



NTCAN

Part 2: Installation, Configuration and Firmware Update



Installation Guide



NOTE

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document and assumes no responsibility for any errors that may appear in this document. In particular descriptions and technical data specified in this document may not be constituted to be guaranteed product features in any legal sense.

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

All rights to this documentation are reserved by **esd**. Distribution to third parties, and reproduction of this document in any form, whole or in part, are subject to **esd**'s written approval.

© 2026 esd electronics gmbh, Hannover

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

Trademark Notices

.NET: The Microsoft®.NET logo is a registered trademark of Microsoft Corporation in the United States and/or other countries.

CiA® and CANopen® are registered community trademarks of CAN in Automation e.V..

Linux® is the registered trademark of Linus Torvalds in the United States and/or other countries.

Microsoft®, Windows®, Windows Vista®, and the Windows logo are registered trademarks of Microsoft Corporation in the United States and/or other countries.

QNX® and Neutrino® are registered trademarks of QNX Software Systems Limited, and are registered trademarks and/or used in certain jurisdictions.

Solaris™ is a trademark of Sun Microsystems, Inc. in the United States and in other countries.

IntervalZero RTX64® and MaxRT wRTOS® are a registered trademark of IntervalZero, Inc.

TENASYS® and INTIME® are registered trademarks of TenAsys Corporation.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

VxWorks® is a registered trademark of Wind River Systems, Inc.

All other trademarks, product names, company names or company logos used in this manual are reserved by their respective owners.

Document file:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\C.2001.21_NTCAN\Installation\NTCAN_Installation_en_51.docx
Date of print:	2026-09-07
Document no.:	QM.1012.C.2001.22

Products covered by this document.

CAN-Driver / SDK	(Driver) Revision
CAN SDK for Windows	4.x.y
Windows 10 (32/64-Bit)	2.x.y
Windows 11 (64-Bit)	3.x.y
Windows 10/11 on Arm	4.x.y
Linux Driver (32-/64-Bit)	3.x.y 4.x.y
VxWorks 5.x/6.x (Non-VxBus)	2.x.y
VxWorks 6.x (VxBus)	3.x.y
VxWorks 7.x (VxBus GEN2)	4.x.y
QNX® 6.x / 7.x / 8.x Driver	3.x.y / 4.x.y
RTX64 Driver	4.x.y
MaxRT wRTOS Driver	4.x.y
INtime Driver	4.x.y

CAN-Hardware	Order No.
EtherCAN	C.2050.xx
EtherCAN/2	C.2051.xx
EtherCAN/3-FD	C.2055.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-PCle/200	C.2042.xx
CAN-PCle/400	C.2043.xx
CAN-PCle/402	C.2045.0x

CAN-Hardware	Order No.
CAN-PCle/402-FD	C.2045.xx
CAN-PCleMini/402	C.2044.xx
CAN-PCleMini/402-FD	C.2044.xx
CAN-PCleMiniHS/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
CPCIsreal-CAN/402	I.3001.04
CPCIsreal-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
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
5.1	4.3	Added installation instructions for IntervalZero MaxRT wRTOS and removed installation instructions for RTX (32-bit)	2026-09-03
5.0	2.2	New chapter to describe installation for Windows on Arm	2025-11-19
	-	Chapters with description of legacy OS systems deleted.	2025-11-19
4.9	2.3	Revised chapter of Windows CAN Driver Configuration	2025-06-16
	2.7	EtherCAN installation chapter completely revised.	2025-06-13
	4.2	QNX® installation chapter completely revised.	2025-04-25
	4.3.4.1	Described extended RTX64 driver options.	2025-06-11
	4.3.4.2	New chapter to describe driver priority layout on RTX64.	2025-06-11
	4.3.4.3	New chapter to describe 64-bit DMA support on RTX64.	2025-06-11
	4.3.4.4	New chapter to describe the multi-core support on RTX64.	2025-06-11
	4.4.4.3	New chapter to describe 64-bit DMA support on INtime.	2025-04-24
	4.5.3	Moved the content covering QNX® 4.x into the chapter for legacy RTOS versions.	2025-04-25
4.8	2.9	Moved the content covering Windows 8.x to the chapter for legacy Windows versions.	2024-01-02
	4.4	New chapter which covers the support for the real-time OS INtime®	2024-01-08
	4.6	Moved the content covering On-Time RTOS-32 into the chapter for legacy RTOS versions.	2024-01-02
4.7	1.5.2	Updated USB Hardware ID table	2021-10-22
	2.4.4	New chapter to enforce driver installation.	2021-10-25
	2.8.1	Revised document for the latest changes in the Windows driver code signing policies.	2021-10-22
	2.9	Moved the chapters for Windows XP, Vista and 7 to the legacy Windows versions.	2021-10-22
	3.2	Moved the content for PowerMAX OS, Solaris, SGI-IRIX and AIX to the chapter for legacy UNIX versions.	2021-10-22
	5.1	Extended table 18 for CAN-USB/2V2 and CAN-USB/3-FD	2021-10-25
	N/A	Editorial changes for Windows 11 support.	2021-10-22
4.6	5.1	Revised and extended description of update process	2018-09-19
	N/A	Editorial changes	2019-07-25
4.5	1.5.1	Updated PCI Hardware ID table	2018-01-22
	1.5.2	Updated USB Hardware ID table	2017-10-30
	2.8	Completely revised to cover Microsoft's SHA-1 deprecation policy.	2018-05-28
	4.1	Added VxWorks 7.0 support	2018-06-07
	4.2.1	Updated for QNX®7	2018-05-08
	N/A	Editorial changes	2018-06-11

