LEN_TO_DATASIZE

Convert the DLC of a CAN message into a the payload size in bytes.

Syntax:

```
#define NTCAN_LEN_TO_DATASIZE(len)
```

Description:

This macro returns the length of a CAN message in bytes regarding the CAN message type (Data or RTR) and CAN message format (Classic or FD).

Remark:

Compliant to /2/ for all CAN CC Data frames with DLC values between 9..15 a payload size of 8 bytes is returned and for CAN CC RTR frames a payload of 0 bytes is returned independent of the DLC value.

Arguments:

len

Member *len* of `CMSG`, `CMSG_T` or `CMSG_X`.

5.16 NTCAN_SET_TDC

Set the TDC mode and SSP shift.

Syntax:

```
#define NTCAN_SET_TDC(mode, shift)
```

Description:

This macro returns the 32 bit value to configure the TDC operation mode and SSP shift value (see chapter 3.15.2) which is passed to the driver with `NTCAN_IOCTL_SET_FD_TDC`.

Remark:

Supported only by CAN FD controllers.

Arguments:

mode

TDC mode (see chapter 5.7 for supported values).

shift

Mode specific signed or unsigned SSP shift value.

6. 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. 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	32	32	64
pointer	32	64	64

Table 18: Data type size in bits for different data models

In order to stay cross-platform portable with respect to different CPU architectures and compilers newer versions of the NTCAN-API header **<ntcan.h>** do not use the native standard integer data types of the C language any more. Instead the data types in the header **<stdint.h>** are supported which defines various integer types and related macros with size constraints.

Specifier	Signing	Bytes	Range
`int8_t`	Signed	1	−128...127
`uint8_t`	Unsigned	1	0...255
`int16_t`	Signed	2	−32,768...32767
`uint16_t`	Unsigned	2	0...65535
`int32_t`	Signed	4	−2,147,483,648...2,147,483,647
`uint32_t`	Unsigned	4	0...4,294,967,295
`int64_t`	Signed	8	−9,223,372,036,854,775,808...9,223,372,036,854,775,807
`uint64_t`	Unsigned	8	0...18,446,744,073,709,551,615

Table 19: Simple C99 data types used by NTCAN-API

These data types are part of the ISO/IEC 9899:1999 standard which is also commonly referred to as *C99* standard.

For platforms which do not follow the C99 standard (e.g. Windows) these types are defined in the `<ntcan.h>` header using compiler- and OS-specific knowledge of the native data types.

Caveat: The introduction of the C99 data types has not affected the binary interface of the NTCAN-API on any platform.
Nevertheless there are several C/C++ compiler which issue warning messages if existing applications based on C native data types are re-compiled with a NTCAN header with C99 data types, even if the data type size in bits and the data type signedness have not been changed.

6.1 Simple Data Types

This section describes the simple data types defined by the NTCAN-API in alphabetical order. They all start with the prefix **NTCAN_** with respect to a clean namespace.

6.1.1 NTCAN_HANDLE

The type defines an opaque operating system specific reference to a physical CAN port. This handle is the input or output parameter of most NTCAN-API functions. As an input parameter the handle is validated by the called function. This type should be used in applications instead of the platform specific native type for cross-platform portability.

It is not guaranteed that the returned handle is identical with the OS specific reference to the device and a current NTCAN-API implementation might even change in the future. Usually this handle (Windows) or file descriptor (POSIX compatible OS) is not required for the CAN communication. In rare cases it might be necessary to obtain this reference. In this case an application can call ***canIoctl()*** with `NTCAN_IOCTL_GET_NATIVE_HANDLE` as argument but working with this handle or file descriptor might cause unwanted side effects.

If an application wants to indicate that a handle is invalid the portable definition **NTCAN_NO_HANDLE** should be used for this instead of the native representation of an invalid handle.

6.1.2 NTCAN_RESULT

The type defines the operating system specific data type for the return value of every NTCAN-API function. This type should be used in applications instead of an OS specific native type for cross-platform portability.

6.2 Compound Data Types

This section describes the compound data types defined by the NTCAN-API in alphabetical order.

6.2.1 CAN_FRAME_STREAM

The `CAN_FRAME_STREAM` union is used for Error Injection as part of the data types `CMSG_FRAME` and `NTCAN_EEI_UNIT`. It's a union of 160 bits for a complete CAN frame with stuff bits that can be accessed as 8-Bit, 16-Bit or 32-Bit array.

Syntax:

```
typedef union {
  uint8_t  c[20];
  uint16_t s[10];
  uint32_t l[5];
} CAN_FRAME_STREAM;
```

Members:

c
Access as array of 8-Bit values.

s
Access as array of 16-Bit values..

l
Access as array of 32-Bit values.

6.2.2 CAN_IF_STATUS

The `CAN_IF_STATUS` is returned by ***canStatus()*** with information about the CAN hardware and the runtime environment.

Syntax:

```
typedef struct
{
  uint16_t hardware;       /* Hardware version                          */
  uint16_t firmware;       /* Firmware version (0 for passive hardware) */
  uint16_t driver;         /* Driver version                            */
  uint16_t dll;            /* NTCAN library version                     */
  uint32_t boardstatus;    /* Hardware status, CAN controller type      */
  uint8_t  boardid[14];    /* Board ID string                           */
  uint16_t features;       /* Device/driver capability flags            */
} CAN_IF_STATUS;
```

Members:

hardware

The hardware revision. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

firmware

The firmware version. Returned as 0 on passive CAN interfaces. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

driver

The driver version. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

dll

The NTCAN-API library version. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

boardstatus

This 32-Bit value is divided into three parts:

Bit 31..24	Bit 23..16	Bit 15..0
CAN Controller Type	Reserved	Hardware Status

The *Hardware Status* reflects device specific errors or problems usually detected during hardware initialization. The table below gives an overview about the supported values:

Constant	Hardware Status
`NTCAN_BSTATUS_OK`	No error.
`NTCAN_BSTATUS_NEED_FW_OK`	Driver and FW are incompatible. Update the FW.
`NTCAN_BSTATUS_HW_ERROR`	Hardware error (usually during initialization).

Table 20: CAN Hardware Status

The *CAN Controller Type* returns a value defined in **<ntcan.h>** for the manufacturer and type of the CAN controller according to the table below.

Constant	CAN Controller
NTCAN_CANCTL_SJA1000	NXP SJA1000 / Philips 82C200
NTCAN_CANCTL_I82527	Intel I82527
NTCAN_CANCTL_FUJI	Fujitsu MBxxxxx MCU
NTCAN_CANCTL_LPC	NXP LPC2xxx / LPC17xx MCU
NTCAN_CANCTL_MSCAN	Freescale MCU (MSCAN)
NTCAN_CANCTL_ATSAM	Atmel ARM CPU
NTCAN_CANCTL_ESDACC	**esd electronics** CAN FD IP Core (Advanced CAN Core)
NTCAN_CANCTL_STM32	ST STM32Fxxx MCU (bxCAN)
NTCAN_CANCTL_CC770	Bosch CC770 (Intel 82527 compatible)
NTCAN_CANCTL_SPEAR	ST SPEAr320 (Bosch C_CAN compatible)
NTCAN_CANCTL_FLEXCAN	Freescale I.MX SoC (FlexCAN)
NTCAN_CANCTL_SITARA	TI AM335x (Sitara) SoC (Bosch D_CAN compatible)
NTCAN_CANCTL_MCP2515	Microchip MCP2515
NTCAN_CANCTL_MCAN	Bosch CAN FD IP Core (M_CAN)
NTCAN_CANCTL_CAST	CAST CAN CC IP Core
NTCAN_CANCTL_ESDLIN	**esd electronics** LIN Core
NTCAN_CANCTL_MSAM	Microchip SAM E70/S70/V70/V71

Table 21: CAN Controller Types

The macros NTCAN_GET_CTRL_TYPE and NTCAN_GET_BOARD_STATUS are available to extract the controller type and the hardware status field from the *boardstatus*.

As storing the CAN controller type in the MSB of boardstatus was introduced at a later time it might be possible that a driver returns invalid values. Please contact **esd electronics** if a driver update is available.

boardid

The device description as zero terminated ASCII string.

features

This member is a bit mask with hardware and/or device driver specific capabilities which should be evaluated by an application to check if a certain feature is supported.

Caveat: The number of indicated features is limited to 16 due to the data type in this structure. The table below covers a 32 bit data type so only the LSW is returned with this data type. The complete bitmask of *features* is just returned in the NTCAN_INFO structure. If NTCAN_INFO is not supported all features indicated in the MSW can be assumed by an application to be 0.

Bit	Flag	Meaning
0	NTCAN_FEATURE_FULL_CAN	If the flag is set, the CAN board is based on a FullCAN controller otherwise on a BasicCAN controller.
1	NTCAN_FEATURE_CAN_20B	If the flag is set, CAN messages with 29-bit and 11-bit CAN-IDs (extended and base frame format) can be transmitted and received. Otherwise just 11-bit CAN-IDs (base frame format) are supported.
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 CAN interface runs with a customer specific firmware with support for an autonomous and cyclic transmission of CAN messages (driver revision <= 2.x).
3	NTCAN_FEATURE_TIMESTAMPED_TX	If the flag is set, the driver supports the transmission of CAN messages at a given time with ***canSendT()*** (driver revision >= 3.x).
4	NTCAN_FEATURE_RX_OBJECT_MODE	If the flag is set, the driver supports the *Object Mode* in addition to the FIFO mode.
5	NTCAN_FEATURE_TIMESTAMP	If the flag is set, the CAN hardware/firmware supports time-stamping of received CAN messages (see chapter 3.9).
6	NTCAN_FEATURE_LISTEN_ONLY_MODE	If the flag is set, the CAN hardware supports the Listen-Only mode (see chapter 3.3.2)
7	NTCAN_FEATURE_SMART_DISCONNECT	If the flag is set, the CAN hardware/firmware operates in the *Smart Disconnect* mode (see chapter 3.3.8)
8	NTCAN_FEATURE_LOCAL_ECHO	If the flag is set, the driver supports receiving interaction frames in FIFO mode with the same handle they are sent.
9	NTCAN_FEATURE_SMART_ID_FILTER	If the flag is set, the driver supports the adaptive ID filter for 29-bit CAN-IDs.
10	NTCAN_FEATURE_SCHEDULING	If the flag is set, the driver supports to schedule the transmission of CAN messages periodically or at a given point in time (see chapter 3.12).
11	NTCAN_FEATURE_DIAGNOSTIC	If the flag is set, the driver and the hardware and/or firmware support extended CAN bus diagnostic (see chapter 3.6.2).
12	NTCAN_FEATURE_ERROR_INJECTION	If the flag is set, the driver and the hardware support error injection (see chapter 3.13).
13	NTCAN_FEATURE_IRIGB	If the flag is set, the hardware supports clock synchronization based on the IRIG B standard.

Bit	Flag	Meaning
14	NTCAN_FEATURE_PXI	If the flag is set, the hardware supports using the PXI back plane clock in addition to the internal clock.
15	NTCAN_FEATURE_CAN_FD	If the flag is set, the hardware supports communication according to the CAN FD standard.
16	NTCAN_FEATURE_SELF_TEST	If the flag is set, the hardware supports a self test mode (see chapter 3.3.3).
17	NTCAN_FEATURE_TRIPLE_SAMPLING	If the flag is set, the CAN controller supports a triple sampling mode (see chapter 3.3.4).
18	NTCAN_FEATURE_TX_PAUSE	If the flag is set, the CAN controller supports a Tx pause mode (see chapter 3.3.5).
19	NTCAN_FEATURE_DAR	If the flag is set the CAN controller supports a mode to globally disable the automatic retransmission after a transmission failure (see chapter 3.3.6)
20	NTCAN_FEATURE_DAR_FRAME	If the flag is set the CAN controller supports a mode to disable on a per frame basis the automatic retransmission after a transmission failure (see chapter 3.3.6)
21	NTCAN_FEATURE_PROG_TERM	If the flag is set the hardware supports to enable or disable an on-board bus termination programmatically.
22	NTCAN_FEATURE_GPIO	If the flag is set the hardware supports GPIO ports which are controlled via the NTCAN API.
23..26	N/A	Reserved for future use.
27	NTCAN_FEATURE_LIN	Network has LIN physics.
28..31	N/A	Reserved for future use.

Table 22: NTCAN Feature Flags

Remarks:

The members which contain a version are composed of major version (4 bit), minor version (4 bit) and a revision (8 bit).

Bit 12..15	Bit 8..11	Bit 0..7
Major	Minor	Revision

Example: The version 1.2.3 is represented as 0x1203.

6.2.3 CMSG

The `CMSG` structure contains a CAN message with the CAN identifier (CAN-ID), the number of data bytes (DLC), additional message specific meta data and up to 8 data bytes.

Syntax:

```
typedef struct
{
   int32_t id;        /* CAN-ID (11-/29-bit) or Event-ID          [Tx, Rx]  */
   uint8_t len;       /* Bit 0-3 = Data Length Code               [Tx, Rx]  */
                      /* Bit 4   = RTR (CAN CC)                   [Tx, Rx]  */
                      /*         = No_BRS (CAN FD)                [Tx, Rx]  */
                      /* Bit 5   = No_Data (Object Mode)          [    Rx]  */
                      /*         = Interaction Data (FIFO Mode)               */
                      /* Bit 6   = Reserved                       [    Rx]  */
                      /* Bit 7   = Type(CAN FD / CAN CC)          [Tx, Rx]  */
   uint8_t msg_lost; /* Counter for lost Rx messages             [    Rx]  */
   uint8_t reserved[1]; /* Reserved                                          */
   uint8_t esi;       /* Error State Indicator (CAN FD)           [    Rx]  */
   uint8_t data[8];    /* 8 data bytes                            [Tx, Rx]  */
} CMSG;
```

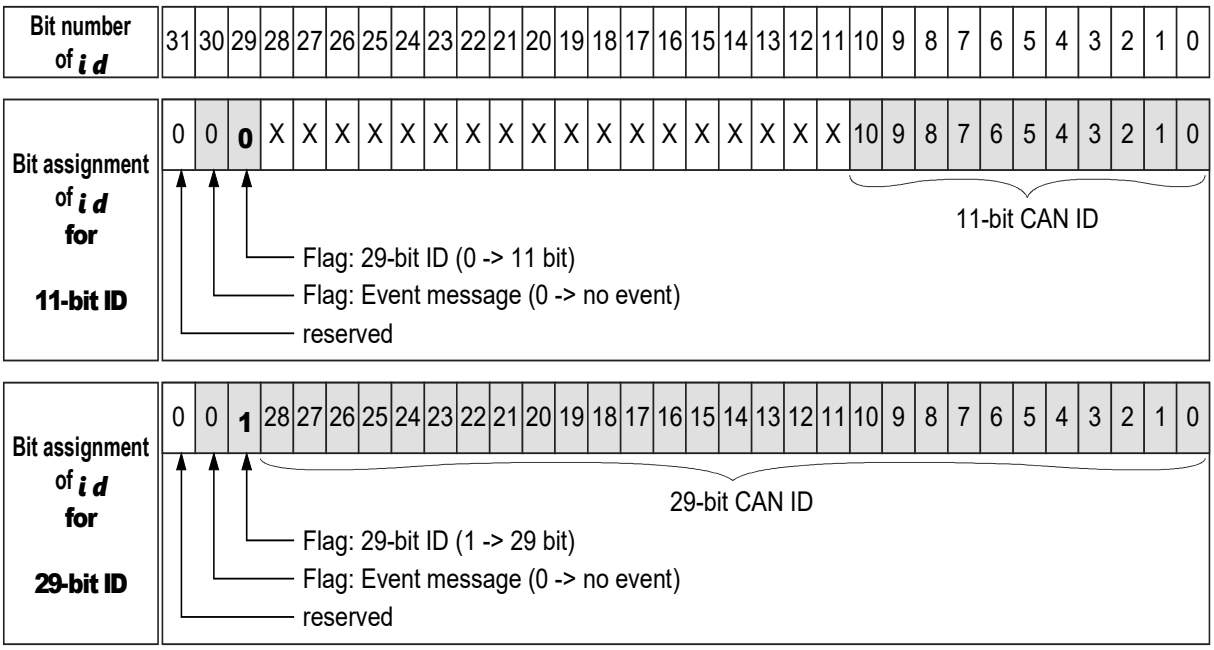
Members:

id

Identifier (CAN-ID) of the CAN message.

Bit 29 of *id* is used to distinguish an 11-bit CAN identifier from a 29-bit CAN identifier. In order to transmit a message with a 29-bit identifier this bit has to be set in addition to the CAN identifier. This can be achieved by bit wise OR the identifier with NTCAN_20B_BASE (defined in **<ntcan.h>**). If a message with a 29-bit identifier is received, this bit is set.

Bit 30 of *id* is used to distinguish an CAN message from a CAN event (refer to chapter 6.2.12 for details).



len

The structure member *len* contains the Data Length Code (DLC) with the number of valid data bytes for transmitted and received CAN messages in the bits 0..3. The bits 4..7 are used to store additional meta information.

Bit	7	6	5	4	3	2	1	0
Description	CAN Message Meta Information				CAN Message DLC			

The Data Length Code (DLC) of a CAN message indicates the number of valid bytes in the *data* array of a received or transmitted `CMSG`. For CAN CC messages /1/ defines a direct mapping between the DLC values 0..8 and the number of data bytes. For all DLC values greater 8 the maximum number of data bytes is limited to 8.

DLC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Max bytes	0	1	2	3	4	5	6	7	8	8	8	8	8	8	8	8

Table 23: Mapping between DLC and payload size for CAN CC

For CAN FD messages /1/ defines a direct mapping between the DLC values 0..8 and the number of data bytes. All DLC values greater 8 are mapped to non consecutive maximum numbers of data bytes according to the table below.

DLC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Max bytes	0	1	2	3	4	5	6	7	8	12	16	20	24	32	48	64

Table 24: Mapping between DLC and payload size for CAN FD

You should use the macro **`NTCAN_DLC`** in your application if you just need to refer to the DLC of the message. The macros **`NTCAN_DATASIZE_TO_DLC`** and **`Error: Reference source not found`** will help to convert between DLC values and payload size.

The payload of a Remote Request Transmission (RTR), defined only for a CAN CC messages, is always 0 independent of the DLC value. The **`Error: Reference source not found`** macro takes care of this.

The meaning of the four meta information bits depend on the message direction (Tx/Rx), the CAN message type (CAN CC / CAN FD) and/or the handle mode (FIFO mode/Object mode) according to the table below:

Dir	Len Bit	Value	
		0	1
Tx	7	Classcical CAN Message	CAN FD Message
	6	Reserved for future use (Set to 0)	Reserved for future use (Set to 0)
	5	Retransmit CAN message in case of communication errors or a lost arbitration procedure (Default behavior of a CAN controller)	No retransmission of the CAN message in case of a communication error or a lost arbitration procedure
	4	CAN CC Data Frame CAN FD Data Frame with Bit Rate Switch	CAN CC Remote Transmission Request (RTR) CAN FD Data Frame without Bit Rate Switch
Rx	7	Classcical CAN Message	CAN FD Message
	6	Don't care (Reserved for internal use)	Don't care (Reserved for internal use)
	5	Handle FIFO Mode: Frame received via physical port Handle Object Mode: Returned Data Valid	Handle FIFO Mode: Frame received via Interaction Handle Object Mode: Returned Data Invalid
	4	CAN CC (Bit 7 = 0) Data Frame CAN FD (Bit 7 = 1) Data Frame with Bit Rate Switch	CAN CC (Bit 7 = 0) Remote Transmission Request (RTR) CAN FD (Bit 7 = 1) Data Frame without Bit Rate Switch

Table 25: Meta Information of the CMSG len.

Bit 7 (Message Type):

The bit 7 of *len* (defined as `NTCAN_FD` in **<ntcan.h>**) is used to mark a message as CAN CC or CAN FD for transmitted as well as received messages. The state of this bit affects the meaning of bit 4 (RTR vs. BRS) as well as mapping between DLC value and size of data bytes. You should use the macro `NTCAN_IS_FD` to distinguish between the message types.

Bit 6 (Reserved):

Bit 6 is reserved for internal use and should be set to 0 for transmitted messages.

Bit 5 (Interaction / Data Valid / DAR Mode):

The meaning of the bit depends on the communication direction (Receive or Transmit)

Receive:

The meaning of the bit depends on the handle type (*Object Mode* or *FIFO Mode)*.

FIFO Mode:
The bit (defined as `NTCAN_INTERACTION` in **<ntcan.h>**) indicates if the message is received via the interaction mechanism, i.e. from another application using the same physical CAN port.
The bit is set if the handle used to receive the message is either explicitly opened with the mode flag `NTCAN_MODE_MARK_INTERACTION` or `NTCAN_MODE_LOCAL_ECHO`. You should use the macro **`NTCAN_IS_INTERACTION`** to check if a frame is received via interaction (local echo) or not.

Value of Bit 5	Mode flag of canOpen()	Function
0 1	`NTCAN_MODE_MARK_INTERACTION`	Message not received via interaction Message received via interaction
0 1	`NTCAN_LOCAL_ECHO`	Message not send on this handle Message send on this handle

Object Mode:
The bit (defined as `NTCAN_NO_DATA` in **<ntcan.h>**) indicates that no data is received for this object yet or the data might be updated while the CAN controller was disconnected from the CAN bus. The bit is only set if the handle used to receive the message is explicitly opened with the mode flag `NTCAN_MODE_OBJECT`.

Transmit:

The bit (defined as `NTCAN_DAR` in **<ntcan.h>**) controls the DAR mode and indicates if the CAN controller should retransmit (default behaviour) the message in case of a bus error or a lost arbitration procedure. If the bit is set, the automatic retransmission is disabled for this message.

The ability of a CAN controller to enable/disable the DAR mode per frame is hardware dependent and can be checked by the application with the feature flag `NTCAN_FEATURE_DAR_FRAME`

Bit 4 (RTR / BRS):

The meaning of the bit depends on the message type (CAN CC / CAN FD).

Remote Transmisiion Request (RTR):

The bit 4 of *len* (defined as `NTCAN_RTR` in **<ntcan.h>**) is used to distinguish a **CAN CC (Bit 7 = 0)** *data frame* from a *remote request (RTR) frame* for received and transmitted messages. In both cases the bits 0..3 of *len* are valid but in case of a RTR frame the bytes of the *data* array are invalid. You should use the macro **`NTCAN_IS_RTR`** to distinguish between the frame types.

The ability of **Full CAN** controller to send/receive arbitrary RTR frames might be limited by the hardware.

Bit Rate Switch (BRS):

The bit 4 of *len* (defined as `NTCAN_NO_BRS` in **<ntcan.h>**) is used to distinguish a **CAN FD (Bit 7 = 1)** transmitted or received frame with a bit rate switch in the data phase from a frame without bit rate switch. You should use the macro `NTCAN_IS_FD_WITHOUT_BRS` to figure out the mode for received messages.

msg_lost

FIFO Mode:
If the receive FIFO of the handle gets overrun by new messages, the oldest messages are overwritten and the *msg_lost* counter is increased so the application can detect this data overrun. A counter different from 0 indicates that the application program processes the CAN data flow slower than the driver provides new data.

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

RX Object Mode (Driver V4.1.x and later):
The counter value is incremented with each update of the CAN message and revolves from 255 to 0. An application which polls the object in regular intervals can use the counter to figure out if and how many times the object was updated between consecutive read operations.

Note:

Caveat: Basically it is possible to transmit and receive CAN FD messages in a `CMSG` structure if the handle was opened with `NTCAN_MODE_FD`. If a CAN FD message with more than 8 bytes is transmitted or received the DLC will represent the real data size but the data itself is obviously limited to 8 bytes by the device driver.

6.2.4 CMSG_T

The CMSG_T structure contains a CAN message with the CAN identifier (CAN-ID), the number of data bytes (DLC), additional message specific meta data, up to 8 data bytes and a 64-Bit timestamp.

Syntax:

```
typedef struct
{
   int32_t id;        /* CAN-ID (11-/29-bit) or Event-ID          [Tx, Rx]  */
   uint8_t len;       /* Bit 0-3 = Data Length Code               [Tx, Rx]  */
                      /* Bit 4   = RTR (CAN CC)                   [Tx, Rx]  */
                      /*         = No_BRS (CAN FD)                [Tx, Rx]  */
                      /* Bit 5   = No_Data (Object Mode)          [    Rx]  */
                      /*         = Interaction Data (FIFO Mode)               */
                      /* Bit 6   = Reserved                       [    Rx]  */
                      /* Bit 7   = Type(CAN FD / CAN CC)          [Tx, Rx]  */
   uint8_t msg_lost; /* Counter for lost Rx messages             [    Rx]  */
   uint8_t reserved[1];/* Reserved                                           */
   uint8_t esi;        /* Error State Indicator (CAN FD)          [    Rx]  */
   uint8_t data[8];    /* 8 data bytes                            [Tx, Rx]  */
   uint64_t timestamp; /* Timestamp of this message               [Tx, Rx]  */
} CMSG_T;
```

Members:

All structure members but *timestamp* are identical to the CMSG structure. Please refer to chapter 6.2.3 for details.

timestamp

64-Bit timestamp (see chapter 3.9 for details).

Note:

Caveat: Basically it is possible to transmit and receive CAN FD messages in a CMSG_T structure if the handle was opened with NTCAN_MODE_FD. If a CAN FD message with more than 8 bytes is transmitted or received the DLC will represent the real data size but the data itself is obviously limited to 8 bytes by the device driver.