Rev.	Chapter	Changes versus previous version	Date
4.4	N/A	Editorial changes	2017-06-08
4.3	2	Description to preinstall device drivers on Windows.	2016-02-08
	2.1	Description of device driver installation on Windows 10.	2016-02-09
	2.2.1	Device Driver Configuration completely revised and extended.	2016-02-12
	2.4	New chapter to troubleshoot Windows driver installation.	2016-03-11
	2.9	Moved Windows 2000 to the legacy Windows versions.	2016-02-09
	N/A	Editorial changes	2016-04-06
4.2	2	Added description of WDF based CAN device driver.	2014-12-15
	2.2.2	Revised description of CAN driver property sheet in device manager with new chapter how to configure IRQ affinity	2015-01-23
	N/A	Added CAN-USB/400 specific information	2015-01-23
	N/A	Editorial changes	2014-12-07
4.1	4.4	Revised RTX installation instructions with regard to RTX64 2014.	2014-11-24
4.0	N/A	The complete document is revised, large parts of the text are rewritten, and the drawings are updated.	2013-08-12
	1	Introduction completely rewritten.	2013-08-12
	2.1	Description of device driver installation on Windows 8.	2013-08-12
	2.9.1	Description of device driver installation on Windows 7.	2013-08-12
	2.8	New chapter about Windows Digital Signatures.	2013-08-12
	3.1	New introduction for Linux driver.	2013-08-12
	3.1.2.2	Linux CAN-Module IDs: Modules inserted (AMC-CAN4, CAN-PCI/400, CAN-PCIe/400, CPCI-CAN/400, PMC-CAN/400), Modules deleted (CAN-USB/2, VME-CAN2, VME-CAN4)	2013-08-12
	3.1.2.3	Unpacking of Linux TGZ archive for drivers released after 07-2012 inserted	2013-08-12
	-	Chapter 'Installation of Linux drivers < 3.x.x.x deleted'.	2013-08-12
	3.1.3	New chapter with installation for Linux CAN (aka SocketCAN)	2013-08-12
	3.1.4	New chapter for Linux EtherCAN/2 installation.	2013-08-12
	4.1	Installation on VxWorks completely rewritten.	2013-08-12
	4.2	QNX®6: Program for setting the PCI Class inserted. CAN interface family C400 support inserted. Additional commands for Resource Manager.	2013-08-12
	4.4	Installation on RTX completely revised and extended with installation instructions for RTX64.	2013-08-12
	5	Description of firmware update completely revised and combined in one chapter for all platforms.	2013-08-12

Technical details are subject to change without further notice.

Typographical Conventions

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

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

The following indicators are used to highlight noticeable descriptions.



Note

Notes to point out something important or useful.



Attention!

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
AMC	Advanced Mezzanine Cards
API	Application Programming Interface
BSD	Berkley Software Distribution
BSP	Board Support Package
CAN	Controller Area Network
CPU	Central Processing Unit
CiA	CAN in Automation
COTS	Commercial off-the-shelf
CPCI	Compact Peripheral Component Interconnect (Computer Bus)
CPCIs _{serial}	Compact Peripheral Component Interconnect Serial (Computer Bus)
CRC	Cyclic Redundancy Check
DLC	Data Length Code
DLL	Dynamic Link Library
DNS	Domain Name Service
EFF	Extended Frame Format
ACC	Advanced CAN Controller
FIFO	First-In-First-Out
FTP	File Transfer Protocol
FW	Firmware
HW	Hardware
I/O	Input/Output
ISA	Industry Standard Architecture (Computer Bus)
IPC	Interprocess Communication
IRIG	Inter-range Instrumentation Group
LSB	Least Significant Bit
LSW	Least Significant Word
MCU	Micro Controller Unit
MSB	Most Significant Bit
MSW	Most Significant Word
NVRAM	Non-Volatile Random Access Memory
N/A	Not Applicable
OS	Operating System
PCI	Peripheral Component Interconnect (Computer Bus)
PCIe	Peripheral Component Interconnect Express (Computer Bus)
PIC	Programmable Interrupt Controller
PMC	PCI Mezzanine Card
PnP	Plug and Play
RAM	Random Access Memory
REC	Receive Error Counter
RTSS	Real-Time Sub-System
SDK	Software Development Kit
SMP	Symmetric Multiprocessor
SoC	System on Chip
SFF	Standard (Base) Frame Format
SW	Software
TEC	Transmit Error Counter
USB	Universal Serial Bus
UP	Uniprocessor
URL	Universal Resource Locator
WDM	Windows Driver Model
WDF	Windows Driver Foundation
WHQL	Windows Hardware Quality Lab
WoW64	Windows 32-bit on Windows 64-bit
XMC	Express Mezzanine Card

Reference

- /1/ esd electronics gmbh, *NTCAN-API Application Developers Manual*, Revision 5.11, 2025
- /2/ ISO 11898-1, *Road vehicles – Controller area network (CAN) – Data link layer and physical signalling*, 2024
- /3/ esd electronic system design gmbh, *EtherCAN/2 Hardware Manual*, Revision 2.0, 2021
- /4/ esd electronic system design gmbh, *CAN-Wiring*, Revision 4.6, 2022
- /5/ esd electronic system design gmbh, *EtherCAN/3 Manual*, Revision 1.1, 2025

Table of Contents

1	Introduction	13
1.1	Scope	13
1.2	Overview	14
1.3	Terminology	15
1.4	CAN Interface Families	17
1.5	Hardware IDs	18
1.5.1	PCI / PCIe / PCIe Mini / CPCI / CPCIserial / PMC / XMC	18
1.5.2	USB	20
1.5.3	Ethernet	20
1.6	Software Deployment	21
1.6.1	Windows	21
1.6.2	Linux / Unix	21
1.6.3	Real-Time Operating Systems	21
2	Windows®	22
2.1	Windows 10 / 11 (x86 / x64)	26
2.1.1	Hardware-First Driver Installation	27
2.1.2	Software-First Driver Installation	32
2.1.3	Driver Lifecycle Management	32
2.2	Windows 10 / 11 (Arm64)	33
2.2.1	Installation	34
2.2.2	Uninstallation	38
2.3	Configuration	39
2.3.1	Device Driver	39
2.3.1.1	Standard Settings	39
2.3.1.2	Expert Settings	42
2.3.1.3	Device Specific Settings	44
2.3.2	System	45
2.3.2.1	Power Management	45
2.3.2.2	Interrupt Affinity	45
2.4	Device Driver Preinstallation	46
2.4.1	Driver Staging	46
2.4.2	Driver Installation for Non-Administrators	47
2.5	Troubleshooting Driver Installation	49
2.5.1	Error Code 31	50
2.5.2	Error Code 39	51
2.5.3	Error Code 52	51
2.5.4	Best Driver already Installed	52
2.6	Device Driver Lifecycle Management	54

2.6.1	Driver Update	54
2.6.2	Driver Rollback	55
2.6.3	Driver Uninstall	56
2.7	EtherCAN Driver	57
2.7.1	Installation	57
2.7.2	Configuration	58
2.7.3	Uninstall	59
2.8	CAN SDK	60
2.8.1	Setup Command Line Parameter	60
2.8.2	Installation Options	61
2.8.3	Uninstall	61
2.8.4	IDE Integration	62
2.9	Digital Signatures	63
2.9.1	Overview	63
2.9.2	Driver Installation	65
2.9.3	Software Installation	66
2.9.4	Digital Signature Verification	67
3	Unix® Operating Systems	69
3.1	Linux®	69
3.1.1	CAN Board Support Overview	71
3.1.2	NTCAN Driver	73
3.1.2.1	Files of the Linux Package	74
3.1.2.2	CAN-Module-ID and Default Parameters of the Driver	76
3.1.2.3	Installation	77
3.1.3	Linux CAN Driver (aka SocketCAN)	83
3.1.3.1	Integration	83
3.1.3.2	Installation	83
3.1.3.3	Configuration	84
3.1.3.4	Restrictions	85
3.1.4	EtherCAN and EtherCAN/2	87
3.1.4.1	Installation	88
3.1.4.2	Configuration	89
3.1.4.3	Miscellaneous	90
4	Real-Time Operating Systems	91
4.1	VxWorks®	91
4.1.1	CAN Board Support Overview	92
4.1.2	Driver Integration	94
4.1.2.1	VxWorks 5.x	94
4.1.2.2	VxWorks 6.x	95
4.1.2.3	VxWorks 7.x	96
4.1.3	Driver Configuration	98

4.1.3.1	VxWorks 5.x	98
4.1.3.2	VxWorks 6.x (Non-VxBus)	101
4.1.3.3	VxWorks 6.x (VxBus)	105
4.1.3.4	VxWorks 7.x (VxBus GEN2)	108
4.1.4	Driver Start	111
4.1.4.1	VxWorks 5.x	111
4.1.4.2	VxWorks 6.x (Non-VxBus)	111
4.1.4.3	VxWorks 6.x/7.x (VxBus)	112
4.1.5	Miscellaneous	113
4.1.5.1	Unresolved Symbols Building the VxWorks Image	113
4.1.5.2	Number of Available NTCAN Handles	113
4.1.5.3	Test Program 'canTest'	113
4.1.5.4	Unexpected Behaviour of Software Timestamps	113
4.1.5.5	Correct Interpretation of Error Codes Returned by the Driver	114
4.1.5.6	Support of the CAN Extended Frame Format	115
4.1.6	Troubleshooting Hints	116
4.1.6.1	Where to Implement Needed Configuration Changes	117
4.1.6.2	Public Interface of the Version 2.x Driver Core	117
4.1.6.3	Public Interface of the VME-CAN4 Driver Core	118
4.1.6.4	Address Translation and Board Access Issues	118
4.1.6.5	Interrupt Connection Issues	124
4.1.6.6	VxBus Driver Prerequisites	127
4.2	QNX®	130
4.2.1	CAN Board Support Overview	131
4.2.2	Driver Integration	131
4.2.3	Driver Installation	132
4.2.4	Driver Start	133
4.2.5	Driver Configuration	134
4.2.5.1	Command Line Parameter	134
4.3	IntervalZero RTX64® and MaxRT wRTOS®	136
4.3.1	Driver Integration	137
4.3.2	Driver Installation	138
4.3.2.1	RTX64	138
4.3.2.2	MaxRT wRTOS	141
4.3.3	Driver Start	144
4.3.4	Driver Configuration	145
4.3.4.1	Device Driver Options	145
4.3.4.2	Priority Layout	146
4.3.4.3	64-Bit DMA memory	147
4.3.4.4	Multi Core Support	148
4.3.5	Driver Unload	149