6.2.5 CMSG_X

The `CMSG_X` structure contains a complete CAN CC or FD CAN message with the CAN identifier (CAN-ID), the number of data bytes (DLC), additional message specific meta data, up to 64 data bytes and a 64-Bit timestamp.

Syntax:

```
typedef struct
{
   int32_t id;        /* CAN-ID (11-/29-bit) or Event-ID        [Tx, Rx]  */
   uint8_t len;       /* Bit 0-3 = Data Length Code             [Tx, Rx]  */
                      /* Bit 4   = RTR (CAN CC)                 [Tx, Rx]  */
                      /*         = No_BRS (CAN FD)              [Tx, Rx]  */
                      /* Bit 5   = No_Data (Object Mode)        [    Rx]  */
                      /*         = Interaction Data (FIFO Mode)           */
                      /* Bit 6   = Reserved                     [    Rx]  */
                      /* Bit 7   = Type(CAN FD / CAN CC)        [Tx, Rx]  */
   uint8_t msg_lost; /* Counter for lost Rx messages           [    Rx]  */
   uint8_t reserved[1]; /* Reserved                                       */
   uint8_t esi;       /* Error State Indicator (CAN FD)         [    Rx]  */
   uint8_t data[64];    /* 64 data bytes                        [Tx, Rx]  */
   uint64_t timestamp;  /* Timestamp of this message            [Tx, Rx]  */
} CMSG_X;
```

Members:

All structure members are identical to the `CMSG_T` structure with the difference that the message payload can be up to 64 data bytes. Please refer to chapter 6.2.3 and 6.2.4 for details.

Caveat: As for CAN FD messages with payloads of more than 8 bytes the DLC no longer represent the payload size in bytes this information has to be either part of the protocol or has to be part of an agreement between sender and receiver of the message.
The application is responsible to initialize unused protocol bytes in the `CMSG_X` which are transmitted because of the nonconsecutive discrete CAN FD message size to appropriate values.

esi

The Error State Indicator (ESI) of a received CAN FD message is set to `NTCAN_ESI_FD_ERROR_PASSIVE` if the transmitting node is in the state Error Passive (see chapter 3.2).

6.2.6 CMSG_FRAME

The `CMSG_FRAME` structure is part of the `NTCAN_EEI_UNIT` structure and can be initialized with ***canFormatFrame()***.

Syntax:

```
typedef struct _CMSG_FRAME {
  CAN_FRAME_STREAM can_frame;       /* Complete CAN Frame             */
  CAN_FRAME_STREAM stuff_bits;      /* Mask of Stuff bits             */
  uint16_t  crc;                    /* CRC of CAN Frame               */
  uint8_t   length;                 /* Length of CAN Frame in Bit     */
  uint8_t   pos_id11;               /* Position of Identifier 11 Bit  */
  uint8_t   pos_id18;               /* Position of Identifier 18 Bit  */
  uint8_t   pos_rtr;                /* Position of RTR Bit            */
  uint8_t   pos_crtl;               /* Position of Control Field      */
  uint8_t   pos_dlc;                /* Position of DLC Bits           */
  uint8_t   pos_data[8];            /* Position of Data Field         */
  uint8_t   pos_crc;                /* Position of CRC Field          */
  uint8_t   pos_crc_del;            /* Position of CRC delimiter      */
  uint8_t   pos_ack;                /* Position of ACK Field          */
  uint8_t   pos_eof;                /* Position of End of Frame       */
  uint8_t   pos_ifs;                /* Position of Inter Frame Space  */
  uint8_t   reserved[3];
} CMSG_FRAME;
```

Members:

can_frame
Complete CAN Frame as bit stream

stuff_bits
A mask of stuff bits, marked with a 1 at the position of a stuff bit.

crc
Contains the calculated CRC.

pos_id11
Contains the bit position of ID 11

pos_id18
Contains the bit position of ID 18

pos_rtr
Contains the bit position of RTR Bit

pos_crtl
Contains the bit position of CRTL Field

pos_dlc
Contains the bit position of DLC

pos_data[8]
Contains the bit position of Data Field

pos_crc
Contains the bit position of CRC Field

pos_crc_del
Contains the bit position of CRC delimiter bit

pos_ack
Contains the bit position of Acknowledge bit

pos_eof
Contains the bit position of End Of Frame

pos_ifs
Contains the bit position of Inter Frame Space.

6.2.7 CSCHED

The CSCHED structure is the argument of the NTCAN_IOCTL_TX_OBJ_SCHEDULE command of ***canIoctl()*** which is used to configure the scheduling of a CAN message in Tx object mode.

Syntax:

```
typedef struct
{
  int32_t  id;            /* 11-bit or 29-bit CAN identifier         [in] */
  int32_t  flags;         /* Mode configuration bitmask              [in] */
  uint64_t time_start;    /* Start time (absolute or relative)       [in] */
  uint64_t time_interval; /* Interval time                           [in] */
  uint32_t count_start;   /* Start value for counting.               [in] */
  uint32_t count_stop;    /* Stop value for counting.                [in] */
} CSCHED;
```

Members:

id

The 11-bit or 29-bit CAN identifier of an existing TX Object Mode entry. See parameter *id* of CMSG for details.

flags

A bitmask to configure the scheduling behaviour of an object according to the table below:

Flag	Description
NTCAN_SCHED_FLAG_EN	Enable this object for scheduling.
NTCAN_SCHED_FLAG_DIS	Disable this object for scheduling.
NTCAN_SCHED_FLAG_REL	The configured start time is a relative time.
NTCAN_SCHED_FLAG_ABS	The configured start time is a absolute time.
NTCAN_SCHED_FLAG_INC8	The configured counter is an 8-Bit incrementer.
NTCAN_SCHED_FLAG_INC16	The configured counter is a 16-Bit incrementer.
NTCAN_SCHED_FLAG_INC32	The configured counter is a 32-Bit incrementer.
NTCAN_SCHED_FLAG_DEC8	The configured counter is an 8-Bit decrementer.
NTCAN_SCHED_FLAG_DEC16	The configured counter is a 16-Bit decrementer.
NTCAN_SCHED_FLAG_DEC32	The configured counter is a 32-Bit decrementer.
NTCAN_SCHED_FLAG_OFS0	The configured counter starts at CAN data byte 0.
NTCAN_SCHED_FLAG_OFS1	The configured counter starts at CAN data byte 1.
NTCAN_SCHED_FLAG_OFS2	The configured counter starts at CAN data byte 2.
NTCAN_SCHED_FLAG_OFS3	The configured counter starts at CAN data byte 3.
NTCAN_SCHED_FLAG_OFS4	The configured counter starts at CAN data byte 4.
NTCAN_SCHED_FLAG_OFS5	The configured counter starts at CAN data byte 5.
NTCAN_SCHED_FLAG_OFS6	The configured counter starts at CAN data byte 6.
NTCAN_SCHED_FLAG_OFS7	The configured counter starts at CAN data byte 7.

Table 26: Flags to configure the scheduling in TX Object Mode

time_start
Start time to schedule the object which is interpreted as an absolute time if `NTCAN_SCHED_FLAG_ABS` is set in *flags* otherwise a relative time (see chapter 3.9 for details).

time_interval
Interval time for a periodic scheduling or 0 for a single-shot configuration (see chapter 3.9 for details).

count_start
Counter start value if an incrementer or decrementer is configured in *flags* for this object. If no counter start position is configured the default position is data byte 0.

count_stop
Counter stop value if an incrementer or decrementer is configured for this object. After reaching this value, the counter is set to *count_start*.

6.2.8 EV_BAUD_CHANGE

The EV_BAUD_CHANGE structure is the payload of the NTCAN_EV_BAUD_CHANGE event which is signalled each time the CAN controller changes the bit rate. If the CAN controller is operated in the CAN FD mode the application will receive two consecutive events. The first event contains the bit rate configured for the data phase and the second event the configured nominal bit rate.

Syntax:

```
typedef struct
{
    uint32_t    baud;       /* New NTCAN baudrate value                  */
    uint32_t    num_baud;   /* New numerical baudrate value (optional)   */
} EV_CAN_BAUD_CHANGE;
```

Members:

baud

New baudrate parameter as returned by ***canGetBaudrate()*** or the value NTCAN_BAUD_FD for the configured data phase bit rate.

num_baud

New bit rate as numerical value in Bit/s.

6.2.9 EV_CAN_ERROR

The EV_CAN_ERROR structure is the payload of the NTCAN_EV_CAN_ERROR event which is signalled each time the CAN controller state has changed or an internal problem processing received messages occurred.

Syntax:

```
typedef struct
{
    uint8_t     reserved1;      /* Reserved for future use              */
    uint8_t     can_status;     /* CAN controller status                */
    uint8_t     dma_stall;      /* DMA stall counter (HW dependent)     */
    uint8_t     ctrl_overrun;   /* Controller overruns                  */
    uint8_t     reserved3;      /* Reserved for future use              */
    uint8_t     fifo_overrun;   /* Driver FIFO overruns                 */
} EV_CAN_ERROR;
```

Members:

can_status

The current CAN controller status according to the table below (see chapter 3.2 for details):

Flag	Value	Description
NTCAN_BUSSTATE_OK	0x00	The controller status is in state *Error Active*.
NTCAN_BUSSTATE_WARN	0x40	The CAN controller warning limit is exceeded.
NTCAN_BUSSTATE_ERRPASSIVE	0x80	The controller status is in state *Error Passive*.
NTCAN_BUSSTATE_BUSOFF	0xC0	The controller status is in state Bus Off.

Table 27: CAN controller state

dma_stall

This value is incremented if a bus master DMA capable hardware has stalled it's DMA state machine as otherwise CAN messages in the FIFO of the host DMA memory would be overwritten. The reason is usually that the hosts kernel or system thread which should process the CAN messages are not scheduled in time. This indication does not mean that messages are lost if not received in combination with one of the other counters. After this event is fired this counter is reset to 0.

ctrl_overrun

This value is incremented if the CAN controller was unable to receive a CAN message because its internal buffer was overrun. This is usually the result of a host system interrupt or kernel/system thread which isn't scheduled fast enough. After this event is fired this counter is reset to 0.

fifo_overrun

According to driver and hardware architecture the CAN controller interrupt handler stores a received message into a common FIFO for later processing. This value is incremented each time the CAN driver was unable to store a received CAN message because this internal FIFO was overrun. This is usually the result of an overloaded target system. After this event is fired this counter is reset to 0.

6.2.10 EV_CAN_ERROR_EXT

The `EV_CAN_ERROR_EXT` structure is the payload of the `NTCAN_EV_CAN_ERROR_EXT` event which is signalled each time the CAN controller detects an error on the bus. The support for this event and the payload is very CAN controller specific. If the event is supported by the CAN controller hardware this capability allows a very detailed analysis of a bus error situation.

Syntax:

```
typedef union
{
    struct {
        uint8_t     status;     /* (SJA1000) CAN controller status       */
        uint8_t     ecc;        /* Error Capture Register                */
        uint8_t     rec;        /* Rx Error Counter                      */
        uint8_t     tec;        /* Tx Error Counter                      */
    } sja1000;
    struct {
        uint8_t     status;     /* (esdACC) CAN controller status        */
        uint8_t     ecc;        /* Error Capture Register                */
        uint8_t     rec;        /* Rx Error Counter                      */
        uint8_t     tec;        /* Tx Error Counter                      */
        uint8_t     txstatus;   /* (esdACC) CAN controller TX status     */
    } esdacc;
} EV_CAN_ERROR_EXT;
```

Members:

sja1000.status
The SJA1000 *Status Register*. Please refer to table 27 for details.
sja1000.ecc
The SJA1000 *Error Code Capture* register. Please refer to to Annex B: for details.
sja1000.rec
The value of the *Receive Error Counter* register.
sja1000.tec
The value of the *Transmit Error Counter* register.

esdacc.status
The ESDACC *Status Register*. Please refer to table 27 for details.
esdacc.ecc
The ESDACC *Error Code Capture* register. Please refer to Annex B: for details.
esdacc.rec
The value of the *Receive Error Counter* register.
esdacc.tec
The value of the *Transmit Error Counter* register.
esdacc.txstatus
The value of the controller transmit status register.

6.2.11 EV_GPIO_DATA

The EV_GPIO_DATA structure is the payload of the NTCAN_EV_GPIO_GET_DO, NTCAN_EV_GPIO_GET_DI, NTCAN_EV_GPIO_SET_DO and NTCAN_EV_GPIO_SET_DIR events which are sent to or received by the driver to request or receive the current state of the GPIOs. The support for this event and the number of supported IO channels is hardware specific.

Syntax:

```
typedef struct
{
    uint32_t    value;       /* GPIO channel value            */
    uint32_t    mask;        /* GPIO channel (change) mask    */
} EV_GPIO_DATA;
```

Members:

value
Value of the GPIO channel 0..31.

mask
Bitmask for GPIO channel 0..31. The mask is only valid if the length parameter of the event message is set to 8 bytes. If received as payload for the NTCAN_EV_GPIO_GET_DI event, this parameter indicates which channels have changed. If transmitted as payload for the NTCAN_EV_GPIO_SET_DO and NTCAN_EV_GPIO_SET_DIR event, this parameter indicates which channels should be changed.

6.2.12 EVMSG

This message is sent by the CAN driver or firmware as out-of-band data to indicate hardware state changes, error situations, etc. It consists of an 8-Bit event ID, an event specific payload and the length of this payload in bytes.

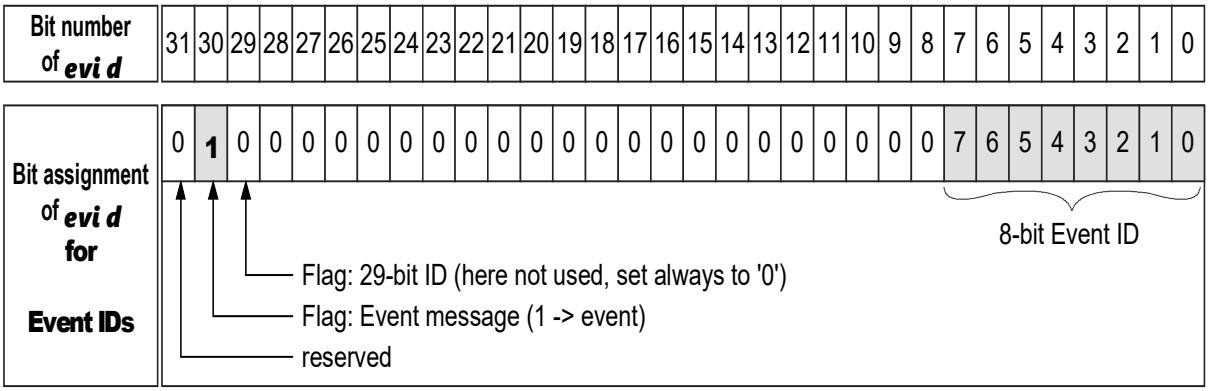
Syntax:

```
typedef struct
{
  int32_t    evid;        /* event-id: possible range:EV_BASE...EV_LAST */
  uint8_t    len;         /* length of message: 0-8                     */
  uint8_t    reserved[3]; /* reserved                                   */
  union
  {
    uint8_t            c[8];
    uint16_t           s[4];
    uint32_t           l[2];
    uint64_t           q;
    EV_CAN_ERROR       error;
    EV_CAN_BAUD_CHANGE baud_change;
    EV_CAN_ERROR_EXT   error_ext;
    EV_GPIO_DATA       gpio;
  } evdata;
} EVMSG;
```

Members:

evid

Identifier of the event message. Bit 30 of *evid* is used to distinguish an event from standard CAN messages. In order to transmit an event this bit has to be set in addition to the event identifier. This can be achieved by bitwise OR the identifier with `NTCAN_EV_BASE` (defined in **<ntcan.h>**).



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

In **<ntcan.h>** in addition to NTCAN_EV_BASE (0x40000000) the constants NTCAN_EV_USER (0x40000080) and NTCAN_EV_LAST (0x400000FF) are defined.

The EVMSG has the same structure size and layout as the CMSG so it can be received together with CAN messages with ***canRead()*** or ***canTake()*** in FIFO mode. The application has to test if bit 30 (NTCAN_EV_BASE) is set to perform a C cast operation to EVMSG for further event handling.

---

len

The structure member *len* indicates the number of data bytes of the event specific payload.

Value of *len* [binary]		**Pyload size**
Bit 7..4	Bit 3..0	[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

Bits 7...4 are reserved for future use and should be set to ‘0’.

evdata

This union contains the event specific payload.

6.2.13 EVMSG_T

This message is sent by the CAN driver or firmware as out-of-band data to indicate hardware state changes, error situations, etc. It consists of an 8-Bit event ID, an event specific payload, the length of this payload in bytes and a 64-Bit timestamp.

Syntax:

```
typedef struct
{
  int32_t   evid;          /* Event-id: possible range:EV_BASE...EV_LAST */
  uint8_t   len;           /* Length of message: 0-8                     */
  uint8_t   reserved[3];   /* Reserved                                   */
  union
  {
    uint8_t            c[8];
    uint16_t           s[4];
    uint32_t           l[2];
    uint64_t           q;
    EV_CAN_ERROR       error;
    EV_CAN_BAUD_CHANGE baud_change;
    EV_CAN_ERROR_EXT   error_ext;
    EV_GPIO_DATA       gpio;
  } evdata;
  uint64_t   timestamp;   /* Time stamp of this message                 */
} EVMSG_T;
```

Members:

All structure members but *timestamp* are identical to the `EVMSG` structure. Please refer to chapter 6.2.12 for details.

The `EVMSG_T` has the same structure size and layout as the `CMSG_T` so it can be received together with CAN messages with ***canReadT()*** or ***canTakeT()*** in FIFO mode. The application has to test if bit 30 (`NTCAN_EV_BASE`) is set to perform a C cast operation to `EVMSG_T` for further event handling.

Timestamp

64-Bit timestamp (see chapter 3.9 for details).

6.2.14 EVMSG_X

This message is sent by the CAN driver or firmware as out-of-band data to indicate hardware state changes, error situations, etc. It consists of an 8-Bit event ID, an event specific payload, the length of this payload in bytes and a 64-Bit timestamp.

Syntax:

```
typedef struct
{
  int32_t   evid;          /* Event-id: possible range:EV_BASE...EV_LAST */
  uint8_t   len;           /* Length of message: 0-8                     */
  uint8_t   reserved[3];   /* Reserved                                   */
  union
  {
    uint8_t                 c[64];
    uint16_t                s[32];
    uint32_t                l[16];
    uint64_t                q[8];
    EV_CAN_ERROR            error;
    EV_CAN_BAUD_CHANGE      baud_change;
    EV_CAN_ERROR_EXT        error_ext;
    EV_GPIO_DATA            gpio;
  } evdata;
  uint64_t   timestamp;   /* Time stamp of this message                 */
} EVMSG_T;
```

Members:

All structure members but the size of *evdata* are identical to the `EVMSG_T` structure. Please refer to chapter 6.2.13 for details.

The `EVMSG_X` has the same structure size and layout as the `CMSG_X` so it can be received together with CAN messages with ***canReadX()*** or ***canTakeX()*** in FIFO mode. The application has to test if bit 30 (`NTCAN_EV_BASE`) is set to perform a C cast operation to `EVMSG_X` for further event handling.

6.2.15 NTCAN_BAUDRATE_CFG

The NTCAN_BAUDRATE_CFG union is part of the NTCAN_BAUDRATE_X structure to define the nominal or the data bit rate. The union member is defined by the parameter *mode* of NTCAN_BAUDRATE_X.

Syntax:

```
typedef struct {
    union {
        uint32_t idx;        /* esd electronics bit rate table index
*/
        uint32_t rate;       /* Numerical bit rate                    */
        uint32_t btr_ctrl;   /* BTR register (CAN Controller layout)  */
        struct {
            uint16_t brp;    /* Bit rate pre-scaler                   */
            uint16_t tseg1;  /* TSEG1 register                        */
            uint16_t tseg2;  /* TSEG2 register                        */
            uint16_t sjw;    /* SJW register                          */
        } btr;
    } u;
} NTCAN_BAUDRATE_CFG
```

Members:

idx

Valid if parameter *mode* of NTCAN_BAUDRATE_X is set to NTCAN_BAUDRATE_MODE_INDEX.
For a **CAN CC** configuration any index of the esd electronics (CiA) bit rate configuration (see table 16) for the nominal bit rate can be used. For a **CAN FD** configuration additional index values for the data bit rate are available but only a limited number of combinations of the ratio between nominal bit rate and data bit rate are supported (see table below).

Table index [hex]	Bit Rate [kBit/s]	NTCAN-API Constant	Supported nominal bit rate [kBit/s]
0x02	500	NTCAN_BAUD_500	250
0x00	1000	NTCAN_BAUD_1000	250, 500
0x11	2000	NTCAN_BAUD_2000	250, 500, 1000
0x12	4000	NTCAN_BAUD_4000	250, 500, 1000
0x13	5000	NTCAN_BAUD_5000	250, 500, 1000
0x14	8000	NTCAN_BAUD_8000	500, 1000
0x15	10000	NTCAN_BAUD_10000	500, 1000

Table 28: **esd electronics** Data Phase Bit Rate Table

rate

Valid if parameter *mode* of NTCAN_BAUDRATE_X is set to NTCAN_BAUDRATE_MODE_NUM.
For a **CAN CC** configuration any numerical value up to 1000000 can be used. For a **CAN FD** configuration only the numerical values of the supported combinations listed in table 28 are supported.

btr_ctrl

Valid if parameter *mode* of NTCAN_BAUDRATE_X is set to NTCAN_BAUDRATE_MODE_BTR_CTRL.
The BTR register in a controller specific representation (see table 17).

btr

Valid if parameter *mode* of `NTCAN_BAUDRATE_X` is set to `NTCAN_BAUDRATE_MODE_BTR_CANONICAL`. A structure which contains the bit timing configuration parameter (BRP, TSEG1, TSEG2 and SJW) in a canonical format.

The current ESDACC CAN FD implementation uses a shared prescaler for the bit rate configuration (see /2/) of the arbitration and the data phase. For this reason the BRP value of the data phase configuration is ignored and only the BRP value of the arbitration phase value is configured.

6.2.16 NTCAN_BAUDRATE_X

The NTCAN_BAUDRATE_X structure is used to define the CAN bit timing configuration of the CAN controller in CAN FD mode as well as CAN CC mode.

Syntax:

```
typedef struct {
    uint16_t            mode;       /* Mode word                          */
    uint16_t            flags;      /* Control flags                      */
    NTCAN_TDC_CFG       tdc;        /* TDC configuration parameters       */
    NTCAN_BAUDRATE_CFG  arb;        /* Nominal bit rate configuration     */
    NTCAN_BAUDRATE_CFG  data;       /* Data bit rate configuration        */
} NTCAN_BAUDRATE_X;
```

Members:

mode

The bit rate configuration mode.

Mode	Description
NTCAN_BAUDRATE_MODE_DISABLE	Detach the CAN port from the CAN bus.
NTCAN_BAUDRATE_MODE_INDEX	Configuration mode for the NTCAN_BAUDRATE_CFG unions *arb* and/or *data.*
NTCAN_BAUDRATE_MODE_BTR_CTRL	Configuration mode for the NTCAN_BAUDRATE_CFG unions *arb* and/or *data.*
NTCAN_BAUDRATE_MODE_BTR	Configuration mode for the NTCAN_BAUDRATE_CFG unions *arb* and/or *data.*
NTCAN_BAUDRATE_MODE_NUM	Configuration mode for the NTCAN_BAUDRATE_CFG unions *arb* and/or *data.*
NTCAN_BAUDRATE_MODE_AUTOBAUD	Start automatic bit rate detection as described in chapter 3.3.7 (only supported for CAN CC configurations).

flags

Flags to control the CAN bus operation mode. Any flag to enable a mode which is not supported by the hardware is ignored silently.

Flags	Description
NTCAN_BAUDRATE_FLAG_FD	Has to be set to configure the CAN port to the CAN FD operation mode where the bit rate configuration in the union *arb* for the nominal bit rate as well as the bit rate configuration in the union *data* for the data phase is considered. If unset the CAN port is configured to operate in the CAN CC operation mode and the union *data* is ignored.
NTCAN_BAUDRATE_FLAG_LOM	Enable the listen only mode (see chapter 3.3.2)
NTCAN_BAUDRATE_FLAG_STM	Enable the self test mode (see chapter 3.3.3*)*
NTCAN_BAUDRATE_FLAG_TRS	Enable the triple sampling mode (see chapter 3.3.4*)*
NTCAN_BAUDRATE_FLAG_TXP	Enable the transmit pause mode (see chapter 3.3.5*)*

Flags	Description
NTCAN_BAUDRATE_FLAG_TDC	If the flag is set, the member *tdc* of the data type NTCAN_TDC_CFG contains valid parameters.
NTCAN_BAUDRATE_FLAG_DAR	If the flag is set, retries of failed transmissions in case of bus errors or a lost arbitration procedure are disabled (see chapter 3.3.6*)*

tdc

Configuration parameters of the TDC mechanism during the data phase in CAN FD mode. The data is only valid if the flag NTCAN_BAUDRATE_FLAG_TDC in *flags* is set.

arb

Configuration of the nominal bit rate for CAN FD mode as well as CAN CC mode.

data

Configuration of the data bit rate for CAN FD mode.