4.3.6	Miscellaneous	150
4.3.6.1	Application Development	150
4.3.6.2	Example Application	150
4.4	TenAsys® INtime®	151
4.4.1	Driver Integration	152
4.4.2	Driver Installation	153
4.4.2.1	INtime for Windows.....	153
4.4.3	Driver Start	157
4.4.4	Driver Configuration	158
4.4.4.1	Command Line Parameter	158
4.4.4.2	Priority Layout.....	159
4.4.4.3	64-Bit DMA memory	160
4.4.5	Driver Unload.....	161
4.4.6	Miscellaneous	161
4.4.6.1	Application Development	161
4.4.6.2	Example Application	161
5	Firmware Update Application.....	162
5.1	Updating the Firmware	163
5.2	Switch between CAN 2.0A and CAN 2.0B Mode.....	165

1 Introduction

This document describes the device driver installation and configuration process of the cross-platform architecture for **esd** Controller Area Network (CAN) hardware as well as the steps to update the firmware of active CAN boards.

Within this document this architecture and its usual implementation as a combination of a device driver and a library is referred to 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).

The CAN bus is specified in /2/.

**Attention:**

Before you start with the software installation refer to the CAN board hardware manual how to install the device mechanically and electrically in your system.

Please refer to the *CAN-Wiring* document [/4/](#) for further information on the wiring of the CANbus, cable selection, correct termination, etc.

The documents are either located on the CD which comes with your hardware or can be downloaded from the **esd** web site (<https://www.esd.eu/>)

1.1 Scope

This document covers the CAN board and operating system specific installation of the NTCAN architecture which usually consists of an OS and CAN hardware specific device driver and a (shared) library which exports the application interface to integrate CAN I/O into an application as well as required files to use the API in your application (header files, startup code, ...).



The NTCAN architecture and the API (which is identical for all platforms) to develop CAN based applications is described in the first part of the CAN-API documentation called 'NTCAN, Part 1: Application Developers Manual' **/1/**.

Introduction

The NTCAN architecture is implemented for the following desktop, embedded, real-time and UNIX operating systems.

Windows	UNIX	Real-Time
Active		
Windows 11 (64 Bit) Windows 10 (32-/64-Bit)	Linux (32-/64-Bit)	QNX 6.x / 7.x / 8.x VxWorks 5.x / 6.x / 7.x RTX64 4.x / MaxRT wRTOS 1.x INtime 6.x / 7.x
Legacy		
Windows 9x/ME Windows NT Windows 2000 Windows XP (32-/64-Bit) Windows Vista (32-/64-Bit) Windows 7 (32-/64-Bit) Windows 8 / 8.1 (32-/64-Bit)	AIX PowerMAX OS SGI-IRIX 6.5 Solaris	LynxOS On-Time RTOS-32 QNX4 RTX

Table 1: 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.

The installation instructions for legacy operating systems are no longer included in this manual! You can find the latest instructions for installation on these operating systems in the old version 4.9 of this manual on our website:

[NTCAN Part 2: Installation Guide \(Legacy OS support\), Rev. 4.9.](#)

1.2 Overview

The document is made up of 5 main chapters with the following topics:

Chapter 1 contains a general overview of the structure of this manual.

Chapter 2 describes the installation process on the various versions of Windows®.

Chapter 3 covers the installation process on Unix® based operating systems (inclusive Linux®).

Chapter 4 describes the device driver integration in various (embedded) real time OS.

Chapter 5 provides information about the firmware update process of active CAN boards.

1.3 Terminology

Within this manual you will encounter the following terms:

Base Frame Format	CAN messages with 11-bit CAN-IDs according to /2/
Board Support Package	A Board Support Package (BSP) is the common name for the hardware specific code which is necessary for an operating system to support an embedded board. It typically consists at least of the code to initialize the hardware to a point to load and start the operating system and all device drivers to support the on-board interfaces.
CAN	Controller Area Network 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.
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 CAN Interface or an embedded system with an on-board <i>CAN Controller</i> .
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 <i>System on Chip (SoC)</i> which integrates one or more CAN controllers as external interface.
CAN Device	The (logical) application view to a physical CAN port.
CAN Family	A CAN family describes the group of CAN boards which are handled with the same device driver (see chapter 1.4).
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 <i>Standard Frame Format</i> (11-bit) or the <i>Extended Frame Format</i> (29-bit)
CAN Interface	A CAN interface is a dedicated esd 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.
Driver Store	Repository for device driver introduced with Windows Vista.
Extended Frame Format	CAN messages with 29-bit CAN-IDs according to /2/
Hot Plugging	Hot Plugging describes the capability of a hardware to be connected or replaced without shutting down the system.
INTx	INTx or Legacy Interrupts describe the method to signal interrupts with the help of dedicated pins in contrast to MSI .
Legacy Interrupt	See INTx .