The current ESDACC CAN FD implementation uses a shared prescaler (see /2/) for the nominal bit rate and the data bit rate. For this reason the BRP value of the data phase configuration in mode NTCAN_BAUDRATE_MODE_BTR/NTCAN_BAUDRATE_MODE_BTR_CTRL is ignored and the BRP value of the arbitration phase value is used, too.

6.2.17 NTCAN_BITRATE

The NTCAN_BITRATE structure is initialized by ***canIoctl()*** if called with the command NTCAN_IOCTL_GET_BITRATE_DETAILS with a detailed information about the configured bit rate.

Syntax:

```
typedef struct
{
  uint32_t  baud;         /* value configured by user via canSetBaudrate()       */
                          /*  or NTCAN_BAUD_FD if canSetBaudrateX() was used     */
  uint32_t  valid;        /* validity of all _following_ infos                   */
                          /* (-1 = invalid, NTCAN_SUCCESS                        */
  uint32_t  rate;         /* CAN nominal/arbitration bitrate in Bit/s            */
  uint32_t  clock;        /* Clock frequency of CAN controller                   */
  uint8_t   ctrl_type;    /* NTCAN_CANCTL_XXX defines                             */
  uint8_t   tq_pre_sp;    /* Number of time quanta before samplep. (SYNC + TSEG1)*/
  uint8_t   tq_post_sp;   /* Number of time quanta after samplepoint (TSEG2)     */
  uint8_t   sjw;          /* Synchronization jump width in time quantas (SJW)    */
  uint32_t  error;        /* Actual deviation of configured baudrate in (% * 100)*/
  uint32_t  flags;        /* Baudrate flags (possibly ctrl. specific, e.g. SAM)  */
  uint32_t  rate_d;       /* CAN data phase bit rate in bit/s                    */
  uint8_t   tq_pre_sp_d;  /* Number of time quantas before samplepoint (DTSEG1)  */
  uint8_t   tq_post_sp_d; /* Number of time quantas past samplepoint (DTSEG2)    */
  uint8_t   sjw_d;        /* Syncronization jump width in time quanta(DSJW)      */
  uint8_t   mode;         /* NTCAN_BAUDRATE_MODE_XXX defines                     */
  uint32_t  reserved[1];  /* for future use                                      */
} NTCAN_BITRATE;
```

Members:

baud

The bit rate value configured with ***canSetBaudrate()***. If the bit rate was configured with ***canSetBaudrateX()*** this value is set to NTCAN_BAUD_FD.

valid

Set to NTCAN_SUCCESS if the data of all members is valid. Any other value indicates invalid information.

rate

The nominal configured bit rate in Bit/s of CAN CC and CAN FD.

clock

The clock frequency of the CAN controller in Hz. You need this information together with *ctrl_type* and the CAN controller data sheet to configure the bit rate register (BRP, TSEG1, TSEG2, SJW) with ***canSetBaudrate()*** or ***canSetBaudrateX()*** directly.

ctrl_type

CAN controller type according to table 21.

tq_pre_sp

Number of time quanta before the sample point (SYNC + TSEG1) for the configured nominal bit rate. To be precise this is the sum of the time quanta, which belong to “Sync Segment”, “Propagation Segment” and “Phase Segment 1”, as described in /2/.

tq_post_sp
Number of time quanta after the sample point (TSEG2) for the configured nominal bit rate. In /2/ this is described as “Phase Segment 2”.

sjw
Number of time quanta of the *Synchronous Jump Width* (SJW) for the configured nominal bit rate.

error
Deviation of the desired bit rate in percent multiplied with 100 if the exact bit rate can not be matched because of hardware restrictions. This information is only populated for a CAN CC bit rate configuration with a numerical value. For a CAN FD configuration it is set to 0.

flags
Controller specific flags.

Flag	Controller	Description
NTCAN_BITRATE_FLAG_SAM	N/A	Set if triple sampling (see chapter 3.3.4) is supported and active.

rate_d
The current bit rate in Bit/s for the data phase of a CAN FD configuration. For a CAN CC configuration it is set to 0.

tq_pre_sp_d
Number of time quanta before the sample point (SYNC + TSEG1) for the configured data phase bit rate. To be precise this is the sum of the time quanta, which belong to “Sync Segment”, “Propagation Segment” and “Phase Segment 1”, as described in /2/. For a CAN CC configuration it is set to 0.

tq_post_sp_d
Number of time quanta after the sample point (TSEG2) for the configured configured data phase bit rate. In /2/ this is described as “Phase Segment 2”. For a CAN CC configuration it is set to 0.

sjw_d
Number of time quanta of the Synchronous Jump Width (SJW) for the configured data phase bit rate. For a CAN CC configuration it is set to 0.

reserved
Reserved for future use.

6.2.18 NTCAN_BUS_STATISTIC

The NTCAN_BUS_STATISIC structure is initialized by ***canIoctl()*** if called with the command NTCAN_IOCTL_GET_BUS_STATISTIC with detailed statistical information about the CAN bus communication.

Syntax:

```
typedef struct
{
  uint64_t           timestamp;         /* Timestamp                     */
  NTCAN_FRAME_COUNT  rcv_count;         /* # of received frames          */
  NTCAN_FRAME_COUNT  xmit_count;        /* # of transmitted frames       */
  uint32_t           ctrl_ovr;          /* # of controller overruns      */
  uint32_t           fifo_ovr;          /* # of FIFO overflows           */
  uint32_t           err_frames;        /* # of error frames             */
  uint32_t           rcv_byte_count;    /* # of received bytes           */
  uint32_t           xmit_byte_count;   /* # of transmitted bytes        */
  uint32_t           aborted_frames;    /* # of aborted frames           */
  uint32_t           rcv_count_fd;      /* # of received CAN FD frames   */
  uint32_t           xmit_count_fd;     /* # of transmitted CAN FD frames */
  uint64_t           bit_count;         /* # of received bits            */
} NTCAN_BUS_STATISTIC;
```

Members:

timestamp
64-Bit timestamp (see chapter 3.9 for details) the statistical data is captured.

rcv_count
Number of received CAN frames as NTCAN_FRAME_COUNT structure subdivided according to the frame type.

xmit_count
Number of transmitted CAN frames as NTCAN_FRAME_COUNT structure subdivided according to the frame type.

ctrl_ovr
Number of controller overruns. The accumulated value which is also indicated with the EV_CAN_ERROR event.

fifo_ovr
Number of FIFO overruns. The accumulated value which is also indicated with the EV_CAN_ERROR event.

err_frames
Number of received error frames (if supported by CAN controller hardware).

rcv_byte_count
Number of received data bytes.

xmit_byte_count
Number of transmitted data bytes.

aborted_frames
Number of aborted frames. Aborting a frame can be forced explicitly by the application or is done implicitly because a transmission timeout is exceeded.

rcv_count_fd
Number of received CAN FD frames. The total number of received CAN CC frames is the difference between the sum of all counters of *rcv_count* and this value.

xmit_count_fd
Number of transmitted CAN FD frames. The total number of transmitted CAN CC frames is the difference between the sum of all counters of *xmit_count* and this value.

bit_count
Number of bits on the CAN bus. This value in combination with the timestamp can be used to calculate a bus load.

For the reason of synchronization a CAN controller inserts stuff-bits into a CAN frame (see 3.2 for details). These bits are already removed by any CAN controller if the CAN frame is passed to the device driver and calculating the number of stuff-bits at run time would degrade the system performance too much. For this reason the number of stuff-bits is derived from a table with empiric data which usually returns good results.

The FPGA based esd electronics *Advanced CAN Controller* (esdACC) allows access to the exact number of stuff-bits.

Remarks:
All counters will wrap around without notice if the maximum value is exceeded which can be stored in its data type.

6.2.19 NTCAN_CTRL_STATE

The NTCAN_CTRL_STATE structure is initialized by ***canIoctl()*** if called with the command NTCAN_IOCTL_GET_CTRL_STATE with information about the current CAN controller bus state (see chapter 3.2 for details).

Syntax:

```
typedef struct
{
  uint8_t    rcv_err_counter;     /* Receive error counter         */
  uint8_t    xmit_err_counter;    /* Transmit error counter        */
  uint8_t    status;              /* CAN controller status         */
  uint8_t    type;                /* CAN controller type           */
} NTCAN_CTRL_STATE;
```

Members:

rcv_err_counter
Current CAN controller *Receive Error Counter*.

xmit_err_counter
Current CAN controller *Transmit Error Counter*.

status
Current CAN controller bus status according to table 27.

type
CAN controller type according to table 21.

6.2.20 NTCAN_EEI_STATUS

The NTCAN_EEI_STATUS structure is part of Error Injection and the argument of ***canIoctl()*** with the command NTCAN_IOCTL_EEI_STATUS.

Syntax:

```
typedef struct _NTCAN_EEI_STATUS {
  uint32_t  handle;              /* Handle for ErrorInjection Unit       */
  uint8_t   status;              /* Status form Unit                     */
  uint8_t   unit_index;          /* Error Injection Unit ID              */
  uint8_t   units_total;         /* Max Error Units in esdacc core       */
  uint8_t   units_free;          /* Free Error Units in esdacc core      */
  uint64_t  trigger_timestamp;   /* Timestamp of trigger time            */
  uint16_t  trigger_cnt;         /* Count of trigger in Repeat mode      */
  uint16_t  reserved0;
  uint32_t  reserved[27];
} NTCAN_EEI_STATUS;
```

Members:

handle
Handle for an Error Injection Unit, returned from NTCAN_IOCTL_EEI_CREATE

status
The status of an Error Injection Unit:

0 EEI_STATUS_OFF
1 EEI_STATUS_WAIT_TRIGGER
2 EEI_STATUS_SENDING
3 EEI_STATUS_FINISHED

unit_index
Index of the Error Injection Unit.

units_total
Number of Error Injection Units.

units_free
Number of free Error Injection Units.

trigger_timestamp
Timestamp (see chapter 3.9) of the trigger time.

trigger_cnt
Count of trigger in Repeat mode

6.2.21 NTCAN_EEI_UNIT

The NTCAN_EEI_UNIT structure is part of the Error Injection implementation and the argument of ***canIoctl()*** with the command NTCAN_IOCTL_EEI_CONFIGURE.

Syntax:

```
typedef struct _NTCAN_EEI_UNIT {
  uint32_t  handle;                   /* Handle for ErrorInjection Unit */
  uint8_t   mode_trigger;             /* Trigger mode                   */
  uint8_t   mode_trigger_option;      /* Options to trigger             */
  uint8_t   mode_triggerarm_delay;    /* Enable delayed arming of trigger
                                         unit*/
  uint8_t   mode_triggeraction_delay; /* Enable delayed TX out          */
  uint8_t   mode_repeat;              /* Enable repeat                  */
  uint8_t   mode_trigger_now;         /* Trigger with next TX point     */
  uint8_t   mode_ext_trigger_option;  /* Switch between trigger and sending
  uint8_t   mode_send_async;          /* Send without timing
                                         synchronization*/
  uint8_t   reserved1[4];
  uint64_t  timestamp_send;           /* Timestamp for Trigger Timestamp*/
  CAN_FRAME_STREAM trigger_pattern;   /* Trigger for mode Pattern Match */
  CAN_FRAME_STREAM trigger_mask;      /* Mask to trigger Pattern        */
  uint8_t   trigger_ecc;              /* ECC for Trigger Field Position */
  uint8_t   reserved2[3];
  uint32_t  external_trigger_mask;    /* Enable Mask for external Trigger*/
  uint32_t  reserved3[16];
  CAN_FRAME_STREAM tx_pattern;        /* TX pattern                      */
  uint32_t  tx_pattern_len;           /* Length of TX pattern            */
  uint32_t  triggerarm_delay;         /* Delay for mode triggerarm delay */
  uint32_t  triggeraction_delay;      /* Delay for mode trigger delay    */
  uint32_t  number_of_repeat;         /* Number of repeats in mode repeat*/
  uint32_t  reserved4;
  CAN_FRAME_STREAM tx_pattern_recessive; /* Internal use only (set to 0   */
  uint32_t  reserved5[9];
} NTCAN_EEI_UNIT;
```

Members:

handle

Handle for an Error Injection Unit returned by ***canIoctl()*** called with the command NTCAN_IOCTL_EEI_CREATE.

mode_trigger

Trigger Mode:

0 Trigger Pattern Matching (EEI_TRIGGER_MATCH)
1 Trigger Arbitration (EEI_TRIGGER_ARBITRATION)
2 Trigger Timestamp (EEI_TRIGGER_TIMESTAMP)
3 Trigger Field Position (EEI_TRIGGER_FIELD_POSITION)
4 Trigger External Input (EEI_TRIGGER_EXTERNAL_INPUT)

mode_trigger_option

Some Trigger Modes have an Option

- Trigger Pattern Matching
  Compare with destuffed sampled bits (EEI_TRIGGER_MATCH_OPTION_DESTUFFED)
- Trigger Arbitration:
  Abort on Error Frame (EEI_TRIGGER_ARBITRATION_OPTION_ABORT_ON_ERROR)
- Trigger Timestamp
  Trigger on bus free (EEI_TRIGGER_TIMESTAMP_OPTION_BUSFREE)

mode_triggerarm_delay

Enable the arm delay in repeat mode to delay the next activation of the trigger module.

mode_triggeraction_delay
Enable the action delay to delay the output of the trigger module.

mode_repeat
Enable the repeat mode. After the CAN TX module finished the Error Injection Unit will be reactivated.

mode_trigger_now
Set trigger now to send immediately after enabling the Error Injection Unit.

mode_ext_trigger_option
Only for the Trigger out on GPIO Pin
0: The trigger out sends only a puls on trigger time.
1: The trigger out is HIGH during CAN TX is active.

mode_send_async
Send without timing synchronization

timestamp_send
Timestamp (see chapter 3.9) for Trigger Timestamp.

trigger_pattern
Trigger Pattern for Trigger Pattern Matching.

trigger_mask
Trigger Pattern Mask for Trigger Pattern Matching.

trigger_ecc
Trigger ECC for Trigger Field Position. It will only use the lower 5 bits in the coding of the ECC register form NXP SJA1000

external_trigger_mask
External Trigger Mask for Trigger Ext. Trigger. The Bits 0 to 3 are the Trigger Out of the Error Injection Units. Bit 31 is the ext. Trigger In over a GPIO Pin.

tx_pattern
This TX Pattern would be sent by the CAN TX module on trigger time.

tx_pattern_len
The length in bits of the TX Pattern.

triggerarm_delay
Delay in bit times.

triggeraction_delay
Delay in bit times.

number_of_repeat
Number of repeats in mode repeat (0 = forever)

tx_pattern_recessive
Internal use only (has to be setto 0).

6.2.22 NTCAN_FORMATEVENT_PARAMS

The NTCAN_FORMATEVENT_PARAMS structure is one argument of ***canFormatEvent()***.

Syntax:

```
typedef struct
{
  uint64_t       timestamp;         /* Timestamp (for busload)          */
  uint64_t       timestamp_freq;    /* Timestamp frequency (for busload)*/
  uint32_t       num_baudrate;      /* Numerical baudrate (for busload) */
  uint32_t       flags;             /* Flags                            */
  uint64_t       busload_oldts;     /* <---+-- used internally, set to  */
  uint64_t       busload_oldbits;   /* <---+   zero on first call       */
  uint8_t        ctrl_type;         /* Controller type (for ext_error)  */
  uint8_t        reserved[7];       /* Reserved (7 bytes)               */
  uint32_t       reserved2[4];      /* Reserved (16 bytes)              */
} NTCAN_FORMATEVENT_PARAMS;
```

Members:

timestamp
The current timestamp.

timestamp_freq
The timestamp frequency of the CAN board used in combination with the NTCAN_EV_BUSLOAD event. This parameter can be obtained with ***canIoctl()*** and the argument NTCAN_IOCTL_GET_TIMESTAMP_FREQ.

num_baudrate
The numerical CAN bit rate of the CAN board used in combination with the NTCAN_EV_BUSLOAD event. This parameter can be obtained with **canIoctl()** and the argument NTCAN_IOCTL_GET_BITRATE_DETAILS.

flags
Special flags.

busload_oldts
Used internally to keep previous timestamp. Has to be set to 0 with the initial call.

busload_oldbits
Used internally to keep previous value of received bits. Has to be set to 0 with the initial call.

ctrl_type
The controller type of the CAN board used in combination with the NTCAN_EV_EXT_ERROR event. This parameter can be obtained with ***canStatus().***

Remarks:
All counters will wrap around without notice if the maximum value which can be stored in a variable of type uint32_t is exceeded.

6.2.23 NTCAN_FILTER_MASK

The `NTCAN_FILTER_MASK` structure is the argument of ***canIoctl()*** with the command `NTCAN_IOCTL_SET_HND_FILTER`.

Syntax:

```
typedef struct
{
    uint32_t acr;          /* Acceptance Code Register */
    uint32_t amr;          /* Acceptance Mask Register */
    uint32_t idArea;       /*
} NTCAN_FILTER_MASK;
```

Members:

acr

Acceptance code register for the area defined by *idArea*.

amr

Acceptance mask register for the area defined by *idArea*.

idArea

The NTCAN-ID area where the acr and amr are applied. The following arguments are supported:

`NTCAN_IDS_REGION_20A`:

11-bit CAN-Ids in the range from 0x00000000...0x000007FF.

`NTCAN_IDS_REGION_20B`:

29-bit CAN-Ids in the range from 0x20000000...0x3FFFFFFF.

`NTCAN_IDS_REGION_EV`:

CAN-Events in the range from 0x40000000...0x400000FF.

Remarks:

If *amr* exceeds the range defined by *idArea* the mask is reduced to this range and *acr* is always masked with this *amr*.

6.2.24 NTCAN_FRAME_COUNT

The NTCAN_FRAME_COUNT structure is part of NTCAN_BUS_STATISTIC.

Syntax:

```
typedef struct {
  uint32_t           std_data;           /* # of std CAN messages    */
  uint32_t           std_rtr;            /* # of std RTR requests    */
  uint32_t           ext_data;           /* # of ext CAN messages    */
  uint32_t           ext_rtr;            /* # of ext RTR requests    */
} NTCAN_FRAME_COUNT;
```

Members:

std_data
Number of CAN data frames in standard frame format (11-bit CAN-IDs).

std_rtr
Number of CAN Remote Request (RTR) frames in standard frame format.

ext_data
Number of CAN data frames in extended frame format (29-bit CAN-IDs).

ext_rtr
Number of CAN Remote Request (RTR) frames in extendef frame format.

Remarks:

All counters will wrap around without notice if the maximum value which can be stored in a variable of type uint32_t is exceeded.

6.2.25 NTCAN_GPIO_CFG

The NTCAN_GPIO_CFG structure is used to get or set the configuration of GPIO ports as argument of ***canIoctl()*** with the commands NTCAN_IOCTL_GET_GPIO_CFG and NTCAN_IOCTL_SET_GPIO_CFG.

Syntax:

```
typedef struct _NTCAN_GPIO_CFG {
    uint8_t     channel;                /* I/O channel                 */
    uint8_t     reserved;               /* Reserved for alignment      */
    uint16_t    properties;             /* Configuration property mask */
    uint8_t     direction;              /* I/O channel direction       */
    uint8_t     voltage;                /* I/O channel voltage         */
    uint8_t     pull;                   /* I/O channel pulling         */
    uint8_t     irq_mode;               /* I/O channel IRQ mode flags  */
    uint32_t    input_filter;           /* (Global) Input filter       */
} NTCAN_GPIO_CFG;
```

Members:

channel
Channel/Port number 0… (number of channels – 1) as returned in NTCAN_INFO.

properties
Bitmask according to Table 30 which defines the supported configuration options when called with NTCAN_IOCTL_GET_GPIO_CFG. The bits must be set to indicate the configuration options which should be evaluated by the driver when called with NTCAN_IOCTL_SET_GPIO_CFG.

direction
I/O direction configured as

- Digital input (NTCAN_GPIO_CFG_DIR_IN)
- Dgital output in Push-Pull mode (NTCAN_GPIO_CFG_DIR_OUT)
- Dgital output in Low-Side mode (NTCAN_GPIO_CFG_DIR_OUT_LS)
- Dgital output in High-Side mode (NTCAN_GPIO_CFG_DIR_OUT_HS)

voltage
I/O voltage configured as

- 3,3V (NTCAN_GPIO_CFG_VOLTAGE_3V3)
- 5V (NTCAN_GPIO_CFG_VOLTAGE_5V)

pull_mode
I/O pull mode configured as

- No pulling (NTCAN_GPIO_CFG_PULL_NONE)
- Pull-Up (NTCAN_GPIO_CFG_PULL_UP)
- Pull-Down (NTCAN_GPIO_CFG_PULL_DOWN)

irq_mode
IRQ mode mask (for inputs) configured to:

- Never indicate an I/O change (NTCAN_GPIO_CFG_IRQ_NONE)
- Indicate a change on the rising edge (NTCAN_GPIO_CFG_IRQ_RISING_EDGE)
- Indicate a change on the falling edge (NTCAN_GPIO_CFG_IRQ_FALLING_EDGE)
- Indicate a change on both edges (Logical OR of the previous values)

input_filter

Input glitch filter which suppresses input signals with a pulse width of less than the given value.

This value is global for all channels and hardware dependent !!

For the esdACC the minimum pulse width is determined by the number of "input_filter" clock cycles. The initial value of 160 therefore corresponds to a minimum pulse width of 2µs at 80MHz clock.

Remarks:

Please refer to the hardware manual which option is supported by the GPIO circuity.

6.2.26 NTCAN_INFO

The NTCAN_INFO structure contains comprehensive information about the driver and the device environment. The NTCAN_INFO structure is initialized by ***canIoctl()*** if called with the command NTCAN_IOCTL_GET_INFO. Most of the information can also be obtained with consecutive calls of ***canStatus()*** and ***canIoctl()*** with other commands but this data simplifies the handling by combining everything at a single place.

Syntax:

```
typedef struct
{
  uint16_t hardware;           /* Hardware version                    */
  uint16_t firmware;           /* Firmware / FPGA version (0 = N/A)   */
  uint16_t driver;             /* Driver version                      */
  uint16_t dll;                /* NTCAN library version               */
  uint32_t features;           /* Device/driver capability flags      */
  uint32_t serial;             /* Serial # (0 = N/A)                  */
  uint64_t timestamp_freq;     /* Timestamp frequency (in Hz,1=N/A)   */
  uint32_t ctrl_clock;         /* CAN controller frequency (in Hz)    */
  uint8_t  ctrl_type;          /* Controller type (NTCAN_CANCTL_XXX)  */
  uint8_t  base_net;           /* Base net number                     */
  uint8_t  ports;              /* Number of physical ports            */
  uint8_t  transceiver;        /* Transceiver type (NTCAN_TRX_XXX)    */
  uint16_t boardstatus;        /* Hardware status                     */
  uint16_t firmware2;          /* Second firmware version (0 = N/A)   */
  char     boardid[32];        /* Board ID string                     */
  char     serial_string[16];  /* Serial # as string                  */
  char     drv_build_info[64]; /* Build info of driver                */
  char     lib_build_info[64]; /* Build info of library               */
  uint16_t open_handle;        /* Number of open handle               */
  uint8_t  ports_lin;          /* Number of physical LIN ports        */
  uint8_t  reserved;           /* Reserved for future use             */
  uint64_t sw_timestamp_freq;  /* SW timestamp resolution (in Hz)     */
  char     order_number[12];   /* Order number as string              */
  uint16_t gpio_ver;           /* GPIO: Version of core               */
  uint16_t gpio_cfg;           /* GPIO: Supported config options      */
  uint8_t  gpio_cnt;           /* GPIO: Number of physical channels   */
} NTCAN_INFO;
```

Members:

hardware
The hardware revision. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

firmware
The firmware version. Returned as 0 on passive CAN interfaces. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

driver
The driver version. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

dll
The NTCAN-API library version. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

features
This member is a bit mask with hardware and/or device driver specific capabilities which should be evaluated by an application to check if a certain feature is supported. Refer to table 22 for a description of the feature flags.

Please note that the bitmask returned here is a 32-bit value in comparison to the bitmask returned via `CAN_IF_STATUS` which is just a 16-bit value. Current and future feature flags defined as bit 16..31 are just returned here.

serial
The hardware serial number of the CAN board. If the hardware does not suppport returning a serial number this value is 0. Refer to the description of `NTCAN_IOCTL_GET_SERIAL` for ***canIoctl()*** how the 32-bit numerical value is converted into a serial string.

timestamp_freq
The resolution of the timestamp counter in Hz. If no timestamp support is available a value of 1 Hz is returned (to prevent division by zero errors).

ctrl_clock
The clock frequency of the CAN controller in Hz.

ctrl_type
CAN controller type according to table 21.

base_net
The configured base net number for this CAN port.

ports
Number of physical CAN and LIN ports made available by the hardware.

transceiver
A value defined in **<ntcan.h>** for the manufacturer and type of the CAN transceiver according to the table below.

Constant	CAN Transceiver
`NTCAN_TRX_PCA82C251`	NXP PCA82C251
`NTCAN_TRX_SN65HVD251`	TI SN65HVD251
`NTCAN_TRX_SN65HVD255`	TI SN65HVD255
`NTCAN_TRX_MCP2561FD`	Microchip MCP2561FD
`NTCAN_TRX_TCAN1051G`	TI TCAN1051G
`NTCAN_TRX_TCAN1051G`	TI SN65HVD230
`NTCAN_TRX_TJA1462`	NXP TJA1462 (SIC transceiver)

Table 29: CAN Transceiver Types

boardstatus
Reflects device specific errors or problems detected during hardware initialization.

firmware2
If the hardware contains a 2nd firmware (e.g. the IRIG B implementation together with an esdACC) this version is returned here, otherwise 0. Refer to the remarks at the end of this abstract for the encoding of the 16-Bit version number.

boardid
The device description as zero terminated ASCII string.

serial_string
The serial number as a zero terminated ASCII string.

In rare cases the serial number can not be coded as 32-bit value returned as member *serial* because of coding scheme limitations. In this case the string is the only source for the serial number.

drv_build_info
The driver build info string with compiler (version) and build time as a zero terminated ASCII string.

lib_build_info
The library build info string with compiler (version) and build time as a zero terminated ASCII string.

open_handle
The number of open handles. If this value is 0 the device driver does not (yet) return this information. A value of 65535 means that 65535 or more handles are open.

ports_lin
Number of physical LIN ports made available by the hardware.

sw_timestamp_frequency
The frequency of the driver internal high resolution (software) timestamp.

order_number
Product order number as a zero terminated ASCII string.

gpio_ver
Version of the GPIO core implementation in the ESDACC..

gpio_cfg
A bitmask which defines which aspects of the GPIO ports are configurable according to the flags defined the table below.

Constant	Description
NTCAN_GPIO_CFG_DIR	Configuration of I/O direction.
NTCAN_GPIO_CFG_VOLTAGE	Configuration of I/O voltage
NTCAN_GPIO_CFG_PULL	Configuration of I/O pull mode
NTCAN_GPIO_CFG_IRQ	Configuration of (input) IRQ mode
NTCAN_GPIO_CFG_FILTER	Configuration of (input) filter.

Table 30: GPIO Port Configuration Options

gpio_cnt
Number of physical GPIO channels supported by the hardware.

Remarks:

The members which contain a version are composed of major version (4 bit), minor version (4 bit) and a revision (8 bit).

Bit 12..15	Bit 8..11	Bit 0..7
Major	Minor	Revision

Example: The version 1.2.3 is represented as 0x1203.

6.2.27 NTCAN_TDC_CFG

The `NTCAN_TDC_CFG` structure is part of the type `NTCAN_BAUDRATE_X` to get/set parameters of the TDC mechanism which affect the SSP position and the TDC mechanism.

Syntax:

```
typedef struct {
    uint8_t tdc_mode;              /* TDC Mode                          */
    uint8_t ssp_offset;            /* SSP Offset                        */
    int8_t  ssp_shift;             /* SSP Shift                         */
    uint8_t tdc_filter;            /* TDC Filter                        */
} NTCAN_TDC_CFG;
```

Members:

tdc_mode
The TDC mode.

- **NTCAN_TDC_MODE_AUTO**: TDC automatic mode (See chapter 3.15.2.1).
- **NTCAN_TDC_MODE_MANUAL**: TDC Manual Mode (See chapter 3.15.2.2).
- **NTCAN_TDC_MODE_OFF**: TDC is disabled.

ssp_offset
The (positive) SSP offset in the mode **NTCAN_TDC_MODE_MANUAL** in mtq.

ssp_shift
The (positive or negative) SSP shift in the mode **NTCAN_TDC_MODE_AUTO** in mtq.

tdc_filter
The TDC filter in mtq (Ignored if unsupported by the CAN controller).

The minimum value which can be configured for the ESDACC if TDC is enabled is 2 mtq. Any smaller value is set to 2mtq.

7. Return Codes

All NTCAN-API functions return a status starting with the prefix 'NTCAN_' which should always be be evaluated by the application. 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 **<ntcan.h>**. For cross-platform portability an application should refer only to these constants, 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 on 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 severity categories *Successful*, *Warning* and *Error*. Furthermore a description of the error reason, a possible solution as well as the NTCAN-API functions which might return this result are part of the description.
Status codes which are listed in the header file but are not described here, are not returned by the driver any more and have only not been removed to ensure the compatibility of existing source code.

7.1 General Return Codes

NTCAN_SUCCESS

No error.

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_ERR_PASSIVE

Transmission error.

Category	Error/Warning
Cause	The CAN controller has changed into the state *Error Passive* during a blocking transmit operation, because the REC or TEC exceeded a value of 128.
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(), canWriteT()***, ***canWriteX(), canSend()***, ***canSendT()***, ***canSendX()***

NTCAN_CONTR_OFF_BUS

Internally bus-state triggered cancellation of a transmit operation.

Category	Error
Cause	The CAN controller has changed into *Off Bus* state 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(), canWriteT()***, ***canWriteX(), canSend()***, ***canSendT()***, ***canSendX()***

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(), canWriteT()***, ***canWriteX(), canSend()***, ***canSendT()***, ***canSendX()***

NTCAN_ERROR_NO_BAUDRATE

Transmission error.

Category	Error
Cause	A CAN message could not be transmitted because the bit rate of the CAN controller is not set or a CAN FD message is requested to be sent and no data phase bit is configured.
Solution	Configure the bit rate with ***canSetBaudrate()***.
Function	***canWrite(), canWriteT()***, ***canWriteX(), canSend()***, ***canSendT()***, ***canSendX()***

NTCAN_ERROR_LOM

Transmission error.

Category	Error
Cause	A CAN message could not be transmitted because the CAN controller is configured in *listen-only* mode which prevents sending CAN messages.
Solution	Disable the *listen-only* mode with ***canSetBaudrate()***.
Function	***canWrite()***, ***canWriteT()***, ***canWriteX(), canSend()***, ***canSendT()***, ***canSendX()***

NTCAN_HANDLE_FORCED_CLOSE

Abortion of a blocking transmit/receive operation.

Category	Warning/Error
Cause	A blocking transmit or receive operation was canceled, 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()***, ***canReadX()***, ***canWrite()***, ***canWriteT()***, ***canWriteX()***

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 29-bit ID is activated, all other 29-bit 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 29-bit ID is deactivated, all other 29-bit 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 with 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	A function to return the result of an asynchronous (overlapped) I/O request was called with FALSE for parameter *bWait* before the operation has not been completed. Refer to the Win32 platform SDK help for mpre information about overlapped I/O.
Solution	See Win32 platform SDK help about *GetOverlappedResult()*.
Function	***canGetOverlappedResult(), canGetOverlappedResultT(), canGetOverlappedResultX()***

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()***, ***canReadX(), canWrite()***, ***canWriteT(), canWriteX() canSendX()***

NTCAN_NET_NOT_FOUND

CAN device not found.

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

NTCAN_NO_CAN_CAPABILITY

The physical port has no CAN capability.

Category	Error
Cause	The physical network referenced by the logical net number has no CAN capability (e.g. is a LIN network).
Solution	Do not use this network with the NTCAN-API but with the API for the respective bus physics (e.g. NTLIN-API).
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(), canReadX()***, ***canTake()***, ***canTakeT(), canTakeX()***

NTCAN_NO_LIN_CAPABILITY

The physical port has no LIN capability.

Category	Error
Cause	The physical network referenced by the logical net number has no LIN capability (e.g. is a CAN network).
Solution	Do not use this network with the NTLIN-API but with the API for the respective bus physics (e.g. NTCAN-API).
Function	***canOpen()***

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 this feature. ➢ If the requested feature can not be supported due to hardware constraints, you might have to use a different **esd electronics** CAN board. ➢ Refer to the chapter 3.18 to check, which features are supported. **esd electronics** is always aimed to provide the maximum of features for the CAN hardware / operating system combination. Please visit www.esd.eu to check for the latest software version.
Function	***canIoctl()***, ***canIdAdd()***, ***canIdDelete()***, ***canSetBaudrate()***

NTCAN_OPERATION_ABORTED

Explicitly triggered cancellation 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()***, ***canReadX()***, ***canWrite()***, ***canWriteT()***, ***canWriteX()***

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()***, ***canReadX()***, ***canTake()***, ***canTakeT()***, ***canTakeX()***

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(), canWriteT()***,***canWriteX()***

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 nodes on the expected CAN-IDs.
Function	***canRead(), canReadT(),canReadX()***

NTCAN_TX_ERROR

Internally triggered cancellation of a blocking transmit operation.

Category	Error
Cause	The message transmission was canceled by the CAN controller and/or CAN driver for the following reasons: • An internal Tx message watchdog expired. • During the transmission of a CAN Remote Frame the Data Frame with this CAN-ID was transmitted at the very same time so the RTR frame was never “visible” on the bus. • The CAN message was sent in DAR mode and the transmission failed due to errors or a lost arbitration procedure.
Solution	If the result was not expected (e.g. in DAR mode) 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(), canWriteT(),canWriteX()***

NTCAN_TX_TIMEOUT

Timeout triggered cancellation of a blocking transmit operation.

Category	Error
Cause	The transmission was canceled because the Tx timeout configured via ***canOpen()*** or via ***canIoctl()*** with the command `NTCAN_IOCTL_SET_TX_TIMEOUT`.was exceeded
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(), canWriteT(),canWriteX()***

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

7.2 Specific 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()***

8. Example C Source

This chapter contains complete code example receiving and transmitting CAN messages with the NTCAN-API.

8.1 Receiving messages (CAN CC /FIFO Mode)

```
#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_fifo(void)
{
    int          net = 0;                     /* Logical net number (here: 0)     */
    uint32_t     mode = 0;                    /* Mode bits for canOpen            */
                                              /* -> Default (0): FIFO mode        */
    int32_t      txqueuesize = NTCAN_NO_QUEUE; /* No Tx queue required            */
    int32_t      rxqueuesize = 8;             /* Maximum number of Rx messages    */
    int32_t      txtimeout = 0;               /* No Tx timeout required           */
    int32_t      rxtimeout = 10000;           /* Rx timeout in ms                 */
    NTCAN_HANDLE rxhandle;                    /* CAN handle returned by canOpen() */
    NTCAN_RESULT retvalue;                    /* Return values of NTCAN API calls */
    uint32_t     baud = NTCAN_BAUD_500;       /* Configured CAN baudrate          */
                                              /* -> 500 kBit/s                    */
    CMSG         cmsg[8];                     /* Buffer for can messages          */
    int          i,j;                         /* Loop counter                     */
    int32_t      count;                       /* # 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);
```

Example C Source

```
    }

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

    do {
        /*
         * Set max numbers of messages that should be returned with
         * a single canRead() call according to the availablre buffer size
         * every time you do the canRead() call.
         */
        count = (int32_t)(sizeof(cmsg) / sizeof(cmsg[0]));

        retvalue = canRead(rxhandle, &cmsg[0], &count, 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() received %d message(s) !\n", count);
            for (j = 0; j < (int)count; j++) {
                int len = NTCAN_LEN_TO_DATASIZE(cmsg[j].len);
                printf("CAN-ID of received message %d : %03x\n",
                       j, NTCAN_ID(cmsg[j].id));
                if (NTCAN_IS_RTR(cmsg[j].len)) {
                    printf("  Received a RTR message (%d)", len);
                } else {
                    printf("  Received a data message with %d bytes : ", len);
                    for (i = 0; i < len; i++) {
                        printf("%02x ", cmsg[j].data[i]);
                    }
                }
                printf("\n");
            }
        }

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

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

    return(0);
}
```

8.2 Receiving messages (CAN CC and CAN FD / FIFO Mode)

In the text box below is an example to receive CAN CC and/or CAN FD message. The differences to the example code to receive just CAN CC messages are marked (especially the use of the new data structures and functions which end on 'X' and the macros to get/set data of the CAN message length field which should be used for all CAN message variants).

```
#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 CC frames or CAN FD
 * frames with an identifier that has been previously enabled for this
 * handle.
 * Finally all proper cleanup operations are performed
 */
int example_rx_fd(void)
{
   int               net=42;             /* Logical net number (here: 42) */
   uint32_t          mode=NTCAN_MODE_FD; /* 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 */
   NTCAN_BAUDRATE_X baud;                /* Bit rate configuration */
   CMSG_X            cmsg[8];            /* Buffer for can messages */
   int               i;                  /* Loop counter */
   int32_t           len;                /* Size in # of messages for canReadX() */

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

   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");

   /* ############################### ################################ */
   baud.mode = NTCAN_BAUDRATE_MODE_INDEX;
   baud.flags = NTCAN_BAUDRATE_FLAG_FD;
   baud.arb.u.idx = NTCAN_BAUD_500;     /* Nominal bit rate: 500KBit/s   */
   baud.data.u.idx = NTCAN_BAUD_2000;   /* Data phase bit rate: 2 MBit/s */
   retvalue = canSetBaudrateX(rxhandle, &baud);

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

   printf("function canSetBaudrateX() 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);
```

Example C Source

```
        return(-1);
    }

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

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

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

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

        if (retvalue == NTCAN_RX_TIMEOUT)
        {
            printf("canReadX() returned timeout\n");
            continue;
        }
        else if(retvalue != NTCAN_SUCCESS)
        {
            printf("canReadX() failed with error %d!\n", retvalue);
        }
        else
        {
            printf("Function canReadX() returned OK !\n");
            printf("ID of received message :%x!\n", cmsg[0].id);
            printf("DLC of received message :%x!\n", NTCAN_DLC(cmsg[0].len));
            if(NTCAN_IS_FD(cmsg[0].len)) {
                printf("BRS of received message :%x!\n", !
                       NTCAN_IS_FD_WITHOUT_BRS(cmsg[0].len));
            } else {
                printf("RTR of received message :%x!\n", NTCAN_IS_RTR(cmsg[0].len));
            }
            for (i=0;i<NTCAN_LEN_TO_DATASIZE(cmsg[0].len);i++)
                printf("Byte %d of received message :%x!\n", i, cmsg[0].data[i]);
        }

        break;
    };

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

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

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

    return(0);
}
```

8.3 Receiving Messages (CAN CC / Object Mode)

In the text box below is an example to receive CAN CC messages using the Rx Object Mode.

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

/*
 * This example demonstrates how the NTCAN-API can be used to open a handle
 * in object mode, set a baudrate and return the latest CAN CC
 * messages which have been received for the given ID set. Finally all proper
 * cleanup operations are performed.
 */
int example_rx_obj(void)
{
    int          net = 0;                       /* Logical net number (here: 0) */
    uint32_t     mode = NTCAN_MODE_OBJECT;      /* Mode bits for canOpen()*/
                                                /*    → Object Mode)      */
    int32_t      txqueuesize = NTCAN_NO_QUEUE; /* No Tx queuesize required */
    int32_t      rxqueuesize = 8;               /* Maximum number of Rx messages */
    int32_t      txtimeout = 0;                 /* No Tx timeout required */
    int32_t      rxtimeout = 0;                 /* No Rx timeout required */
                                                /* as data is polled      */
    NTCAN_HANDLE rxhandle;                      /* CAN handle returned by canOpen() */
    NTCAN_RESULT retvalue;                      /* Return values of NTCAN API calls */
    uint32_t     baud = NTCAN_BAUD_500;         /* Configured CAN baudrate */
                                                /*   -> 500 kBit/s         */
    CMSG         cmsg[8];                       /* Buffer for CAN messages */
    int          i, j;                          /* Loop counter            */
    int32_t      count;                         /* # of messages for canTake() */
    const int    polled_messages = 3;           /* Number of polled messages   */

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

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

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

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

    /* Enable all CAN-IDs we want to poll in the acceptance filter */
    retvalue  = canIdAdd(rxhandle, 100); /* Enable CAN-ID 100 */
    retvalue |= canIdAdd(rxhandle, 200); /* Enable CAN-ID 200 */
    retvalue |= canIdAdd(rxhandle, 300); /* Enable CAN-ID 300 */

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

Example C Source

```
    }
    /* ################################################################ */

    cmsg[0].id = 100;   /* Prepare to receive CAN-ID 100 in object mode */
    cmsg[1].id = 200;   /* Prepare to receive CAN-ID 200 in object mode */
    cmsg[2].id = 300;   /* Prepare to receive CAN-ID 300 in object mode */

    do {
        /*
         * Set number of messages that should be returned with canTake()
         * The requires
         */
        count = polled_messages;

        retvalue = canTake(rxhandle, &cmsg[0], &count);

        if (retvalue != NTCAN_SUCCESS) {
            printf("Error: canTake() failed with error %d!\n", retvalue);
        } else if (count != polled_messages){
            printf("Error: canTake() returned %d messages instead of %d ?!?\n",
                count, polled_messages);
        } else {
            for (i = 0; i < polled_messages; i++)
            {
                printf("ID: %3d ", cmsg[i].id);
                if (cmsg[i].len & NTCAN_NO_DATA) {
                    printf("-> No data received yet for this CAN-ID\n");
                }
                else
                {
                    if (NTCAN_IS_RTR(cmsg[i].len))
                    {
                        printf("-> R (%d)\n", NTCAN_DLC(cmsg[i].len));
                    }
                    else
                    {
                        uint8_t len = NTCAN_LEN_TO_DATASIZE(cmsg[i].len);

                        printf("-> D (%d) Data: ", NTCAN_DLC(cmsg[i].len));
                        for (j = 0; j < len; j++)
                            printf("%02x ", cmsg[i].data[j]);
                        printf("\n");
                    }
                }
            }
        }

        /*
         * Note: If you remove the break below to repeat execution replace it
         *       with a delay operation to prevent a high CPU load.
         */
        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);
    else
        printf("canClose() returned OK !\n");

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

    return(0);
}
```

8.4 Transmitting messages (CAN CC)

```
#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
```

Example C Source

```
        printf("function canWrite() returned OK !\n");

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

    retvalue = canClose (txhandle);

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

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

    return(0);
}
```

8.5 Transmitting messages (CAN FD)

In the text box below is an example to transmit a CAN FD message. The differences to the example code to transmit a CAN CC messages are marked (especially the use of the new data structures and functions which end on 'X').

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

/*
 * This example demonstrates how the NTCAN-API can be used to open a handle,
 * configure a nominal and data phase bit rate and transmit a CAN FD frame
 * Finally all proper cleanup operations are performed
 */
int example_tx_fd(void)
{
   int               net=42;             /* logical net number (here: 42) */
   uint32_t          mode=NTCAN_MODE_FD; /* 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=1000;     /* timeout for receiving data in ms */
   NTCAN_HANDLE      txhandle;           /* can handle returned by canOpen() */
   NTCAN_RESULT      retvalue;           /* return values of NTCAN API calls */
   NTCAN_BAUDRATE_X  baud;               /* Bit rate configuration */
   CMSG_X            cmsg[8];            /* can message buffer */
   int               no_brs=0;           /* No bit rate switch 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");

   /* ################################################################ */
   baud.mode = NTCAN_BAUDRATE_MODE_INDEX;
   baud.flags = NTCAN_BAUDRATE_FLAG_FD;
   baud.arb.u.idx = NTCAN_BAUD_500;     /* Nominal bit rate: 500KBit/s   */
   baud.data.u.idx = NTCAN_BAUD_2000;   /* Data phase bit rate: 2 MBit/s */
   retvalue = canSetBaudrateX(txhandle, &baud);

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

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

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

   /*
   * Initialize the first message in buffer to CAN id = 0, len = 12
   * and data0-data12 = 0..11
   */
   cmsg[0].id  = 0x00;
   cmsg[0].len = NTCAN_DATASIZE_TO_DLC(12);
   cmsg[0].len |= (NTCAN_FD | (no_brs << 4));
```

Example C Source

```
    for (i = 0; i < 12; i++)
        cmsg[0].data[i] = (uint8_t)i;

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

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

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

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

    retvalue = canClose (txhandle);

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

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

    return(0);
}
```

8.6 Timestamped TX messages (CAN CC)

In the text box below is an example to transmit timestamped TX CAN CC message. The transmission is triggered with the non-blocking ***canSendT()***. The main difference to the immediate transmission of messages are marked **bold**.

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

/*
 * This example demonstrates how the NTCAN-API can be used to open a handle,
 * set a CAN CC bitrate and transmit CAN CC messages at a certain
 * point of time in the future using the Timestamped TX feature with a
 * non-blocking request. The example code acquires the current timestamp and
 * starts transmission of 10 frames after 1000 ms with a time difference of 100
 * ms between each frame. Finally all proper cleanup operations are performed
*/
void timestamped_tx(uint16_t can_id)
{

    NTCAN_HANDLE m_hCan;
    uint64_t timestampFreq, timestamp;
    CMSG_T msgT[10];
    NTCAN_RESULT rc;
    int32_t len;
    int i;

    /* Open CAN handle for net 42 */
    rc = canOpen(42, NTCAN_MODE_TIMESTAMPED_TX, 100, 100, 1000, 1000, &m_hCan);
    if (rc != NTCAN_SUCCESS) {
        printf("Opening handle failed with %d\n", rc);
        return;
    }

    /* Request timestamp/tick frequency of interface */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_GET_TIMESTAMP_FREQ, &timestampFreq);
    if (rc != NTCAN_SUCCESS) {
        printf("Gathering timestamp frequency failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    /* Set baudrate to 500KBit/s */
    rc = canSetBaudrate(m_hCan, NTCAN_BAUD_500);
    if (rc != NTCAN_SUCCESS) {
        printf("Configuration CAN bit rate failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    /* Request timestamp/tick frequency of interface */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_GET_TIMESTAMP, &timestamp);
    if (rc != NTCAN_SUCCESS) {
        printf("Gathering timestamp failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    // Start transmission in one second from now
    timestamp += timestampFreq;

    /*
     * Setup the Tx object with the given CAN-ID and initialize the message.
     */
    memset(msgT, 0, sizeof(msgT));
    len = sizeof(msgT) / sizeof(*msgT);
    for (i = 0; i < len; i++) {
        msgT[i].id = (int32_t)(can_id + i);
        msgT[i].len = NTCAN_DATASIZE_TO_DLC(8);
        msgT[i].timestamp = timestamp;
        strcpy((char *)msgT[i].data, "Hello !!");
        timestamp += timestampFreq / 10;    /* Next transmission in 100 ms */
    }
```

Example C Source

```
    /*
     * Non-blocking call to schedule message transmission.
     */
    rc = canSendT(m_hCan, msgT, &len);
    if (rc != NTCAN_SUCCESS) {
        printf("canSendT() failed with %d\n", rc);
    }

    /*
     * NOTE: All frames which are not transmitted before the handle is closed
     *       will be aborted !!!
     */
    SLEEP(3000);   /* OS specific delay for 3 seconds !!! */

    (void)canClose(m_hCan);
    return;
}
```

8.7 Timestamped TX messages (CAN FD)

In the text box below is an example to transmit timestamped TX CAN FD message. The differences to the example code to transmit timestamped TX CAN CC messages are marked (especially the use of the new data structures and functions which end on 'X'). The transmission is triggered with the blocking ***canWriteT()***. The main difference to the immediate transmission of messages are marked **bold**. Please note that the Tx timeout in this example is increased to 5000 ms to prevent that the scheduled transmission request is aborted because the blocking request returns with timeout beforehand.