IRIG B	Time code format used to provide time-of-day information to communication systems which have to correlate data (reception) with time.
MSI	Message Signaled Interrupts are defined in the PCIe standard as a method to signal interrupts as an in-band message instead of using dedicated pins. One of the advantages compared to traditional INTx signaling is that MSIs are never shared.
Plug and Play (PnP)	Hardware capability to support automatic device detection and configuration without user intervention. Examples for PnP capable buses are PCI, PCIe, CPCI, PMC, PCI104 and USB.
RTSS	Real-Time Sub-System for Windows provide by RTX/RTX64.
Standard Frame Format	Same as <i>Base Frame Format</i> .
UAC	User Account Control is a security infrastructure introduced with Microsoft Windows Vista.
VxBus	Software infrastructure to integrate device drivers in the real time operating system VxWorks with minimal BSP support.
VxD	Device driver model of Windows 9x/ME.
WDM	The Windows Driver Model was introduced with Windows 98/2000 as a common driver model to replace VxD and the Windows NT Driver Model with a unified API.
WDF	The Windows Driver Foundation is a WDM based driver model (framework) introduced in 2006 which provides a robust object-based interface for device drivers.
WHQL	Windows Hardware Quality Labs (WHQL) Testing is Microsoft's testing process which involves running a series of tests on third-party hardware/software. For device driver passing the WHQL tests, Microsoft creates a digitally signature.
WoW64	WoW64 (Windows 32-bit on Windows 64-bit) is a subsystem included on all 64-bit versions of Windows that is capable of running 32-bit applications.

1.4 CAN Interface Families

A single **esd** NTCAN device driver supports in most cases more than one CAN interface. For this reason, the operating system specific installation and configuration instructions of a device driver in this manual are usually intended for complete CAN interface family covering several CAN interfaces. Each CAN family device driver is assigned a unique signature which is used in log messages, etc. to distinguish drivers of different CAN interface families on the same host system and a device driver name which follows one of two possible naming conventions.

The table below gives an overview on the CAN interfaces or boards that are covered by a CAN family device driver, their assigned signatures and device driver names:

CAN Interface Families	Family Name	Signature	Driver Name (Naming Convention I)	Driver Name (Naming Convention II)
CAN-PCI/200, CAN-PCI/266 CAN-PCle/200, CAN-PC104/200 CPCI-CAN/200, PMC-CAN/266	C200	C200	c200	pci200-sja1000
CAN-PCI/331, CPCI-CAN/331, PMC-CAN/331	C331	C331	c331	pci331-i20
CAN-PCI/360, CPCI-CAN/360	C360	C360	c360	pci360-i20
CAN-PCI/400, CAN-PCle/400, CPCI-CAN/400, PMC-CAN/400	C400	C400	c400	pci400-esdacc
CAN-PCI/402, CAN-PCle/402, CAN-PCIMini/402, CPCI-CAN/402, CPCIsreal-CAN/402, PMC-CAN/402, XMC-CAN/402, CAN-M.2/402-2-FD, CAN-PCleMiniHS/402 + CAN FD enable derivatives XMC-CAN/402-FD-LIN-DIO, PMC-CAN/402-FD-LIN-DIO, XMC-CAN/404-FD-LIN-DIO,	C402	C402	c402	pcie402-esdacc
CAN-PCI/405	C405	C405	c405	pci405-pcimsgx
CAN-USB/Mini	USB1	U331	usb331	N/A
CAN-USB/Micro, CAN-USB/2 CAN-USB/2V2, CAN-AIR/2	USB2	U2292	usb2292	N/A
CAN-USB/3-FD	USB3	U3FD	usb3fd	N/A
CAN-USB/400 CAN-USB/400-IRIG-B	U400	U400	u400	N/A
CAN-ISA/200, CAN-PC104/200, Memory mapped NXP SJA1000	C200I	C200I	c200i	isa200-sja1000
CAN-ISA/331, CAN-PC104/331	C331I	C331I	c331i	isa331-i20
CAN-VME2	CAN2	ICAN2	ican2	N/A
CAN-VME4	CAN4	ICAN4	ican4	N/A

Table 2: Overview of the CAN Interface Families

1.5 Hardware IDs

To identify and **esd** (P'n'P) CAN interface in a system or to troubleshoot an installation it is sometimes helpful to know it's bus specific hardware ID. This chapter gives an overview on the hardware IDs for the **esd** Commercial off-the-shelf (COTS) CAN interfaces.

The knowledge about the hardware IDs helps troubleshooting installations and interpreting log messages.

1.5.1 PCI / PCIe / PCIe Mini / CPCI / CPCIs serial / PMC / XMC

An **esd** CAN interface attached to the PCI, PCIe, PCIe Mini, CPCI, CPCIs serial, PMC or XMC bus can be unambiguously identified by the ID pair for the main chip (Vendor ID and Device ID) and a vendor specific ID pair for the device (Subsystem Vendor ID and Subsystem ID). Each ID is 16-bit and the Vendor and Subsystem Vendor IDs are assigned by the PCI SIG.