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

/*
 * This example demonstrates how the NTCAN-API can be used to open a handle,
 * set a CAN FD bitrate and transmit CAN FD messages at a certain
 * point of time in the future using the Timestamped TX feature with a
 * blocking request. The example code acquires the current timestamp and
 * starts transmission of 10 frames after 1000 ms with a time difference of 100
 * ms between each frame. Finally all proper cleanup operations are performed
*/
void timestamped_tx_fd(uint16_t can_id)
{

    NTCAN_HANDLE m_hCan;
    NTCAN_BAUDRATE_X baudX;
    uint64_t timestampFreq, timestamp;
    CMSG_X msgX[10];
    NTCAN_RESULT rc;
    int32_t len;
    int i;

    /* Open CAN handle for net 42 */
    rc = canOpen(42, NTCAN_MODE_FD | NTCAN_MODE_TIMESTAMPED_TX, 100, 100,
                 5000, 5000, &m_hCan);
    if (rc != NTCAN_SUCCESS) {
        printf("Opening handle failed with %d\n", rc);
        return;
    }

    /* Request timestamp/tick frequency of interface */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_GET_TIMESTAMP_FREQ, &timestampFreq);
    if (rc != NTCAN_SUCCESS) {
        printf("Gathering timestamp frequency failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    /* Set baudrate to 500KBit/s / 2MBit/s */
    baudX.mode = NTCAN_BAUDRATE_MODE_INDEX;
    baudX.flags = NTCAN_BAUDRATE_FLAG_FD;
    baudX.arb.u.idx = NTCAN_BAUD_500;
    baudX.data.u.idx = NTCAN_BAUD_2000;
    baudX.reserved = 0;
    rc = canSetBaudrateX(m_hCan, &baudX);
    if (rc != NTCAN_SUCCESS) {
        printf("Configuration CAN bit rate failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    /* Request timestamp/tick frequency of interface */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_GET_TIMESTAMP, &timestamp);
    if (rc != NTCAN_SUCCESS) {
        printf("Gathering timestamp failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    // Start transmission in one second from now
    timestamp += timestampFreq;
```

Example C Source

```
    /*
     * Setup the Tx object with the given CAN-ID and initialize the message.
     */
    memset(msgX, 0, sizeof(msgX));
    len = sizeof(msgX) / sizeof(*msgX);
    for (i = 0; i < len; i++) {
        msgX[i].id = (int32_t)(can_id + i);
        msgX[i].len = NTCAN_DATASIZE_TO_DLC(64);
        msgX[i].len |= NTCAN_FD;
        msgX[i].timestamp = timestamp;
        strcpy((char *)msgX[i].data, "Hello world!");
        timestamp += timestampFreq / 10;   /* Next transmission in 100 ms */
    }

    /*
     * Blocking call to schedule message transmission.
     * NOTE: All scheduled frames which transmission time exceed the configured
     *       Tx timeout will be aborted before return !!!
     */
    rc = canWriteX(m_hCan, msgX, &len, NULL);
    if (rc != NTCAN_SUCCESS) {
        printf("canWriteX() failed with %d\n", rc);
    }

    (void)canClose(m_hCan);
    return;
}
```

8.8 Scheduling messages (CAN CC)

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

/*
* This example demonstrates how the NTCAN-API can be used to open a handle,
* set a baudrate and define a CAN message which is transmitted autonomously
* in background with an incrementing, rotating counter.
* Finally all proper cleanup operations are performed
*/
/*
 * Schedule a CAN message
 */
void sched_test(uint16_t can_id, uint32_t time_interval_ms)
{
    NTCAN_HANDLE m_hCan;
    uint64_t timestampFreq;
    CMSG msg;
    CSCHED schedule;
    NTCAN_RESULT rc;

    /* Open CAN handle for net 42 */
    rc = canOpen(42, 0, 10, 10, 1000, 1000, &m_hCan);
    if (rc != NTCAN_SUCCESS) {
        return;
    }

    /* Request timestamp/tick frequency of interface */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_GET_TIMESTAMP_FREQ, &timestampFreq);
    if (rc != NTCAN_SUCCESS) {
        printf("Gathering timestamp frequency failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    /* Set baudrate to 1 MBit/s */
    rc = canSetBaudrate(m_hCan, NTCAN_BAUD_1000);
    if (rc != NTCAN_SUCCESS) {
        printf("Configuration CAN bit rate failed with %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    /*
     * Setup the Tx object with the given CAN-ID and initialize the message.
     */
    memset(&msg, 0, sizeof(CMSG));
    msg.id = (uint32_t)can_id;
    msg.len = 8;
    strcpy((char *)msg.data, "Hello");

    /*
     * Define a scheduling set for the given CAN-ID with a 16 bit
     * counter (little endian) at byte offset 6 which is incremented
     * with each transmission and counts from 0 up to 5 before it
     * start again with 0 with the given scheduling interval converted
     * from milliseconds into ticks.
     */
    memset(&schedule, 0, sizeof(CSCHED));
    schedule.id = (uint32_t)can_id;
    schedule.flags = NTCAN_SCHED_FLAG_EN | NTCAN_SCHED_FLAG_INC16 |
                     NTCAN_SCHED_FLAG_OFS6;
    schedule.time_start = 0;
    schedule.time_interval = ((timestampFreq * time_interval_ms) / 1000ULL);
    schedule.count_start = 0x0;
    schedule.count_stop = 0x5;

    /* Create a Tx object */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_TX_OBJ_CREATE, &msg);
    if (rc != NTCAN_SUCCESS) {
        printf("Creation of Tx object failed with error %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }
```

Example C Source

```
    /* Configure the scheduling for the Tx object */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_TX_OBJ_SCHEDULE, &schedule);
    if (rc != NTCAN_SUCCESS) {
        printf("Configuration of scheduling failed with error %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    /* Start the scheduling set */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_TX_OBJ_SCHEDULE_START, NULL);
    if (rc != NTCAN_SUCCESS) {
        printf("Start of scheduling failed with error %d\n", rc);
        (void)canClose(m_hCan);
        return;
    }

    SLEEP(5000);   /* OS specific delay for 5 seconds !!! */

    /* Update the Tx message */
    strcpy((char *)msg.data, "World");
    rc = canIoctl(m_hCan, NTCAN_IOCTL_TX_OBJ_UPDATE, &msg);
    if (rc != NTCAN_SUCCESS) {
        printf("Update of scheduling failed with error %d\n", rc);
    }

    SLEEP(5000);   /* OS specific delay for 5 seconds !!! */


    /* Stop the scheduling */
    rc = canIoctl(m_hCan, NTCAN_IOCTL_TX_OBJ_SCHEDULE_STOP, NULL);
    if (rc != NTCAN_SUCCESS) {
        printf("Stop of scheduling failed with error %d\n", rc);
    }

    (void)canClose(m_hCan);
}
```

9. CLI Application *canTest*

An NTCAN implementation is shipped with the demo application *canTest* written in ANSI-C. This console application is deployed as source code and as an executable for the target system and is written as cross-platform application so it can be compiled and executed on all supported platforms. The program is intended to

- Demonstrate the use of the various NTCAN API calls and data structures described in the previous chapters
- Perform basic functional test on the CAN bus for CAN Classic as well as CAN FD.
- Gather information about the hardware and software environment in case you have to report a problem to **esd electronics**.

The binary version of the test program is usually distributed as '**cantest***'* (all lowercase letters). An exception of this rule is the VxWorks release where the program has to be called with '**canTest**' as an allusion to the prevalent naming convention of this platform.

If *canTest* is called without parameters, the program lists the CAN ports of all configured **esd electronics** CAN interfaces in the host system together with information about the CAN hardware and software environment as listed in the table below followed by the command line syntax. For an overview without the syntax use -2 as test number (1st parameter).

Information	Description
ID	CAN module identifier name.
Dll	Revision number of the NTCAN library in the format *major.minor.revision*.
Driver	Revision number of the CAN driver in the format *major.minor.revision*.
Firmware	Revision number of local the firmwares (if applicable) in the format *major.minor.revision*. For passive CAN boards 0.0.0 is returned. For active CAN boards with one FW 0.0.0 is returned for the second one.
S/N	Hardware serial number of the CAN board. Returns N/A if not supported by the CAN board.
Hardware	Hardware revision number in the format *major.minor.revision*.
Baudrate	Configured bitrate as described for ***canSetBaudrate()***.or ***canSetBaudrateX().***
Status	Hardware status of the CAN module (if supported by the module).
Features	Supported properties of hardware and/or device driver. Returns the (16 bit) value of the parameter *feature* of the data structure `CAN_IF_STATUS` or the (32-bit) value *features* of the data structure `NTCAN_INFO` if supported by the device driver. To get a textual description of the features call *canTest* with -3 as test number (1st parameter)
Controller	CAN controller type and clock. If the device supports returning the data structure `NTCAN_CTRL_STATE` the actual CAN controller state and the receive and transmit error counter are shown, too.
Transceiver	The transceiver type.
Timestamp	Current timestamp value captured at the moment *canTest* is called (if supported by the CAN board).
TimestampFreq	Frequency of the timestamp counter (if supported by the CAN board).

Table 31: CAN board information listed with *canTest*

The following figure shows an example console output of *canTest* on Windows with one active board (CAN-PCIe/402-2) and the virtual CAN driver.

```
CAN Test FD Rev 3.1.10 (r17609)  -- (c) 1997 - 2023 esd electronics gmbh

Available CAN-Devices:
Net   0: ID=CAN_PCIE402 (2/0 CAN/LIN ports) S/N: GP001890
         Versions (hex): Lib=5.0.06 Drv=4.2.00 HW=1.0.16 FW=0.0.49 (0.0.00)
         Baudrate=00000000 (1000 KBit/s) Status=0000 Features=001c8ffa
         Ctrl=esd Advanced CAN Core @ 80 MHz Transceiver=TI SN65HVD265
         Bus state (Error Active / REC:0 / TEC:0)
         TimestampFreq=80.000000 / 10.000000 MHz Timestamp=00009721AADA5CD8
Net   1: ID=CAN_PCIE402 (2/0 CAN/LIN ports) S/N: GP001890
         Versions (hex): Lib=5.0.06 Drv=4.2.00 HW=1.0.16 FW=0.0.49 (0.0.00)
         Baudrate=7fffffff (Not set) Status=0000 Features=001c8ffa
         Ctrl=esd Advanced CAN Core @ 80 MHz Transceiver=TI SN65HVD265
         Bus state (Error Active / REC:0 / TEC:0)
         TimestampFreq=80.000000 / 10.000000 MHz Timestamp=00009721AADFBD5D
Net  42: ID=CAN_VIRTUAL (1/0 CAN/LIN ports) S/N: N/A
         Versions (hex): Lib=5.0.06 Drv=4.0.03 HW=1.0.00 FW=0.0.00 (0.0.00)
         Baudrate=7fffffff (Not set) Status=0000 Features=00028f7a
         Ctrl=esd Advanced CAN Core @ 80 MHz Transceiver=NXP PCA82C251
         Bus state (Error Active / REC:0 / TEC:0)
         TimestampFreq=10.000000 / 0.000000 MHz Timestamp=000012E4518E90DC

Syntax: cantest test-Nr [net id-1st id-last count
        txbuf rxbuf txtout rxtout baud[:dbaud:[tdc]] testcount data0
data1 ...]
Test   0:  canSend()
Test  20:  canSendT()
Test  50:  canSend() with incrementing ids
Test  60:  canSendX()
Test  90:  canSend() after using NTCAN_IOCTL_GET_TX_MSG_COUNT
Test   1:  canWrite()
Test  21:  canWriteT()
Test  51:  canWrite() with incrementing ids
Test  61:  canWriteX()
Test   2:  canTake()
Test  12:  canTake() with time-measurement for 10000 can-frames
Test  22:  canTakeT()
Test  32:  canTake() in Object-Mode
Test  42:  canTakeT() in Object-Mode
Test  62:  canTakeX()
Test  72:  canTakeX() with time-measurement for 10000 can-frames
Test  82:  canTakeX() in Object-Mode
Test  92:  canTake() after using NTCAN_IOCTL_GET_RX_MSG_COUNT
Test   3:  canRead()
Test  13:  canRead() with time-measurement for 10000 can-frames
Test  23:  canReadT()
Test  63:  canReadX()
Test  73:  canReadX() with time-measurement for 10000 can-frames
Test   4:  canReadEvent()
Test  64:  Retrieve bus statistics (every tx timeout)
Test  74:  Reset bus statistics
Test  84:  Retrieve bitrate details (every tx timeout)
Test   5:  canSendEvent()
Test   6:  Overlapped-canRead()
Test  16:  Overlapped-canReadT()
Test  66:  Overlapped-canReadX()
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
Test 100:  Object Scheduling test
Test 110:  Object Scheduling test with cmsg_x
Test  -2:  Overview without syntax help
Test  -3:  Overview without syntax help but with feature flags details
Test  -4:  Overview without syntax help but with bit rate index table
```

If called without any parameter the list of CAN ports, the parameters of *canTest* and a list of available test cases are shown. Each parameter has a default value, which depends on the test case and is used if the parameter is not set. The parameters always have to be set in the order as they are displayed. This means if the value of *testcount* should be changed all previous parameters have to be set, too. If the parameters after *testcount* are not entered, they will be set to their default values for the test.

If the number format is not described explicitly decimal values, hexadecimal values and octal values are allowed in the common C/C++ notation.

- To write numbers in hexadecimal, precede the value with a 0x. Thus, 0x23 is the decimal value 35.
- To write numbers in octal, precede the value with a 0. Thus, 023 is the decimal value 19.

The table below describes the arguments of *canTest:*

Parameter	Description	Default
Test-Nr	Number of one of the supported tests as decimal value	N/A (mandatory)
Net	Logical net number to perform the test as decimal value	0
Id-first	Start of NTCAN-ID range. The format follows the description in chapter 3.4.	0
Id-Last	End of NTCAN-ID range. The format follows the description in chapter 3.4.	0
Count	The number of CAN messages which are to be transmitted or received with an CAN I/O. This parameter corresponds to the input parameter *len* of ***canWrite()***, ***canRead()***,... If *count* is set to a negative value the absolute value of count is used for the test and a receive test is configured with support to mark interaction frames. A transmit test for CAN CC is configured to send CAN RTR frames instead of data frames and a transmit CAN for CAN FD is configured to send the frames without a bit rate switch. If *count* is set to 0 a receive test is configured to discard interaction frames and the internal value of count is set back to the default value of 1. For transmit tests a value of 0 for *count* is not supported.	1
Txbuf	Size of NTCAN handle Tx queue as decimal value. This parameter corresponds to the parameter *txqueuesize* of ***canOpen()***.	10
Rxbuf	Size of NTCAN handle Rx queue. This parameter corresponds to the parameter r*xqueuesize* of ***canOpen()***.	100
Txtout	Timeout for blocking transmit tests in ms as decimal value. This parameter corresponds to the parameter *txtimeout* of ***canOpen()***. Timeout to wait after a non-blocking transmit tests in ms as decimal value.	1000
Rxtout	Timeout for blocking receive tests in ms as decimal value. This parameter corresponds to the parameter *rxtimeout* of ***canOpen()***.	5000

Parameter	Description	Default
Baud	32-Bit CAN bitrate parameter according to the description of ***canSetBaudrate()***. Instead of a numerical value the strings ***no*** to keep the configured bit rate, ***disable*** to leave the CAN bus and ***auto*** to start the automatic bit rate detection are supported, too. To get a list of the relation between index and bit rate call *canTest* with -4 as test number. The standard bit rates of this list can also be provided as strings with the mnemonic **250K** to define a bit rate of 250 Kbit/s. The data bit rate of a CAN FD test has to be separated from the nominal bit rate by a colon. To configure a nominal bit rate of 500 KBit/s and a data bit rate of 2 Mbit/s you can provide **2:17** or **500K:2M** as argument. Appending a decimal point or comma to the arbitration bit rate (or using a decimal point or comma as separator for a CAN FD bit rate) configures the controller to enable the listen-only respectively self-test mode.	CAN Classic 2 (= 500 Kbit/s) CAN FD 2:17 (=500 Kbit/s and 2 MBit/s)
Testcount	Number of times the test should be repeated as decimal value. A value of -1 repeats the test endlessly.	10 (Transmit) -1 (Receive)
Data0..Data8 Data0..Data64	CAN CC: Without configuration two 32-bit counter values (8 bytes) will be transmitted. If one to eight bytes are specified, the specified bytes will be transmitted. The parameter has no effect for receive tests. CAN FD: Without configuration two 32-bit counter values (8 bytes) will be transmitted. The remaining 56 bytes are filled up with six 32 bit values starting with the numerical value 2. If one to 64 bytes are specified, the specified bytes will be transmitted. The parameter has no effect for receive tests.	N/A

Table 32: Command line parameter of canTest

The table below describes the available tests. For every test the execution time is measured and in case of an error the error code is converted into a text with ***canFormatError()***.

Test #	Description
0	Use the non-blocking NTCAN-API ***canSend()*** to transmit CAN CC messages. The messages are transmitted with CAN-ID *id-first* and *id-last* is ignored. If *count* is set to a negative value the test will send RTR frames instead of data frames with the absolute value of *count.* The parameter *txtout* is used as the delay between consecutive loops if *testcount* is greater 1.
20	Use the non-blocking NTCAN-API ***canSendT()*** to transmit CAN CC messages. If the *Timestamped TX* feature is not supported the test is identical to test 0. Otherwise the test. Otherwise most test parameter are also identical to test 0 with the exception of the timeouts. The parameter *txtout* is used as base offset for the transmission in ms and the parameter r*xtout* as offset between consecutive frames. Example: If you send 10 frames (count = 10) with txtout set to 2000 ms and rxtout set to 10 ms the test will start transmitting 10 frames in 2000 ms from now with a delay of 10 ms between each frame.
50	Use the non-blocking NTCAN-API ***canSend()*** to transmit CAN CC messages. The messages are transmitted with increasing CAN-IDs from *id-first* to *id-last* with a wraparound if *id-last* is reached. If *count* is set to a negative value the test will send RTR frames instead of data frames with the absolute value of *count.* The parameter *txtout* is used as the delay between consecutive loops if *testcount* is greater 1.
60	Use the non-blocking NTCAN-API ***canSendX()*** to transmit CAN FD messages. The messages are transmitted with increasing CAN-IDs from *id-first* to *id-last* with a wraparound if *id-last* is reached. If *count* is set to a negative value the test will send frames without switching the bit rate during the data phase with the absolute value of *count.* The parameter *txtout* is used as the delay between consecutive loops if *testcount* is greater 1.
1	Use the blocking NTCAN-API ***canWrite()*** to transmit CAN messages. The messages are transmitted with CAN-ID *id-first* and *id-last* is ignored. If *count* is set to a negative value the test will send RTR frames instead of data frames with the absolute value of *count.* If *tx_timeout* is set to a negative value the messages will be transmitted in DAR mode if supported by the hardware.
51	Use the blocking NTCAN-API ***canWrite()*** to transmit CAN messages. The messages are transmitted with increasing CAN-IDs from *id-first* to *id-last* with a wraparound if *id-last* is reached. If *count* is set to a negative value the test will send RTR frames instead of data frames with the absolute value of *count.* If *tx_timeout* is set to a negative value the messages will be transmitted in DAR mode if supported by the hardware.
61	Use the blocking NTCAN-API ***canWriteX()*** to transmit CAN FD messages. The messages are transmitted with increasing CAN-IDs from *id-first* to *id-last* with a wraparound if *id-last* is reached. If *count* is set to a negative value the test will send frames without switching the bit rate during the data phase with the absolute value of *count.* If *tx_timeout* is set to a negative value the messages will be transmitted in DAR mode if supported by the hardware.
2	Use the non-blocking NTCAN-API ***canTake()*** in FIFO mode to receive CAN messages. The messages are received in the CAN-ID range from *id-first* to *id-last* and dumped on return.
12	Use the non-blocking NTCAN-API ***canTake()*** in FIFO mode to receive 10000 CAN messages. The messages are received in the CAN-ID range from *id-first* to *id-last* without being dumped.
22	Same as test 2 using ***canTakeT()***.

Test #	Description
32	Use the non-blocking NTCAN-API ***canTake()*** in object mode to receive CAN messages. The messages are received in the CAN-ID range from *id-first* to *id-last* and dumped on return.
42	Same as test 32 using ***canTakeT()***.
62	Same as test 2 using ***canTakeX()***.
72	Same as test 12 using ***canTakeX()***.
82	Same as test 32 using ***canTakeX()***.
3	Use the blocking NTCAN-API ***canRead()*** in FIFO mode to receive CAN messages. The messages are received in the CAN-ID range from *id-first* to *id-last* and dumped on return.
13	Use the blocking NTCAN-API ***canRead()*** in FIFO mode to receive 10000 CAN messages. The messages are received in the CAN-ID range from *id-first* to *id-last* without being dumped.
23	Same as test 3 using ***canReadT()***.
63	Same as test 3 using ***canReadX()***.
73	Same as test 13 using ***canReadX()***.
4	Use the blocking NTCAN-API ***canRead()*** in FIFO mode to receive CAN messages. The messages are received in the Event-ID range from *id-first* to *id-last* and dumped on return using ***canFormatEvent()***.
64	Use ***canIoctl()*** to retrieve and dump the CAN bus statistic data. The parameter *txtout* is used as the delay between consecutive loops if *testcount* is greater 1.
74	Use ***canIoctl()*** to reset the CAN bus statistic data. The parameter *txtout* is used as the delay between consecutive loops if *testcount* is greater 1.
84	Use ***canIoctl()*** to retrieve and dump the configured bitrate details for the CAN port. The parameter *txtout* is used as the delay between consecutive loops if *testcount* is greater 1.
6	 Same as test 3 but with support for the *Overlapped* mechanism of Windows to receive data asynchronously (see ***canRead()*** for details). For this reason this test is not available for any other supported OS.
16	 Same as test 23 but with support for the *Overlapped* mechanism of Windows to receive data asynchronously (see ***canReadT()*** for details). For this reason this test is not available for any other supported OS.
66	 Same as test 63 but with support for the *Overlapped* mechanism of Windows to receive data asynchronously (see ***canReadX()*** for details). For this reason this test is not available for any other supported OS.
7	 Same as test 1 but with support for the *Overlapped* mechanism of Windows to transmit data asynchronously (see ***canWrite()*** for details). For this reason this test is not available for any other supported OS.
8	Use ***canIoctl()*** to create an Auto-RTR object with CAN-ID *id-first.* This CAN message can be requested for the time given in parameter *rxtout*.
9	Use the non-blocking NTCAN-API ***canSend()*** to transmit a RTR message followed by ***canRead()*** to receive the reply.
-2	Show the CAN device overview without the parameter description.
-3	Show the CAN device overview without the parameter description but with a textual description of the feature flags and with detailed build information of the library and/or device driver if supported.

Test #	Description
-4	Show the CAN device overview without the parameter description and a table which gives an overview on the relation between the esd electronics bitrate table index and the default CiA bitrates.

Table 33: Test Cases of 'canTest'

10. Application Development

An implementation of the NTCAN architecture for any of the supported platforms usually comprises a CAN hardware specific device driver, a library which exports the NTCAN-API to the (user mode) application, the necessary files (header, …) for the C/C++ development and the console mode application *canTest* described in chapter 9 as source code and binary version (if applicable on the target). The installation and integration into the prevalent development environment for the target is described in /1/.

10.1 CAN SDK for Windows

On the Microsoft® Windows platform the development of NTCAN based applications is accompanied by the ***esd electronics*** CAN Software Development Kit (SDK).

10.1.1 GUI Tools

The CAN SDK comprises several tools which ease and support the development and debugging of CAN applications for Windows but also for other platforms in several ways:

- ***CANreal***: Sophisticated and flexible CAN bus monitoring software.

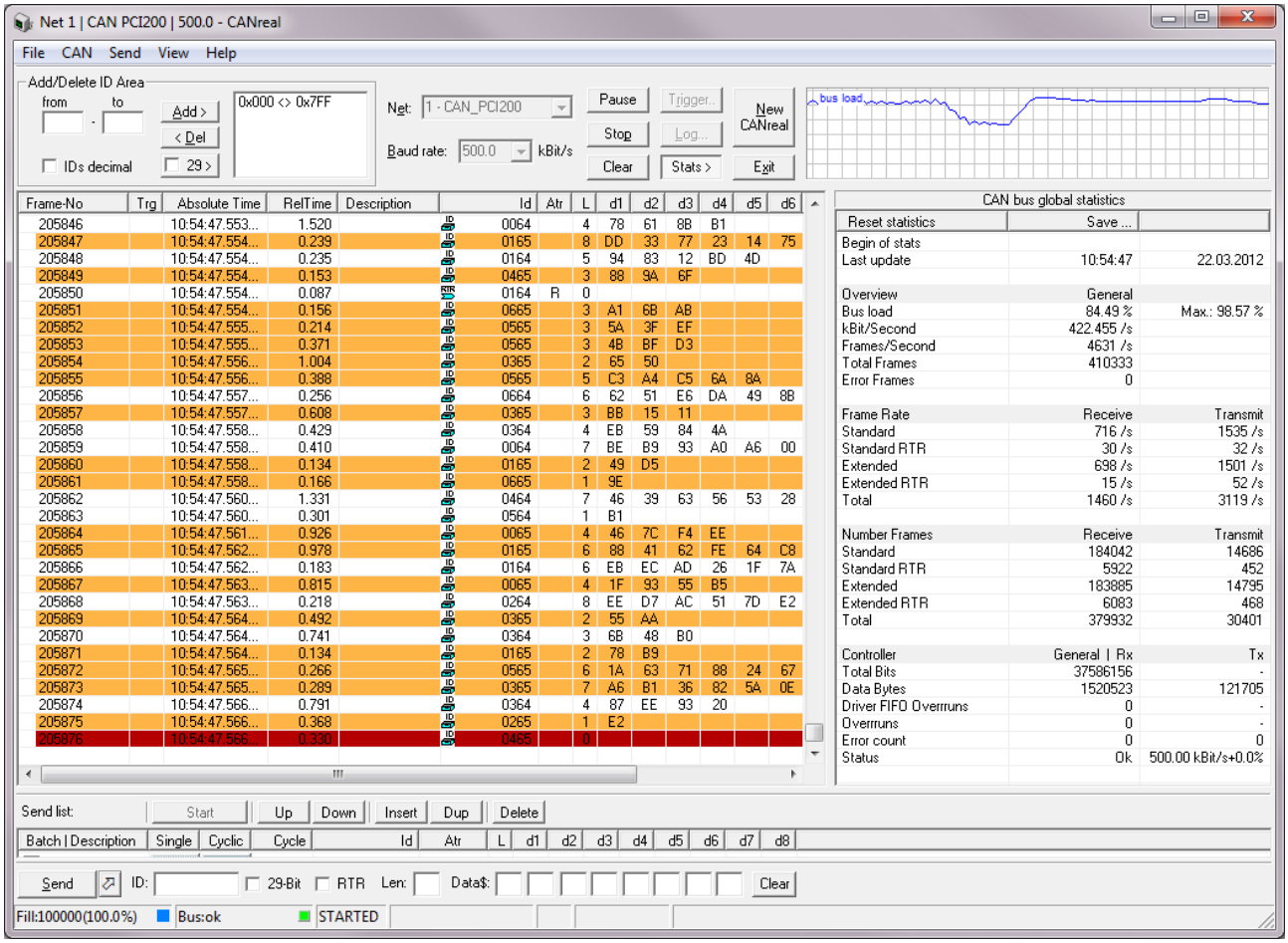
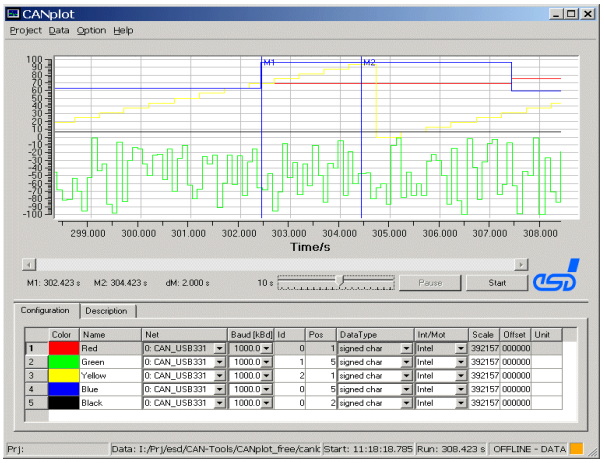
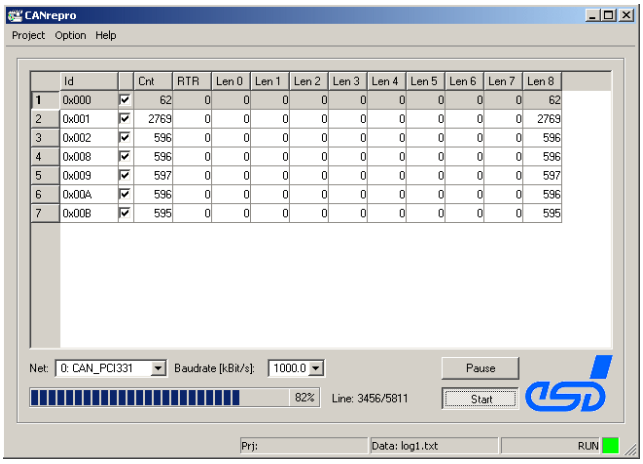
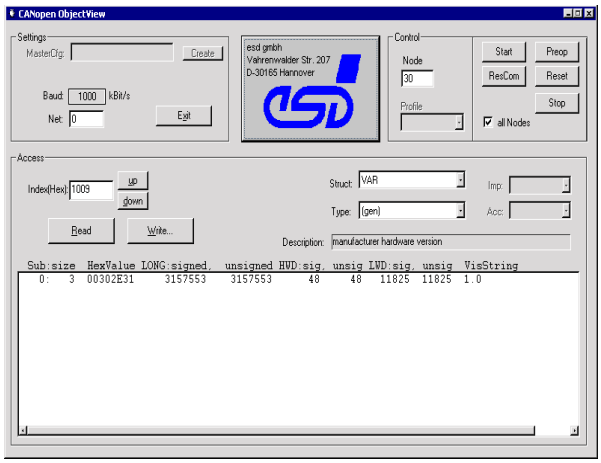


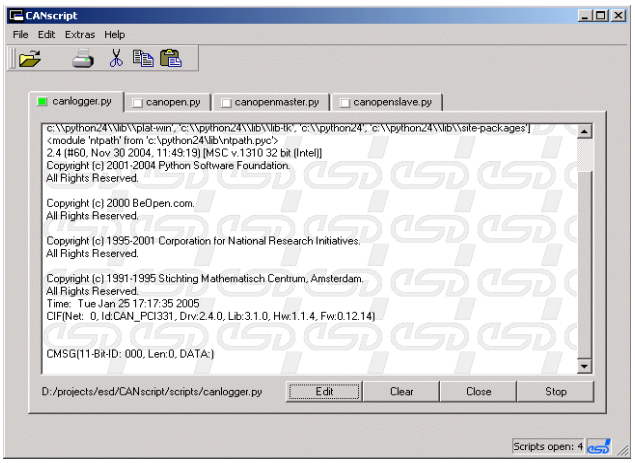
Figure 19: CANreal Bus Monitor

- ***CANrepro***: Replay pre-recorded CAN messages of ***CANreal***.
- ***CANscript***: Automate CAN based processes with a Python based scripting language.
- ***CANplot***: Powerful visualization of CAN messages
- ***COBview***: CANopen Object Viewer to manually configure and start CANopen devices.









All tools come with a separate documentation which is installed together with the CAN SDK.

10.1.2 Programming Language Support & Language Bindings

The NTCAN API library is implemented in C. The CAN SDK contains example out of the box projects for the following C/C++ IDEs:

IDE	Support
Microsoft® Visual Studio	Example projects and import libraries for different versions of Microsoft Visual Studio.
Borland C++ Builder	Example project and import libraries for Borland C++ Builder 6.
Code::Blocks	Example project and import libraries for Code::Blocks 20.03 ready to support MinGW (GCC) C/C++

Table 34: Supported C/C++ IDEs for Windows

As an alternative to using C/C++, the CAN SDK contains language bindings to NTCAN for the following programming languages and runtime environments:

Language / Environment	Support
Embarcadero Delphi	Libraries and example project to support Embarcadero Delphi
Microsoft® Visual Basic 6	Libraries and example project to support Microsoft Visual Basic 6
Python	PyNTCAN library as language binding to Python 3.x
National Instruments LabVIEW®	LabVIEW VI library.
PureBasic	Language Binding to support PureBasic development.
Microsoft .NET Runtime	Class library for .NET 3.5 and .NET Standard 2.0 which enables the development of NTCAN based applications with any .NET capable programming language (C#, F#, Visusal Basic.NET,)

Table 35: Supported Language Bindings for Windows

11. Attachment

11.1 esd electronics NTCAN Programming with LabVIEW

This chapter contains descriptions of the LabVIEW block diagrams to make use of the esd electronics NTCAN API.
There are two different methods to use esd electronics CAN hardware from within LabVIEW.
One method is the use of NTCAN-VIs (native VIs) to the esd electronics NTCAN API C-functions. Those VIs are named with the prefix "*Ntcan*". Their VI icons look like this:

The other method is a LabVIEW signal based approach, using a plain text project file holding informations on the CAN bus configuration, CAN objects and their mappings to LabVIEW signals. These CAN VIs (signal based VIs) are named with the prefix "*Can*". Their VI icons look like this:

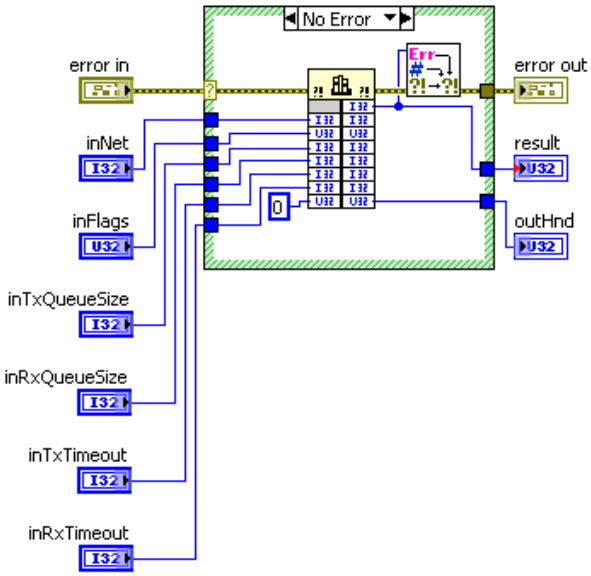
Be careful with mixing up LabVIEW signal based CAN handles obtained by "*Can Project Open*" and handles obtained by the native NTCAN VI "*Ntcan Open*".It is possible to use a LabVIEW signal based handle as input for native NTCAN VIs, but the other way will not work!

11.1.1 Wrapper VIs for direct use of the esd electronics NTCAN API

11.1.1.1 Initialization and Cleanup

11.1.1.1.1 canOpen

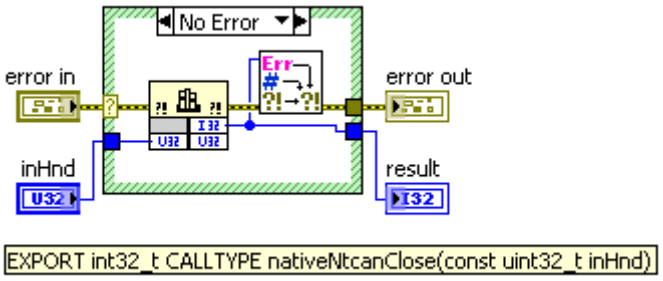
NTCAN API function description see page 98.




```
EXPORT int32_t CALLTYPE nativeNtcanOpen(const  int32_t inNet,
                                        const uint32_t inFlags,
                                        const  int32_t inTxQueueSize,
                                        const  int32_t inRxQueueSize,
                                        const  int32_t inTxTimeout,
                                        const  int32_t inRxTimeout,
                                        uint32_t * const outHnd)
```

11.1.1.1.2 canClose

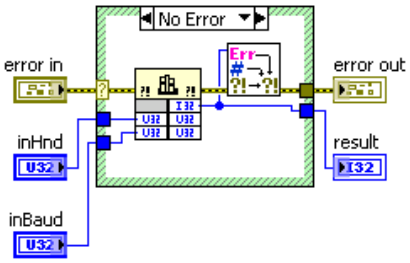
NTCAN API function description see page 102.



11.1.1.2 Configuration

11.1.1.2.1 canSetBaudrate

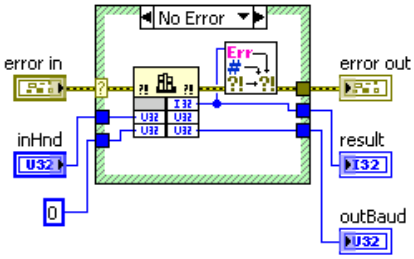
NTCAN API function description see page 103.




```
EXPORT int32_t CALLTYPE nativeNtcanSetBaudrate(const uint32_t inHnd,
                                               const uint32_t inBaud)
```

11.1.1.2.2 canGetBaudrate

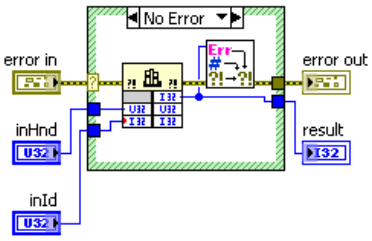
NTCAN API function description see page 109.




```
EXPORT int32_t CALLTYPE nativeNtcanGetBaudrate(const uint32_t   inHnd,
                                               const uint32_t * inBaud)
```

11.1.1.2.3 canIdAdd

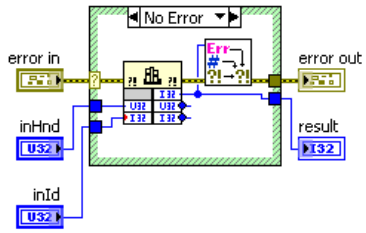
NTCAN API function description see page 113.




```
EXPORT int32_t CALLTYPE nativeNtcanIdAdd(const uint32_t inHnd, /* Handle                  */
                                         const int32_t  inId)  /* CAN message identifier */
```

11.1.1.2.4 canIdDelete

NTCAN API function description see page 116.

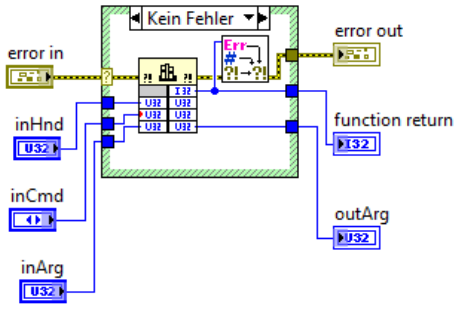



```
EXPORT int32_t CALLTYPE nativeNtcanIdDelete(const uint32_t inHnd, /* Handle                    */
                                            const int32_t  inId)  /* CAN message identifier   */
```

11.1.1.2.5 canIoctl

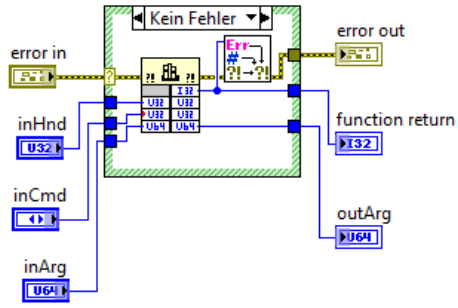
NTCAN API function description see page 118.

NTCAN Ioctl 32 (32-bit value as input and output)



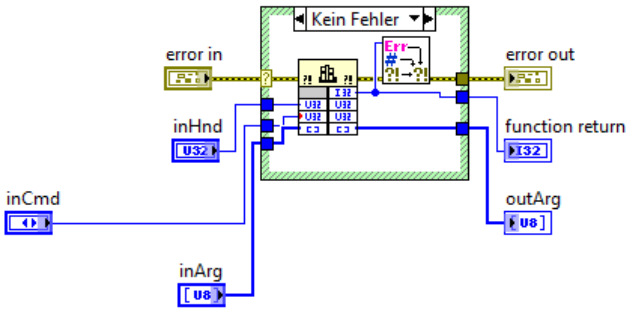

```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,    /* Handle                    */
                                         const uint32_t inCmd,    /* Command specifier         */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

NTCAN Ioctl 64 (64-bit value as in- and output)




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,    /* Handle                    */
                                         const uint32_t inCmd,    /* Command specifier         */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

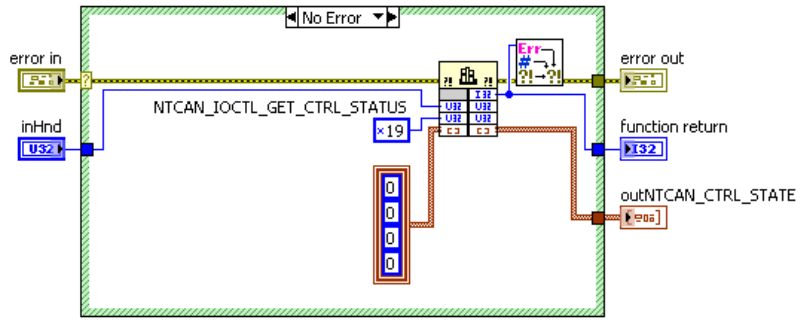
NTCAN Ioctl Raw (Pointer as in- and output)




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,    /* Handle                          */
                                         const uint32_t inCmd,    /* Command specifier               */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

NTCAN Ioctl Get Ctrl Status (NTCAN_CTRL_STATE structure as output)

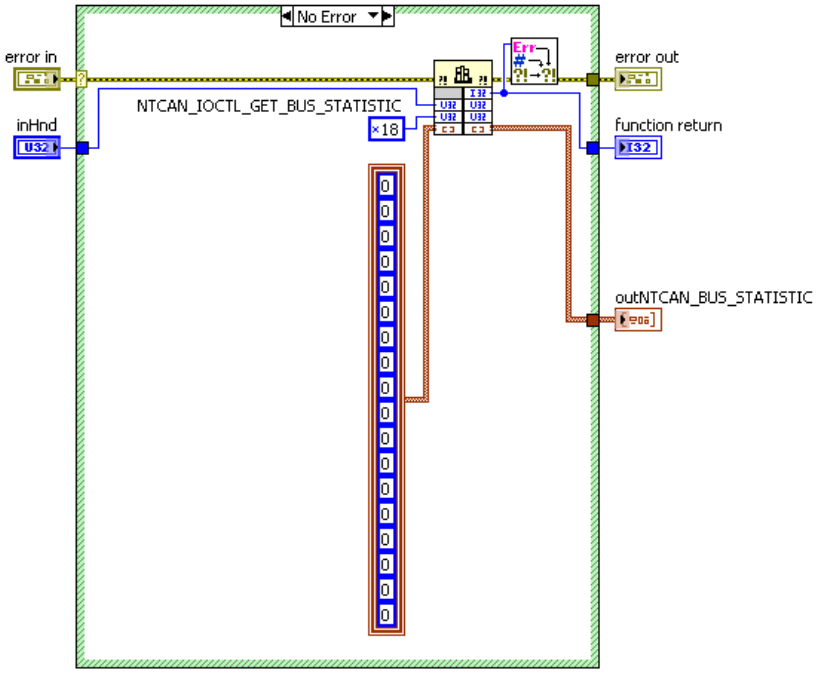
NTCAN API function description see page 209.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,    /* Handle                          */
                                         const uint32_t inCmd,    /* Command specifier               */
                                         void  * const  inoutArg) /* Ptr to command specific argument */
```

NTCAN Ioctl Get Bus Statistic (NTCAN_BUS_STATISTIC structure as output)

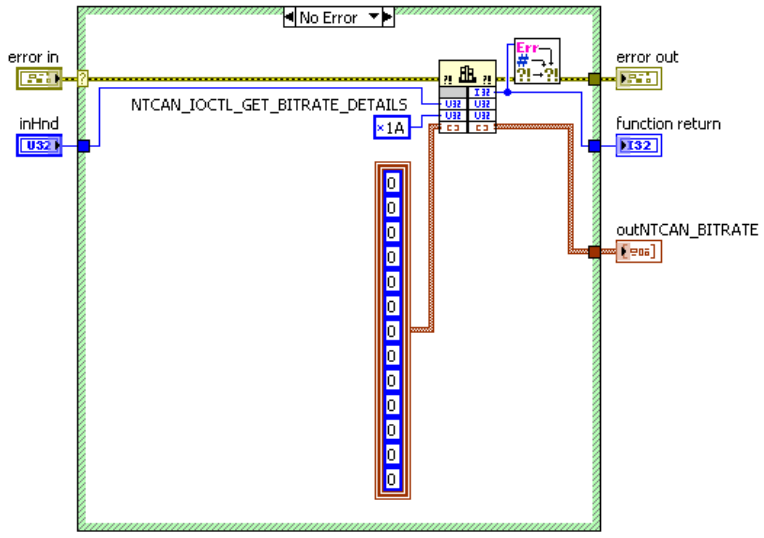
NTCAN API function description see page 207.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg) /* Ptr to command specific argument */
```

NTCAN Ioctl Get Bit Rate Details (NTCAN_BITRATE structure as output)

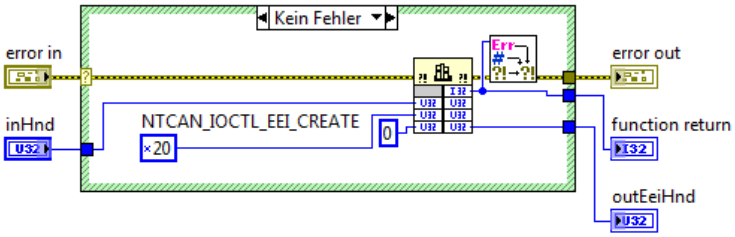
NTCAN API function description see page 205.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg) /* Ptr to command specific argument */
```

NTCAN Ioctl EEI Create

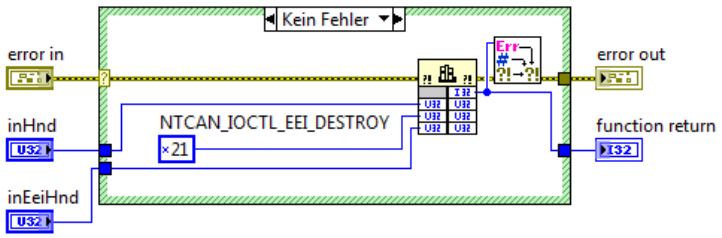
For general description of the Error Injection see page 74. The error injection related I/O controls are listed on page 125.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

NTCAN Ioctl EEI Destroy

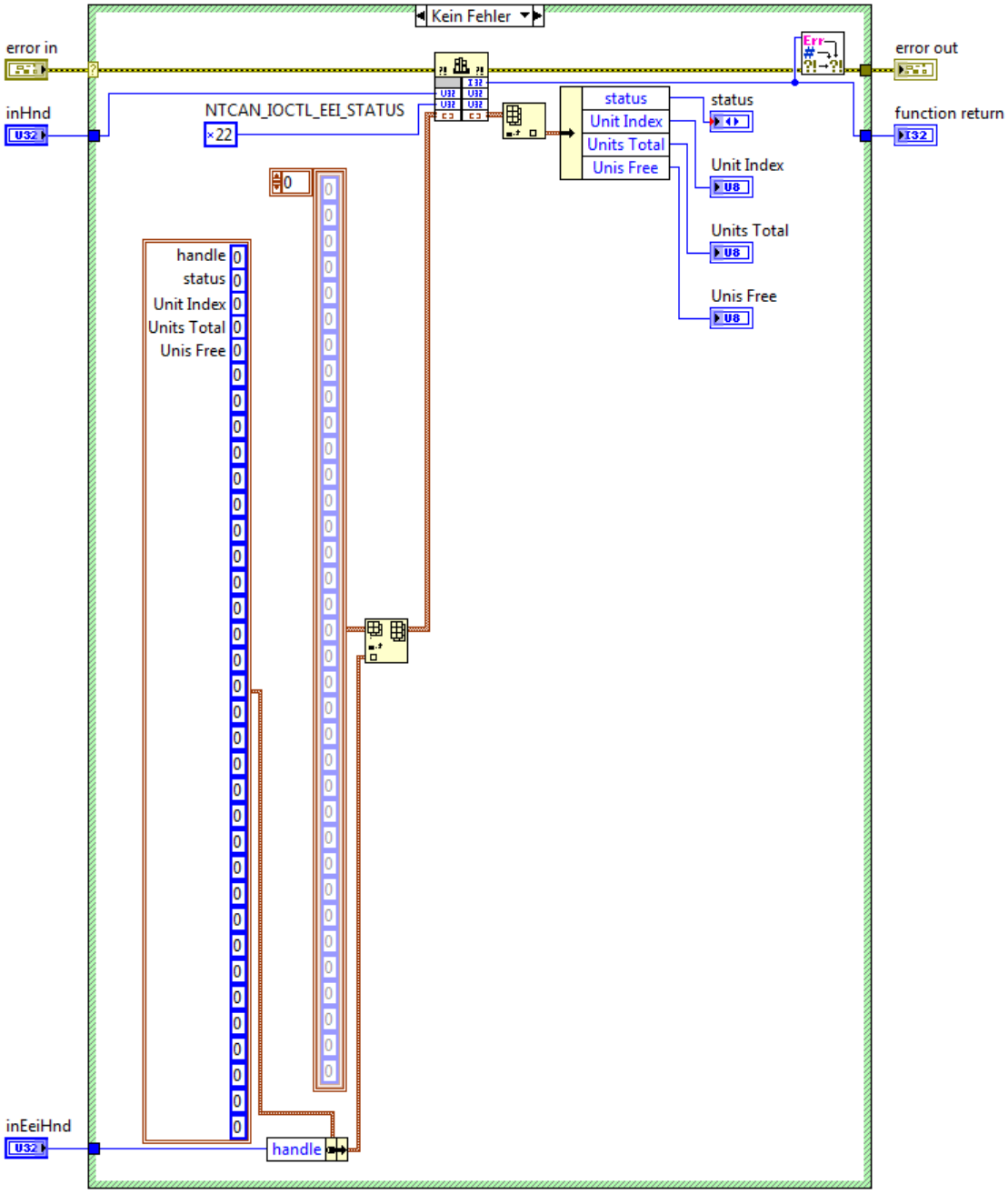
For general description of the Error Injection see page 74. The error injection related I/O controls are listed on page 125.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

NTCAN Ioctl EEI Status

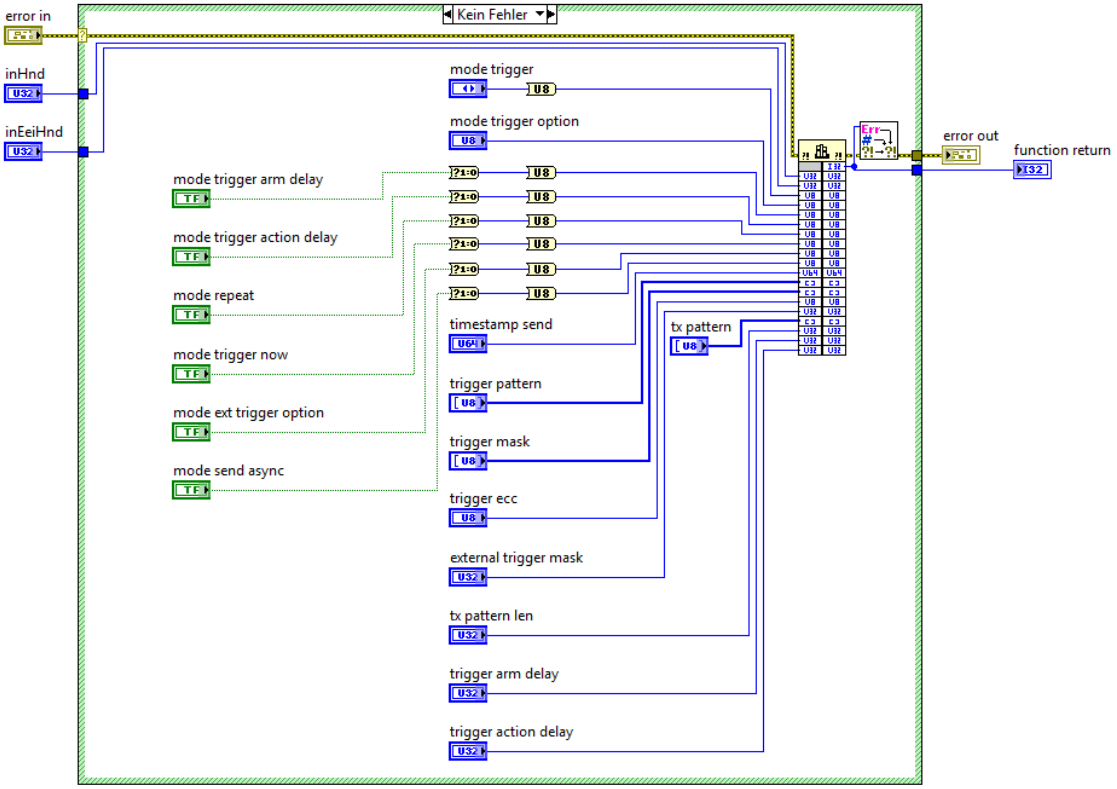
For general description of the Error Injection see page 74. The error injection related I/O controls are listed on page 125.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