The PCI SIG also defines an ID pair of of class and subclass for the device type. The class and subclass ID are 8-bit each. As a class dedicated to CAN interfaces was not introduced before PCI Local Bus Specification Revision 2.3 you will find **esd** CAN interfaces utilize three different classes.

<i>PCI Class</i>	<i>PCI Subclass</i>	<i>Description</i>
0x06	0x80	Other Bridge Device / Other System Peripheral
0x02	0x80	Other Network Controller
0x0C	0x09	CAN bus

Table 3: PCI device classes assigned to esd CAN interfaces



Attention:

Some operating systems and BIOS versions are known to not assign I/O resources to PCI devices with the class 0x06 and subclass 0x80 (Other Bridge Device). Please contact **esd** in these cases for a solution.

The table below gives an overview on the COTS **esd** CAN interfaces, their local bus IDs and device classes.

CAN Interface	Vendor ID	Device ID	Subsystem Vendor ID	Subsystem ID	Class / Subclass
CAN-PCI/200	0x10B5	0x9050	0x12FE	0x0004	0x06 / 0x80
CAN-PCI/266	0x10B5	0x9056	0x12FE	0x0009	0x06 / 0x80
CAN-PCle/200	0x10B5	0x9056	0x12FE	0x0200	0x0C / 0x09
CAN-PCI104/200	0x10B5	0x9030	0x12FE	0x0501	0x0C / 0x09
CPCI-CAN/200	0x10B5	0x9030	0x12FE	0x010B	0x02 / 0x80
PMC-CAN/266	0x10B5	0x9056	0x12FE	0x000E	0x06 / 0x80
CAN-PCI/331 CPCI-CAN/331 PMC-CAN/331	0x10B5	0x9050	0x12FE	0x0001	0x06 / 0x80
PMC-CAN/331 (3.3 V)	0x10B5	0x9030	0x12FE	0x000C	0x06 / 0x80
CAN-PCI/360	0x10E3	0x0860	0x12FE	0x0000	0x06 / 0x80
CPCI-CAN/360	0x10E3	0x0860	0x12FE	0x0007	0x06 / 0x80
CAN-PCI/400-2 CAN-PCI/400-4	0x10B5	0x9056	0x12FE	0x0200	0x0C / 0x09
CAN-PCle/400-2 CAN-PCle/400-4	0x10B5	0x9056	0x12FE	0x0201	0x0C / 0x09
CPCI-CAN/400-2	0x10B5	0x9056	0x12FE	0x0141	0x0C / 0x09
CPCI-CAN/400-4	0x10B5	0x9056	0x12FE	0x0142	0x0C / 0x09
CPCI-CAN/400-4I	0x10B5	0x9056	0x12FE	0x0143	0x0C / 0x09
CPCI-CAN/400-2-PXI	0x10B5	0x9056	0x12FE	0x0144	0x0C / 0x09
CPCI-CAN/400-4-PXI	0x10B5	0x9056	0x12FE	0x0145	0x0C / 0x09
CPCI-CAN/400-4I-PXI	0x10B5	0x9056	0x12FE	0x0146	0x0C / 0x09
PMC-CAN/400-4	0x10B5	0x9056	0x12FE	0x04C2	0x0C / 0x09
PMC-CAN/400-4I	0x10B5	0x9056	0x12FE	0x04C3	0x0C / 0x09
CAN-PCle/402, CAN-PCI/402, CPCI-CAN/402, CPCIs-serial-CAN/402, CAN-PCIMini/402	0x12FE	0x0402	0x12FE	0x0401 0x0402 0x0403	0x0C / 0x09
CAN-PCI/402-FD, CAN-PCle/402-FD, CAN-PCIMini/402-FD, CPCIs-serial-CAN/402-FD, PMC-CAN/402-FD, XMC-CAN/402-FD CAN-M.2/402-2-FD CAN-PCleMiniHS/402 XMC-CAN/402-FD-LIN-DIO PMC-CAN/402-FD-LIN-DIO	0x12FE	0x0402	0x12FE	0x1402 0x1403	0x0C / 0x09
XMC-CAN/404-FD-LIN-DIO	0x12FE	0x0404	0x12FE	0x1404	0x0C / 0x09
CAN-PCI/405	0x1014	0x0156	0x12FE	0x0008	0x02 / 0x80

Table 4: Hardware IDs for CAN PCI, PCIe, PCIe Mini, CPCI, CPCIs-serial and PMC bus interfaces

1.5.2 USB

An **esd** CAN interface attached to the USB bus can be unambiguously identified by an ID pair which consists of a Vendor ID which is assigned by the USB committee and a vendor specific Device ID. Each ID is 16-bit numerical value.

The table below gives an overview on the COTS **esd** USB CAN interfaces and their unique Ids.