NTCAN Ioctl EEI Configure

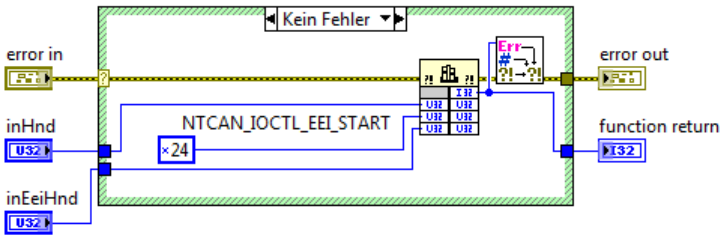
For general description of the Error Injection see page 74. The error injection related I/O controls are listed on page 125.




```
EXPORT int32_t CALLTYPE nativeNtcanEeiConfigure(const uint32_t inHnd,                  /* Handle                                  */
                                                const uint32_t inEeiHnd,               /* Handle for ErrorInjection Unit          */

                                                const uint8_t inModeTrigger,           /* Trigger mode                            */
                                                const uint8_t inModeTriggerOption,     /* Options to trigger                      */
                                                const uint8_t inModeTriggerArmDelay,   /* Enable delayed arming of trigger unit*/
                                                const uint8_t inModeTriggerActionDelay,/* Enable delayed TX out                   */
                                                const uint8_t inModeRepeat,            /* Enable repeat                           */
                                                const uint8_t inModeTriggerNow,        /* Trigger with next TX point              */
                                                const uint8_t inModeExtTriggerOption,  /* Switch between trigger and sending      */
                                                const uint8_t inModeSendAsync,         /* Send without timing synchronization     */

                                                const uint64_t inTimestampSend,        /* Timestamp for Trigger Timestamp         */
                                                uint8_t * const inTriggerPattern,      /* Trigger for mode Pattern Match          */
                                                uint8_t * const inTriggerMask,         /* Mask to trigger Pattern                 */
                                                const uint8_t  inTriggerEcc,           /* ECC for Trigger Field Position          */
                                                const uint32_t inExternalTriggerMask,  /* Enable Mask for external Trigger        */

                                                uint8_t * const inTx_pattern,          /* TX pattern                              */
                                                const uint32_t inTx_patternLen,        /* Length of TX pattern                    */
                                                const uint32_t inTriggerArmDelay,      /* Delay for mode triggerarm delay         */
                                                const uint32_t inTriggerActionDelay)   /* Delay for mode trigger delay            */
```

NTCAN Ioctl EEI Start

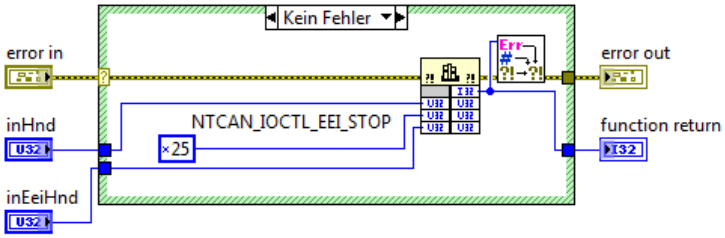
For general description of the Error Injection see page 74. The error injection related I/O controls are listed on page 125.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

NTCAN Ioctl EEI Stop

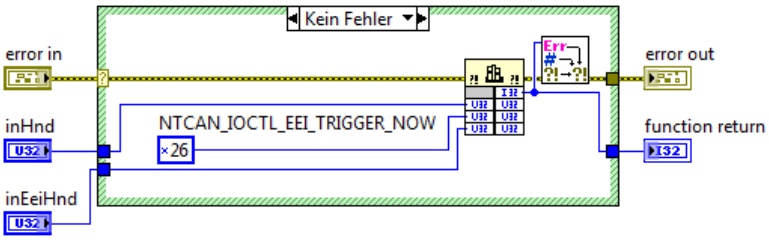
For general description of the Error Injection see page 74. The error injection related I/O controls are listed on page 125.




```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

NTCAN Ioctl EEI Trigger Now

For general description of the Error Injection see page 74. The error injection related I/O controls are listed on page 125.

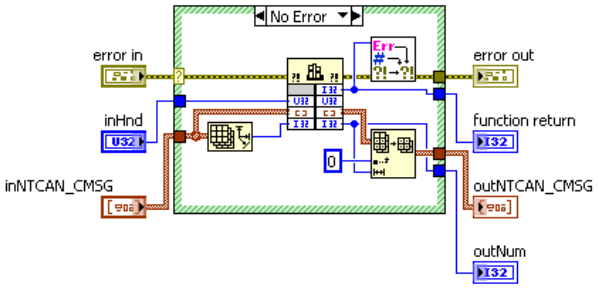



```
EXPORT int32_t CALLTYPE nativeNtcanIoctl(const uint32_t inHnd,     /* Handle                          */
                                         const uint32_t inCmd,     /* Command specifier               */
                                         void  * const  inoutArg);/* Ptr to command specific argument */
```

11.1.1.3 Receiving CAN messages

11.1.1.3.1 canTake

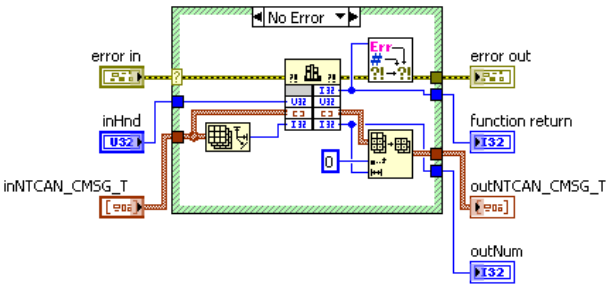
NTCAN API function description see page 129.




```
EXPORT int32_t CALLTYPE nativeNtcanTake(const uint32_t inHnd,          /* Handle                    */
                                        CMSG         * const inoutCmsg, /* Pointer to CAN data buffer */
                                        int32_t      * const inoutNum); /* OUT: # of CMSG received   */
                                                                        /* IN: max # of CMSG to return */
```

11.1.1.3.2 canTakeT

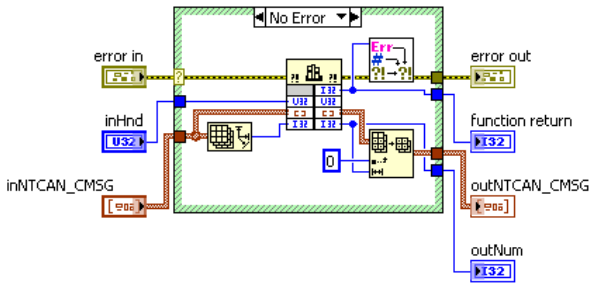
NTCAN API function description see page 131.




```
EXPORT int32_t CALLTYPE nativeNtcanTakeT(const uint32_t inHnd,          /* Handle                    */
                                         CMSG_T       * const inoutCmsgT,/* Pointer to CAN data buffer */
                                         int32_t      * const inoutNum); /* OUT: # of CMSG received   */
                                                                         /* IN: max # of CMSG to return */
```

11.1.1.3.3 canRead

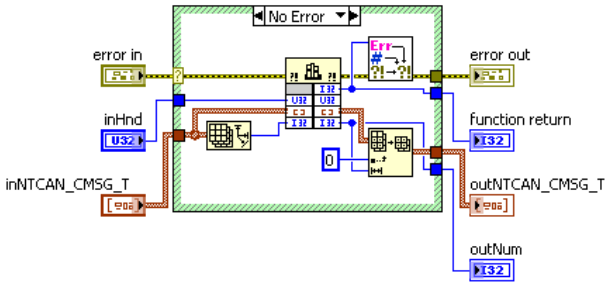
NTCAN API function description see page 135.




```
EXPORT int32_t CALLTYPE nativeNtcanRead(const uint32_t inHnd,          /* Handle                    */
                                        CMSG         * const inoutCmsg, /* Pointer to CAN data buffer */
                                        int32_t      * const inoutNum); /* OUT: # of CMSG received   */
                                                                        /* IN: max # of CMSG to return */
```

11.1.1.3.4 canReadT

NTCAN API function description see page 138.

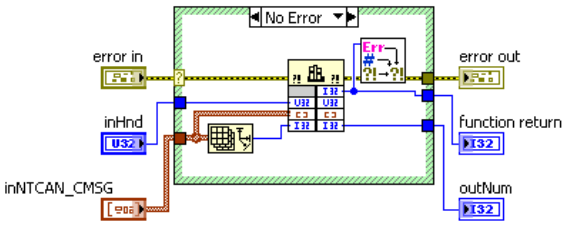



```
EXPORT int32_t CALLTYPE nativeNtcanReadT(const uint32_t inHnd,          /* Handle                          */
                                         CMSG_T       * const inoutCmsgT,/* Pointer to CAN data buffer      */
                                         int32_t      * const inoutNum); /* OUT: # of CMSG received         */
                                                                         /* IN: max # of CMSG to return     */
```

11.1.1.4 Transmitting CAN messages

11.1.1.4.1 canSend

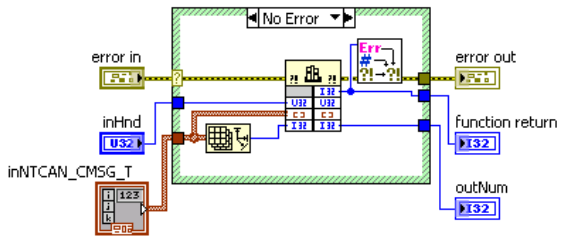
NTCAN API function description see page 142.




```
EXPORT int32_t CALLTYPE nativeNtcanSend(const uint32_t inHnd,          /* Handle                          */
                                        CMSG         * const inCmsg,   /* Pointer to CAN data buffer      */
                                        int32_t      * const inoutNum) /* OUT: # of messages to transmit  */
                                                                       /* IN:  # of msgs stored in TX FIFO */
```

11.1.1.4.2 canSendT

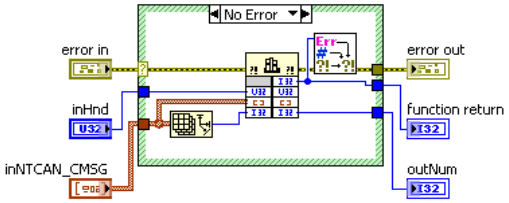
NTCAN API function description see page 144.




```
EXPORT int32_t CALLTYPE nativeNtcanSendT(const uint32_t inHnd,          /* Handle                          */
                                         CMSG_T       * const inCmsgT,  /* Pointer to CAN data buffer      */
                                         int32_t      * const inoutNum) /* OUT: # of messages to transmit  */
                                                                        /* IN:  # of msgs stored in TX FIFO */
```

11.1.1.4.3 canWrite

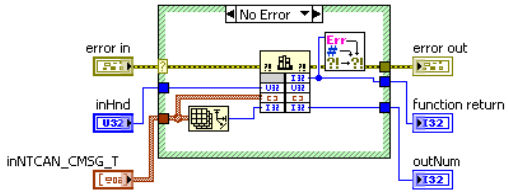
NTCAN API function description see page 148.




```
EXPORT int32_t CALLTYPE nativeNtcanWrite(const uint32_t inHnd,          /* Handle                          */
                                         CMSG         * const inCmsg,   /* Pointer to CAN data buffer      */
                                         int32_t      * const inoutNum);/* OUT: # of messages to transmit  */
                                                                        /* IN:  # of transmitted messages  */
```

11.1.1.4.4 canWriteT

NTCAN API function description see page 150.

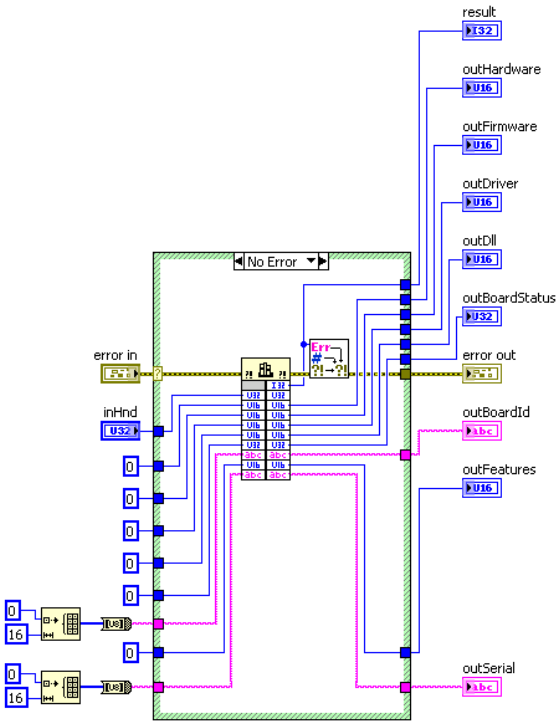



```
EXPORT int32_t CALLTYPE nativeNtcanWriteT(const uint32_t inHnd,          /* Handle                          */
                                          CMSG_T       * const inCmsgT  /* Pointer to CAN data buffer      */
                                          int32_t      * const inoutNum);/* OUT: # of messages to transmit  */
                                                                         /* IN:  # of transmitted messages  */
```

11.1.1.5 Miscellaneous functions

11.1.1.5.1 canStatus

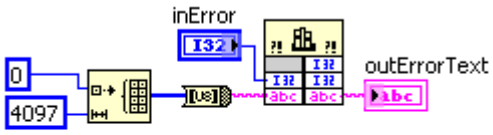
NTCAN API function description see page 156.




```
EXPORT int32_t CALLTYPE nativeNtcanStatus(const uint32_t inHnd,              /* Handle */
                                          uint16_t * const outHardware,     /* Hardware version */
                                          uint16_t * const outFirmware,     /* Firmware version (0 for passive hardware) */
                                          uint16_t * const outDriver,       /* Driver version */
                                          uint16_t * const outDll,          /* NTCAN library version */
                                          uint32_t * const outBoardStatus,  /* Hardware status */
                                          char     * const outBoardId,      /* Board ID string */
                                          uint16_t * const outFeatures,     /* Device/driver capability flags */
                                          char     * const outSerialNumber); /* Board's serial number */
```

11.1.1.5.2 canFormatError

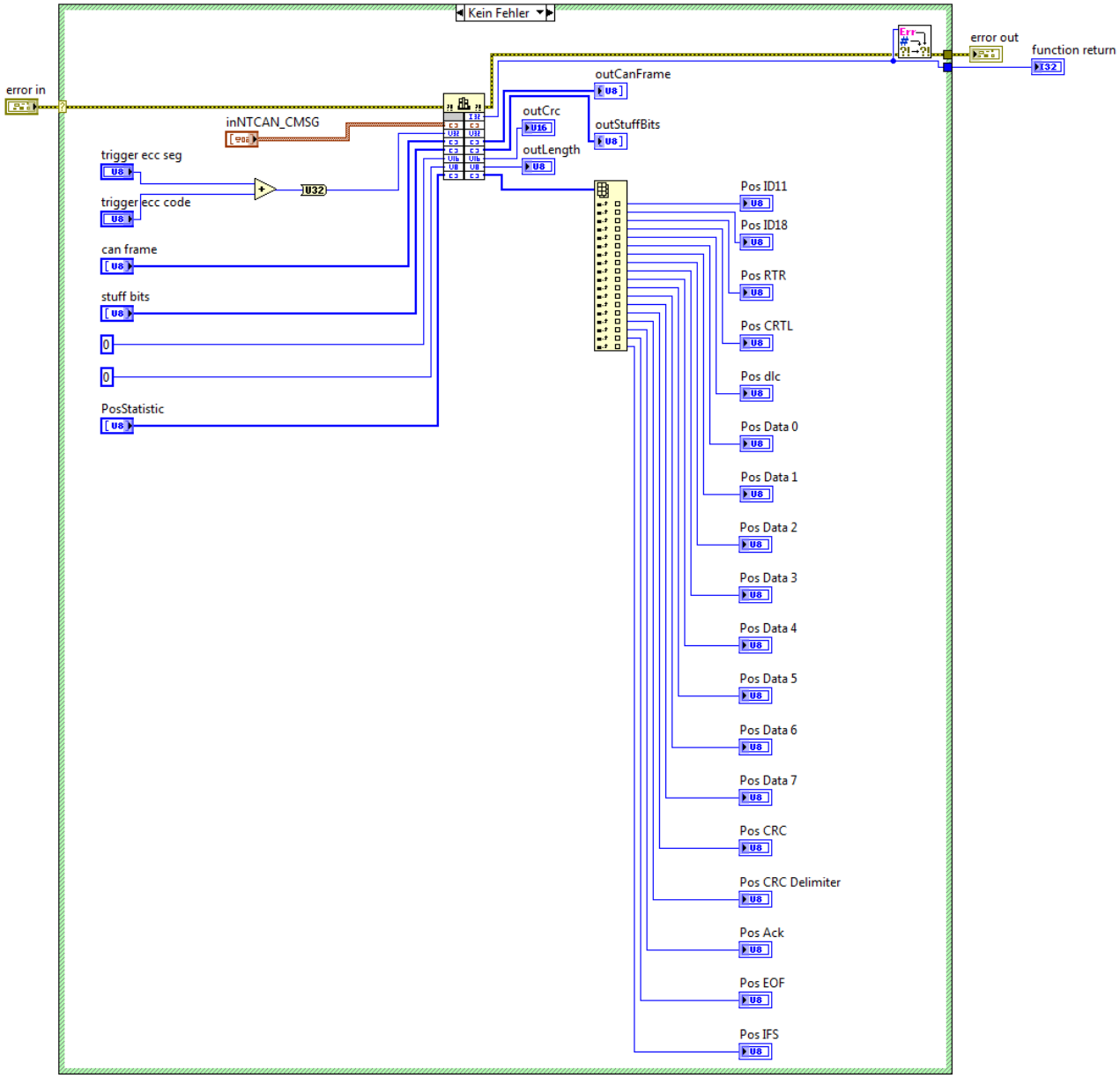
NTCAN API function description see page 163.




```
EXPORT int CALLTYPE NtcanError2String(const int inError, char * const outErrorText);
```

11.1.1.5.3 canFormatFrame

NTCAN API function description see page 167.

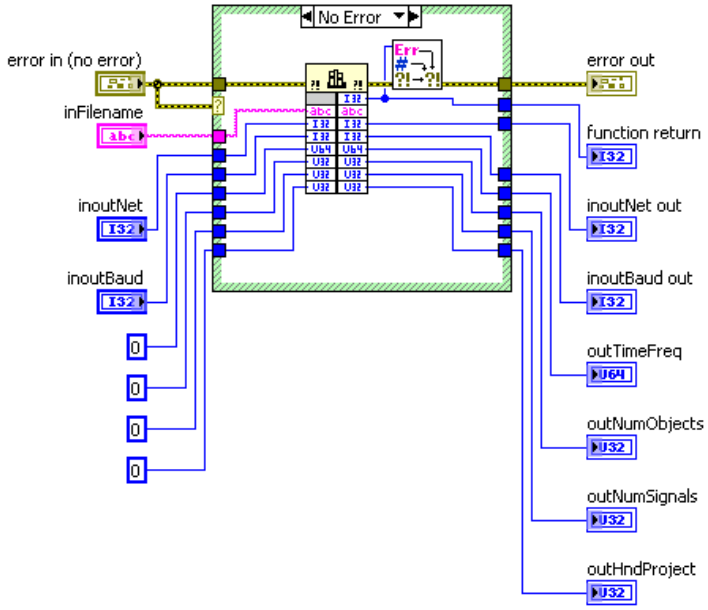



```
EXPORT int CALLTYPE NtcanFormatFrame(CMSG * const inCmsg,
                                     const uint32_t inEccExt,
                                     uint8_t * const outCanFrame,
                                     uint8_t * const outStuffBits,
                                     uint16_t * const outCrc,
                                     uint8_t * const outLength,
                                     uint8_t * const outPosStatistic);
```

11.1.2 LabVIEW signal based access to CAN

11.1.2.1.1 CanProjectOpen

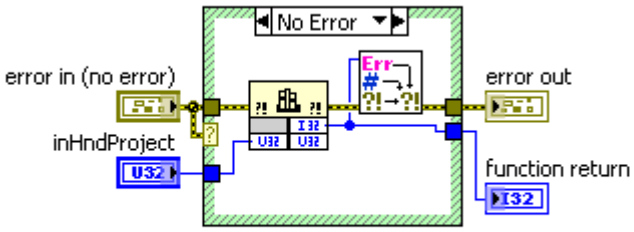
CanProjectOpen opens the specified project file (which is holding informations about the CAN bus configuration, CAN objects and their mappings to LabVIEW signals) to obtain a project handle. Please see the accompanying commented example project files for more information.




```
EXPORT int CALLTYPE CanProjectOpen(const char const *inFilename,
                                   int * const inoutNet, uint32_t * const inoutBaud,
                                   uint64_t * const outTimeFreq, uint32_t * const outNumObjects,
                                   uint32_t * const outNumSignals, uint32_t * const outHndProject)
```

11.1.2.1.2 CanProjectClose

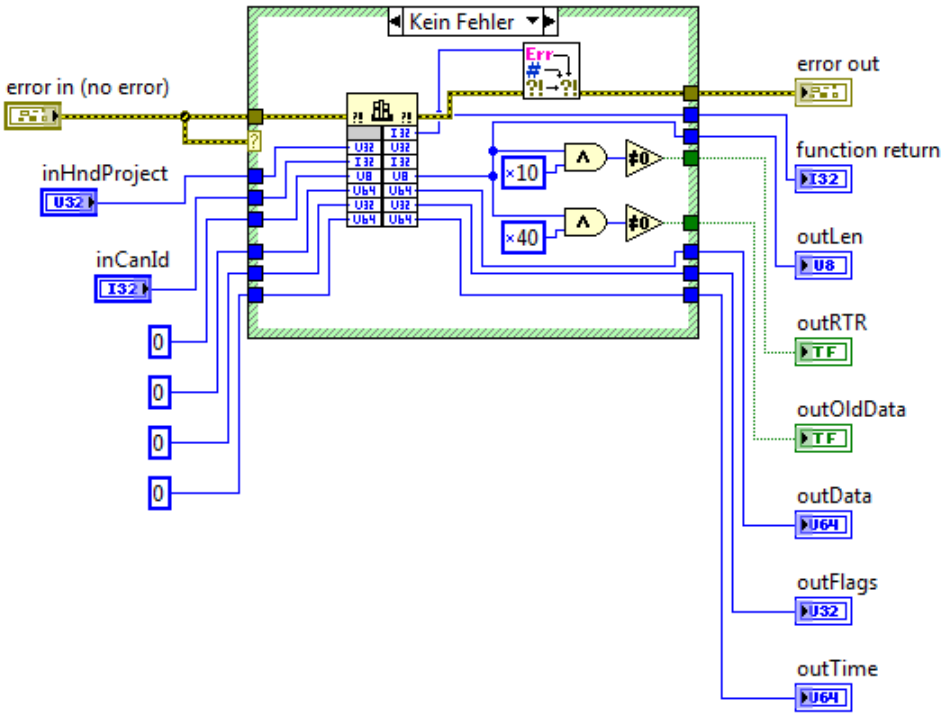
CanProjectClose closes a project handle. It is advised to always close any opened project handle, before stopping a LabVIEW project.




```
EXPORT int CALLTYPE CanProjectClose(const uint32_t inHndProject);
```

11.1.2.1.3 CanObjectPoll

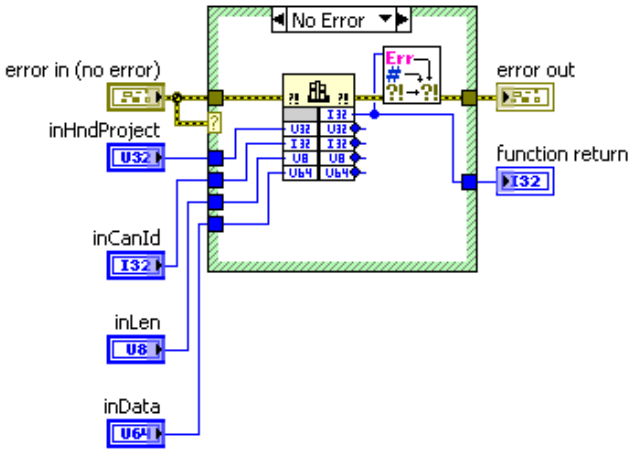
CanObjectPoll gets the most recent CAN data received on the given CAN-ID.