CAN Interface	Vendor ID	Device ID
CAN-USB/Mini	0x0AB4	0x0001
CAN-USB/Micro	0x0AB4	0x0011
CAN-USB/2 (Gen 1 + Gen 2)	0x0AB4	0x0010
CAN-USB/3-FD	0x0AB4	0x0014
CAN-AIR/2	0x0AB4	0x0018
CAN-CBX-AIR/2	0x0AB4	0x0019
CAN-CBX-AIR/3	0x0AB4	0x001B
CAN-USB/400	0x0AB4	0x0400
CAN-USB/400-IRIG-B	0x0AB4	0x0401
CAN-USB/400-FD	0x0AB4	0x0402
CAN-USB/400-FD-IRIG-B	0x0AB4	0x0403

Table 5: Hardware IDs for CAN USB bus Interfaces

1.5.3 Ethernet

An **esd** CAN interface attached to the Ethernet has a unique 48-bit MAC address which consists of a 24-bit vendor ID assigned by the Internet Assigned Number Authority (IANA) and a 24-bit vendor specific part. All Ethernet CAN Gateways use a unique MAC address with the format below:

esd Vendor ID	Device ID
00-02-27	xx-xx-xx

Table 6: MAC addresses of esd Ethernet devices

1.6 Software Deployment

The necessary (device driver) software is usually shipped on a CD/DVD together with the CAN interface or can be downloaded (Windows/Linux) as an archive from the **esd** website (www.esd.eu). The scope of delivery consists at least of a device driver and the NTCAN-API library. For most supported platforms the necessary files to develop NTCAN based applications (header files, example source, documentation, etc.) is also part of the software package.

A CD/DVD usually contains the (device driver) software and HW/SW manuals for all CAN interfaces supported on a platform, the downloadable software distributions only for a CAN interface family.

1.6.1 Windows

If you install from CD/DVD you have all necessary software and can continue reading chapter 2 which describes the installation process for the various versions of the Windows OS.

If you want to download the software from the **esd** website for Windows 10 and later, you need the CAN interface family (see chapter 1.4) and CPU architecture specific (x86 / x64 / ARM64) device driver package. If you want to develop NTCAN based software on the target PC, you must also download and install the CAN SDK for Windows before you proceed reading chapter 2.

Device driver for legacy versions of Windows 9x/ME/NT/2000/XP/Vista/7/8.x can also still be downloaded but they are no longer technically supported.

1.6.2 Linux / Unix

The (device driver) software for all Unix based systems (except Linux) is shipped exclusively on CD/DVD with the hardware and cannot be downloaded from the **esd** website. So usually, you have all necessary software and can continue reading chapter 3 which describes the installation process for the various Unix systems.

Because of the rapid change in the Linux kernel source tree which often leads to problems compiling the device driver for the latest kernel versions the Linux drivers can also be download from the **esd** website. You need a CAN interface family (see chapter 1.4), Linux kernel version (2.4.x/2.6.x/3.x) and CPU architecture specific (32-/64-bit) device driver package which also contains all necessary files to develop NTCAN based software on the target PC before you proceed reading chapter 3.

Some CAN interfaces are supported directly by Linux CAN (aka SocketCAN) which is part of the Linux kernel since version 2.6.25. As the SocketCAN has an individual API **esd** provides a NTCAN wrapper library for SocketCAN so you can use NTCAN based applications with this driver, too (see chapter 3.1.3).

Device driver for other Unix versions but Linux are no longer technically supported.

1.6.3 Real-Time Operating Systems

The (device driver) software for all real-time operating systems is shipped exclusively on CD/DVD with the hardware and cannot be downloaded from the **esd** website. So, you have all necessary software and can continue reading chapter 4 which describes the installation process for the various supported systems.

2 Windows®

This chapter describes the necessary steps to install the (kernel mode) device driver for **esd Plug and Play** (PnP) capable CAN hardware which are connected to a local PnP capable bus (PCI, USB, ...) on the various versions of Microsoft Windows®. There is no device driver support for non-PnP **esd** CAN modules on legacy buses (ISA, PC104, Parallel Port,...).

**Note:**

The standard driver installation is based on the *hardware-first* installation mechanism which involves that the installation of the device driver is triggered by plugging in the **esd** CAN hardware to the system. Starting with Windows Vista Microsoft also integrated the tools for an in-box *software-first* installation mechanism which allows a driver preinstallation without the presence of the hardware during this process.

The CAN device drivers for Windows are either based on the **WDM** (Windows Driver Model) or the **WDF** (Windows Driver Foundation). The WDM was introduced with Windows 2000 and WDF was introduced by Microsoft in 2006 as a robust object-based interface for device drivers on top of WDM. The WDF consists of a Kernel Mode Driver Framework (**KMDF**) and User Mode Driver Framework (UMDF). All WDF based CAN device driver are KMDF driver.

The KMDF comes as a library which is either already part of Windows or is installed together with the CAN device driver once per system with the help of a WDF co-installer. The WDF based CAN driver currently use the KMDF library version 1.9 which is part of Windows since Windows 7.

The installation procedure for WDM and WDF based driver described for the various Windows versions in chapter 2.1 to 2.9 is identical.

**Note:**

The KMDF library version 1.9 does not support Windows 2000 so a **WDF** based CAN device driver requires Windows XP or later to run. Please refer to the WDF driver release notes for further differences between these two driver types.

**Attention:**

The **WDM** based device drivers for the C400, C402 and C405 family do not support a change to a low power state (standby or hibernation) and you must disable this on your Windows system. Please refer to chapter 2.3.2 for details.

The tables on the next pages give you an overview on the various files which are part of a WDM/WDF driver for 32- or 64-bit Windows.