```
EXPORT int CALLTYPE CanObjectPoll(const uint32_t inHndProject,
                                  const int32_t inCanId, uint8_t * const outLen,
                                  uint64_t * const outData, uint32_t * const outFlags,
                                  uint64_t * const outTime);
```

11.1.2.1.4 CanObjectSend

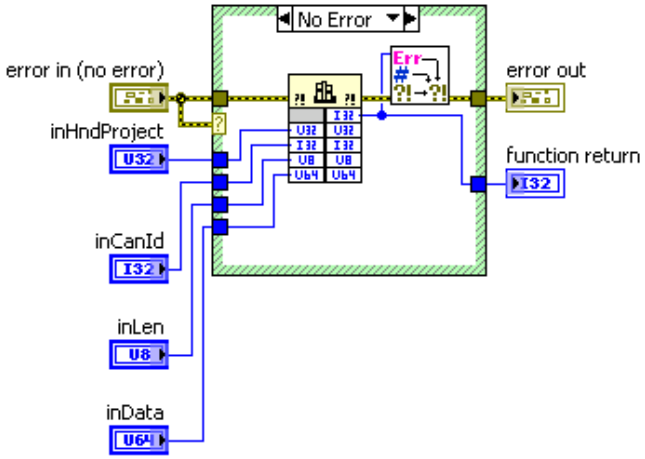
CanObjectSend immediately transmits a CAN frame and in parallel updates the data on the given CAN-ID.




```
EXPORT int CALLTYPE CanObjectSend(const uint32_t inHndProject,
                                  const int32_t inCanId,
                                  const uint8_t inLen,
                                  const uint64_t inData);
```

11.1.2.1.5 CanTxObjectUpdate

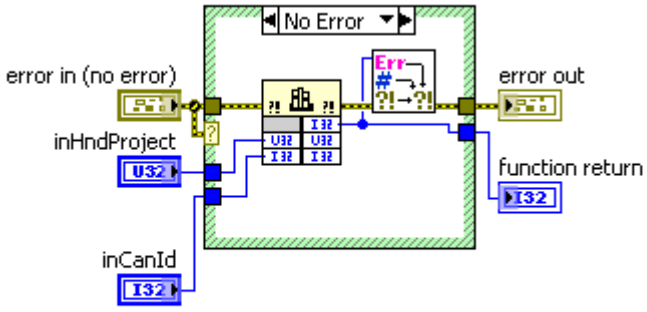
CanTxObjectUpdate updates the data on the given CAN-ID (CAN frame won't be transmitted in this process).




```
EXPORT int CALLTYPE CanTxObjectUpdate(const uint32_t inHndProject,
                                      const int32_t inCanId, const uint8_t inLen,
                                      const uint64_t inData);
```

11.1.2.1.6 CanObjectTrigger

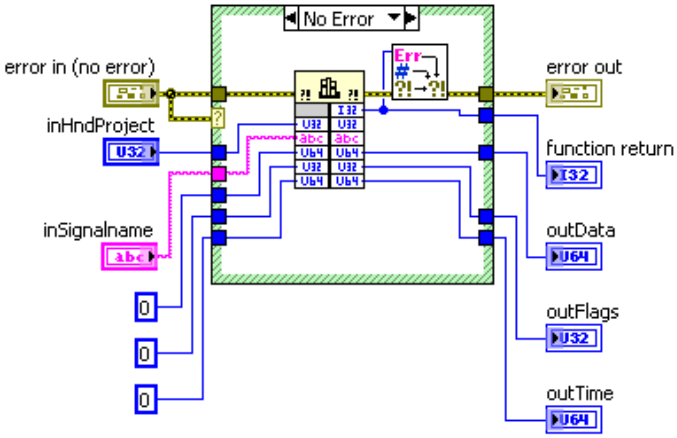
CanObjectTrigger transmits recent data (set by either canTxObjectUpdate or canObjectSend) for given CAN-ID on the CAN-bus.




```
EXPORT int CALLTYPE CanObjectTrigger(const uint32_t inHndProject,
                                     const int32_t inCanId);
```

11.1.2.1.7 CanSignalPoll

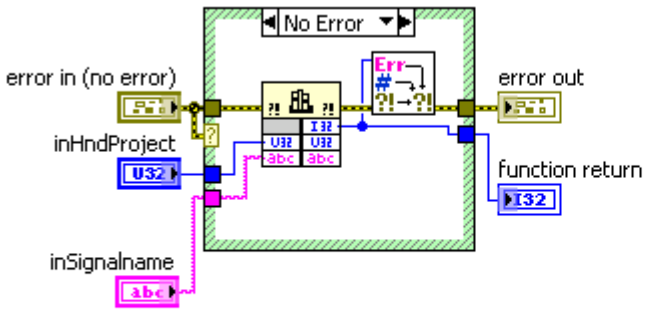
CanSignalPoll gets the most recent value received for the given signal name.




```
EXPORT int CALLTYPE CanSignalPoll(const uint32_t inHndProject,
                                  const char * const inSignalname,
                                  uint64_t * const outData, uint32_t * const outFlags,
                                  uint64_t * const outTime);
```

11.1.2.1.8 CanSignalTrigger

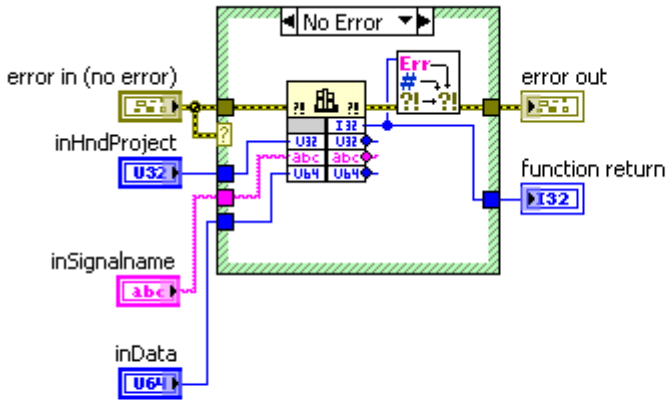
CanSignalTrigger transmits the CAN frame belonging to the given LabVIEW signal name on the CAN-bus.




```
EXPORT int CALLTYPE CanSignalTrigger(const uint32_t inHndProject,
                                     const char * const inSignalname);
```

11.1.2.1.9 CanSignalUpdate

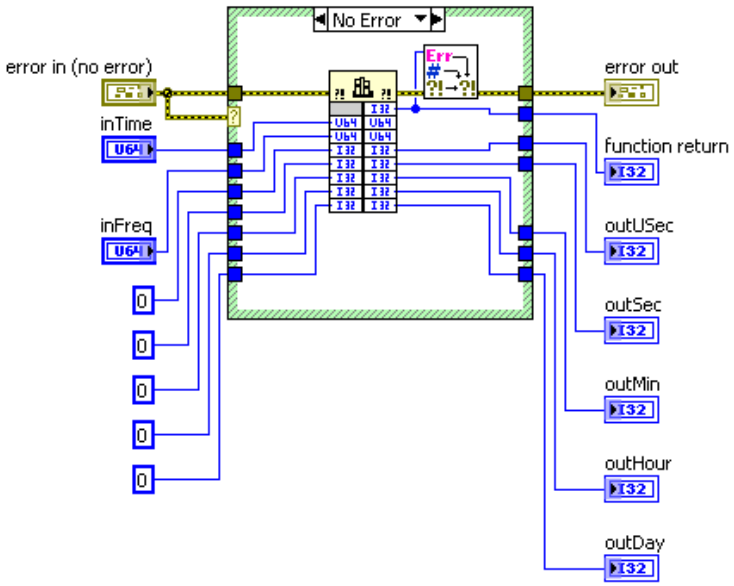
CanSignalUpdate updates the given LabVIEW signal (no CAN frame is transmitted in the process).




```
EXPORT int CALLTYPE CanSignalUpdate(const uint32_t inHndProject,
                                    const char * const inSignalname,
                                    const uint64_t inData);
```

11.1.2.1.10 CanConvertTime

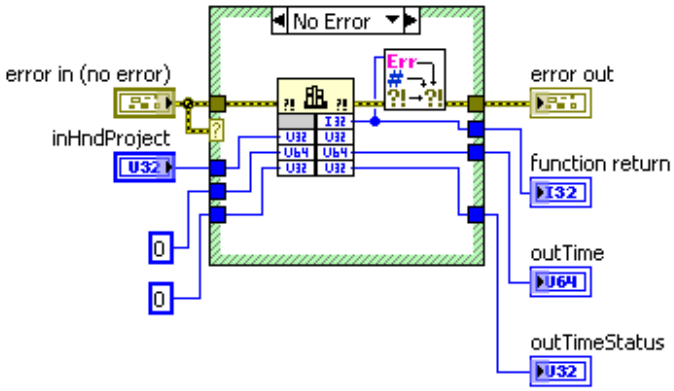
CanConvertTime breaks down the given 64-bit time-stamp value into days, hours, minutes, seconds and micro seconds.




```
EXPORT int CALLTYPE CanConvertTime(const uint64_t inTime,   const uint64_t inFreq,
                                   int32_t * const outUSec, int32_t * const outSec,
                                   int32_t * const outMin,  int32_t * const outHour,
                                   int32_t * const outDay);
```

11.1.2.1.11 CanTimeGet

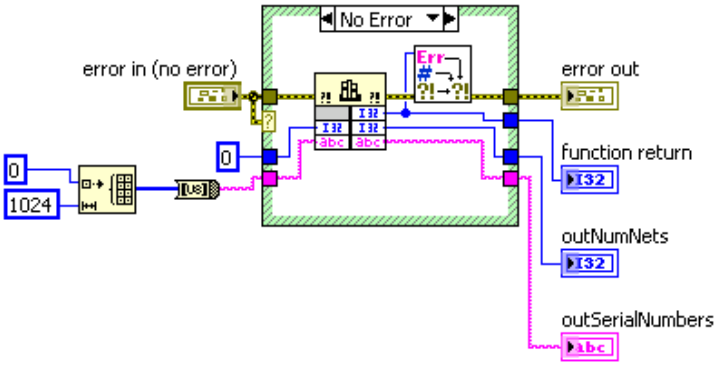
CanTimeGet gets current time stamp.




```
EXPORT int CALLTYPE CanTimeGet(const uint32_t inHndProject,
                               uint64_t * const outTime,
                               uint32_t * const outTimeStatus);
```

11.1.2.1.12 CanInfo

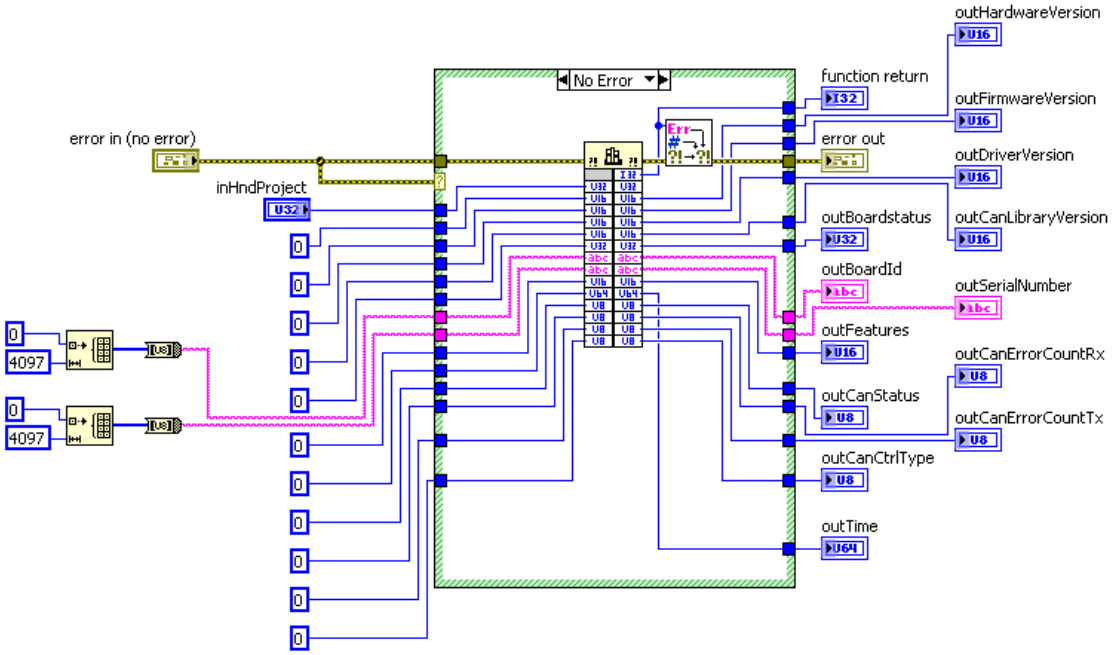
CanInfo returns the number of available CAN nets.




```
EXPORT int CALLTYPE CanInfo(int32_t * const outNumNets, char * outSerialNumbers);
```

11.1.2.1.13 CanStatus

CanStatus returns several information for given project handle.




```
EXPORT int CALLTYPE CanStatus(const uint32_t inHndProject,
                              uint16_t * const outHardwareVersion, uint16_t * const outFirmwareVersion,
                              uint16_t * const outDriverVersion, uint16_t * const outCanLibraryVersion,
                              uint32_t * const outBoardstatus,
                              char * const outBoardId, char * const outSerialNumber,
                              uint16_t * const outFeatures,
                              uint64_t * const outTime,
                              uint8_t * const outCanStatus, uint8_t * const outCanErrorCountRx,
                              uint8_t * const outCanErrorCountTx, uint8_t * const outCanCtrlType)
```

Annex A: Bus Timing

A.1 Can Bit Timing

The ESDACC supports nominal bit rates between 10 kBit/s and 1000 kBit/s.

A CAN bit timing logic monitors the serial bus-line and performs sampling and adjustment of the sample point by synchronizing on the start-bit edge and resynchronizing on the following edges.

According to the CAN specification /2/ the nominal bit time is split into four segments:

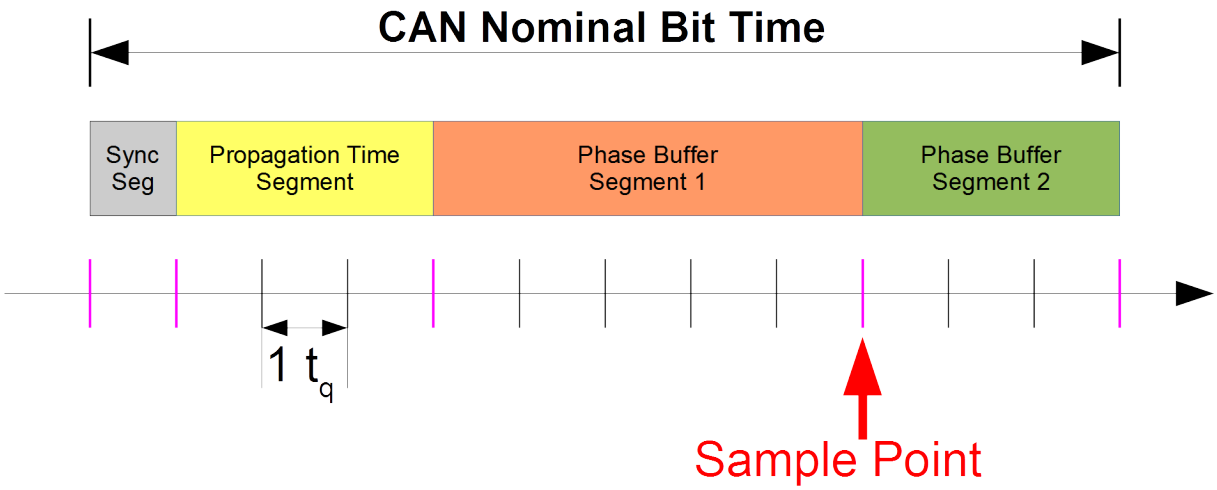


Figure 20: CAN Nominal Bit Time

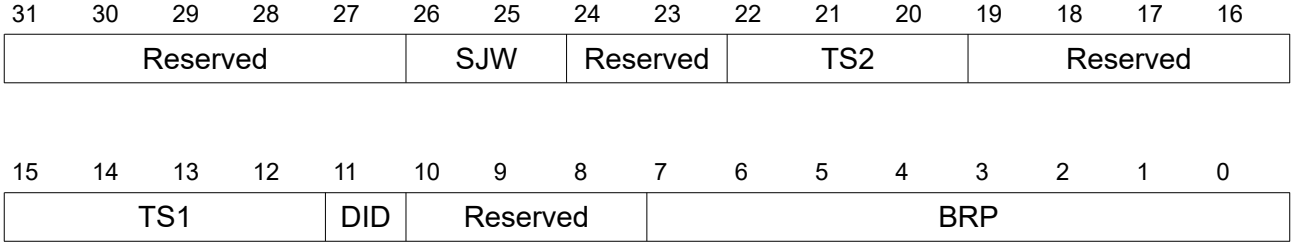
Each segment consists of a specific number of time quanta. The length of one time quantum (t_q), which is the basic time unit of the bit time, is determined by the clock rate (CAN_CLK) of the CAN controller and a Baud Rate Prescaler. Apart from the fixed length of the synchronization segment, the time quanta of each segment are programmable.

Parameter	Description
Sync_Seg	The edges of the bus level are expected to occur within the **Synchronization Segment**. It has a fixed length of one time quantum (1 t_q). If an edge occurs outside of Sync_Seg, its distance is called the phase error of this edge. If the edge occurs before Sync_Seg, the phase error is negative, else it is positive.
Prop_Seq	The **Propagation Segment** is used to compensate physical delay times within the CAN network. These delay times consist of the signal propagation time on the bus and the internal delay time of the CAN nodes.
Phase_Seg1	The **Phase Buffer Segment 1** is used to compensate for positive phase errors and may be lengthened during resynchronization. The end of this segment defines the **Sample Point** (SP).
Phase_Seq2	The **Phase Buffer Segment 2** is used to compensate for negative phase errors and may be shortened during resynchronization.
Synchronization Jump Width	The **Synchronization Jump Width** (SJW) defines an upper bound to the amount of lengthening or shortening of the bit segments in the respective Phase Buffer Segments.

Table 36: CAN Bit Time Parameters

A.2 ESDACC Bus Timing Register for CAN CC

The ESDACC BTR Register is shown in the figure below and described in table 37.



Bit	Field	Description
31..27	Reserved	
26..25	SJW	Synchronization Jump Width. The actual SJW value used for the synchronization will be the programmed SJW value + 1. According to /1/ the programmed SJW value must not exceed the time quanta assigned to the Phase Buffer Segment 1.
24..23	Reserved	
22..20	TS2	Time segment after the sample point. The actual TS2 value used for the bit timing will be the programmed TS2 value + 1. According to /1/ the minimum value of TS2 is the configured SJW.
19..16	Reserved	
15..12	TS1	Time segment before the sample point covering the Propagation Segment and the Phase Buffer Segment 1. The actual TS1 value used for the bit timing will be the programmed TS1 value + 1.
11	DID*	Disable Implicit Divider. 0h = Enable implicit divider of CAN_CLK. 1h = Disable implicit divider of CAN_CLK.
10..8	Reserved	
7..0	BRP	Baud Rate Prescaler. Value which is divided by CAN controller clock rate (CAN_CLK) in Hz for generating the bit time quantum (t_q). The CAN Nominal Bit Time (see picture Error: Reference source not found) is built up from a multiple of this quantum. **Note:** Without DID (CAN_CLK/2) is used for t_q otherwise CAN_CLK.

Table 37: ESDACC Bus Timing Register

* Supported since ESDCACC version 00.53

The CAN bit time may be programmed in the range of 4 to 25 time quanta. The CAN time quantum may be programmed in the range from 1 to 256 CAN_CLK periods (2 to 128 CAN_CLK periods for implementations without DID support).

The following formulas are used for calculating the CAN bit rate:

t_q	= (2 * BRP) / CAN_CLK with DID = 0
	- OR -
t_q	= BRP / CAN_CLK with DID = 1
t_{BS1}	= t_q * (TS1 + 1)
t_{BS2}	= t_q * (TS2 + 1)
NominalBitTime	= t_q + t_{BS1} + t_{BS2} = t_q * (3 + TS1 + TS2)
CAN Baudrate	= 1 / NominalBitTime

A.3 ESDACC Bus Timing Register for CAN FD

With the introduction of ***canSetBaudrateX()*** and the possibility to define the bit timing register in a CAN controller independent, canonical way it is not necessary to describe the internal register layout of the ESDACC bit rate register in the CAN FD operation mode.

Annex B: Bus Error Code

B.1 SJA1000 and ESDACC

For detailed bus diagnostic the ESDACC implements a superset of the NXP SJA1000 (see /5/) *Bus Error Code* which contains information about the error type (see chapter 3.2), the location in the CAN message bit stream and the information if the error was detected by the CAN controller during the transmission or reception of CAN data.

The table below contains an overview on the byte layout.

Bit	Description	Details
7	Error Code (Class)	00 = Bit Error 01 = Form Error 10 = Stuff Error 11 = Other Error
6		
5	Direction	1 = Error during reception 0 = Error during transmission
4	Bit Stream Position	Position in the bit stream of the CAN frame as the error was detected. Refer to table 39 for further details.
3		
2		
1		
0		

Table 38: Bus Error Code

The knowledge of the error class and the position within the CAN bit stream in combination with the I/O direction makes a detailed error analysis possible. The following two tables contain an overview of possible error indications, the effect on the respective error counter and a description of the error reason separated for occurrence during frame reception and transmission.

Type	Bit 0..4	Bit position	Cnt	Description
Stuff	0 0 0 1 0	ID.28 to ID.21	+1	More than 5 consecutive bits with same level received
	0 0 1 1 0	ID.20 to ID.18		
	0 0 1 0 0	SRR Bit		
	0 0 1 0 1	IDE Bit		
	0 0 1 1 1	ID.17 to ID.13		
	0 1 1 1 1	ID.12 to ID.5		
	0 1 1 1 0	ID.4 to ID.0		
	0 1 1 0 0	RTR Bit		
	0 1 1 0 1	Reserved bit 1		
	0 1 0 0 1	Reserved bit 0		
	0 1 0 1 1	Data length code		
	0 1 0 1 0	Data field		
	0 1 0 0 0	CRC sequence		
	1 1 0 0 0	CRC delimiter		
	1 1 1 0 1	Reserved bit 0 (FD)		
	1 1 1 1 0	BRS Bit		
	1 1 1 1 1	ESI Bit		
Form	1 1 0 0 0	CRC delimiter	+1	Rx dominant
	1 1 0 1 1	Acknowledge delimiter		Rx dominant → Indication of a CRC error
	1 1 0 1 0	End of frame		Rx dominant within first 6 bits
	1 0 1 1 1	Error delimiter		Rx dominant within first 7 bits
Bit	1 1 0 0 1	Acknowledge slot	+1	Rx recessive but should be dominant → Transmitter can't write dominant bit
	1 0 0 0 1	Active error flag	+8	
	1 1 1 0 0	Overload flag	+8	
Other	1 1 0 1 0	End of frame	+0	Rx dominant in last bit → Data retransmission is possible
	1 0 0 1 0	Intermission	+0	Rx dominant → Overload flag will be sent by receiver
	1 0 0 1 1	Tolerate dominant bits	+8	Rx dominant in first bit upon error flag or for more than 7 bits upon error or overload flag
	1 0 1 1 1	Error delimiter	+0	Rx dominant in last bit of delimiter → Overload flag will be sent by receiver
	1 0 1 0 0	Stuff Count		

Table 39: Error detection and indication during reception

Type	Bit 0..4	Bit position	Cnt	Description
Bit	0 0 0 1 1	Start of Frame	+8	Failed to write dominant bit → Rx is recessive.
	0 0 0 1 0	ID.28 to ID.21		
	0 0 1 1 0	ID.20 to ID.18		
	0 0 1 0 0	SRR Bit		
	0 0 1 0 1	IDE Bit		
	0 0 1 1 1	ID.17 to ID.13		
	0 1 1 1 1	ID.12 to ID.5		
	0 1 1 1 0	ID.4 to ID.0		
	0 1 1 0 0	RTR Bit		
	0 1 1 0 1	Reserved bit 1		
	0 1 0 0 1	Reserved bit 0		
	0 1 0 1 1	Data length code		
	0 1 0 1 0	Data field		
	0 1 0 0 0	CRC sequence		
	1 1 0 0 0	Active error		
	1 1 1 0 0	Overload flag		
Stuff	0 0 0 1 0	ID.28 to ID.21	+1	Rx dominant
	0 0 1 1 0	ID.20 to ID.18		Rx dominant → Indication of a CRC error
	0 0 1 0 0	SRR Bit		Rx dominant within first 6 bits
	1 0 1 1 1	Error delimiter		Rx dominant within first 7 bits
	0 0 1 0 1	IDE Bit		
	0 0 1 1 1	ID.17 to ID.13		
	0 1 1 1 1	ID.12 to ID.5		
	0 1 1 1 0	ID.4 to ID.0		
	0 1 1 0 0	RTR Bit		
Bit	1 1 0 0 1	Acknowledge slot	+1	Rx recessive but should be dominant → Transmitter can’t write dominant bit
	1 0 0 0 1	Active error flag	+8	
	1 1 1 0 0	Overload flag	+8	
Other	1 1 0 1 0	End of frame	+0	Rx dominant in last bit → Data retransmission is possible
	1 0 0 1 0	Intermission	+0	Rx dominant → Overload flag will be sent by receiver
	1 0 0 1 1	Tolerate dominant bits	+8	Rx dominant in first bit upon error flag or for more than 7 bits upon error or overload flag
	1 0 1 1 1	Error delimiter	+0	Rx dominant in last bit of delimiter → Overload flag will be sent by receiver
	1 0 1 1 0	Passive error flag		

Type	Bit 0..4	Bit position	Cnt	Description
	1 0 1 0 0	Stuff Count		
	1 1 1 0 1	Reserved Bit 0 (FD)		
	1 1 1 1 0	BRS Bit		
	1 1 1 1 1	ESI Bit		

Table 40: Error detection and indication during transmission