A **WDM** based kernel mode device driver package for 32-bit Windows contains the following files where <drvname> is the device family specific driver name following driver naming convention I (see chapter 1.4).

Filename	Description
x86/<drvname>.sys	The 32-bit WDM based device driver.
x86/ntcan.dll	The 32-bit NTCAN library
x86/canui32.dll	The 32-bit <i>Device Manager</i> property sheet extension
x86/calcan32.dll	The 32-bit protocol helper library
<drvname>.inf	The driver's INF file.
<drvname>.cat	The driver's catalogue file with cryptographic hashes.

Table 7: Files of 32-bit WDM based driver package

A **WDM** based kernel mode device driver package for 64-bit Windows contains the following files where <drvname> is the device family specific driver name following driver naming convention I (see chapter 1.4).

Filename	Description
amd64/<drvname>a.sys	The 64-bit WDM based device driver.
amd64/ntcan64.dll	The 64-bit NTCAN library (Renamed to ntcan.dll during install)
amd64/canui64.dll	The 64-bit <i>Device Manager</i> property sheet extension
amd64/calcan64.dll	The 64-bit protocol helper library
x86/ntcan.dll	The 32-bit NTCAN library (for WoW64)
x86/calcan32.dll	The 32-bit protocol helper library (for WoW64)
<drvname>a.inf	The driver's INF file.
<drvname>a.cat	The driver's catalogue file with cryptographic hashes.

Table 8: Files of 64-bit WDM based driver package

A **WDF** based device driver package for 32-bit Windows contains the following files where **<drvname>** is the device family specific driver name following driver naming convention I (see chapter 1.4).

Filename	Description
x86/<drvname>k.sys	The 32-bit WDF (KMDF) based device driver.
x86/WdfCoInstallerMMmmm.dll	The 32-bit WDF co-installer.*
x86/ntcan.dll	The 32-bit NTCAN library
x86/canui32.dll	The 32-bit <i>Device Manager</i> property sheet extension
x86/calcan32.dll	The 32-bit protocol helper library
<drvname>k.inf	The driver's INF file.
<drvname>k.cat	The driver's catalogue file with cryptographic hashes.

Table 9: Files of 32-bit WDF based driver package

A **WDF** based device driver package for 64-bit Windows contains the following files where **<drvname>** is the device family specific driver name following driver naming convention I (see chapter 1.4).

Filename	Description
amd64/<drvname>ak.sys	The 64-bit WDF (KMDF) based device driver.
amd64/WdfCoInstallerMMmmm.dll	The 64-bit WDF co-installer.*
amd64/ntcan64.dll	The 64-bit NTCAN library (Renamed to ntcan.dll during install)
amd64/canui64.dll	The 64-bit <i>Device Manager</i> property sheet extension
amd64/calcan64.dll	The 64-bit protocol helper library
x86/ntcan.dll	The 32-bit NTCAN library (for WoW64)
x86/calcan32.dll	The 32-bit protocol helper library (for WoW64)
<drvname>ak.inf	The driver's INF file.
<drvname>ak.cat	The driver's catalogue file with cryptographic hashes.

Table 10: Files of 64-bit WDF based driver package

*MM is the major version and mmm is the minor version number of the WDF co-installer. For the WDF version 1.09 this would result in the filename WdfCoInstaller01009.dll.

Chapter 2.3 covers the device driver configuration with the Windows *Device Manager* for Windows 2000 and later.

The mechanisms for a device driver preinstallation introduced by Microsoft with Windows Vista are described in chapter 2.4 and chapter 2.6 covers the post installation aspects of device driver update, device driver rollback and device driver uninstall.

The installation and configuration of the EtherCAN/2 driver software (for all supported versions of Windows) is described in the separate chapter 2.7 as this CAN interface is using a user mode device driver and a hardware specific configuration tool.

Development of NTCAN based applications (on any hardware) requires the installation of the CAN Software Development Kit (SDK) for Windows which comes as separate package. The installation of this package is described for in chapter 2.8.

Chapter 2.9 contains information on device driver and software signing.

**Note:**

This chapter sometimes refers to Microsoft keyboard short-cuts in combination with the *Windows Key* or *WinKey*. This is the key with the Windows logo shown on it and it is usually found between the *Ctrl* and *Alt* keys on your keyboard. A simultaneous key press with another letter is written in this manual as e.g. **WinKey + R** (which does not mean that you must capitalize the letter).

2.1 Windows 10 / 11 (x86 / x64)

This chapter covers the device driver installation for Windows 10 (x86/x64) and Windows 11 (x64).

The installation procedure is identical for the 32-bit and the 64-bit version of Windows 10 but different driver binaries are required. On the 64-bit OS versions all libraries to run 32-bit NTCAN based applications in the *WoW64* subsystem are installed automatically.



Attention!

The device driver for the CAN hardware of the C400, C402 and C405 family currently do not support the *Hybrid Shutdown* mechanism which is the default for these Windows versions. Please refer to chapter 2.3.2 to disable it before you install the device driver.



Note:

Please read the current `Release Notes` file that comes with the software!

Please note the drivers delivered on CD are most likely outdated. We rather recommend checking the `esd electronics gmbh` website for newer driver releases and use these ones to circumvent any problems caused by known and fixed issues before they occur.

2.1.1 Hardware-First Driver Installation

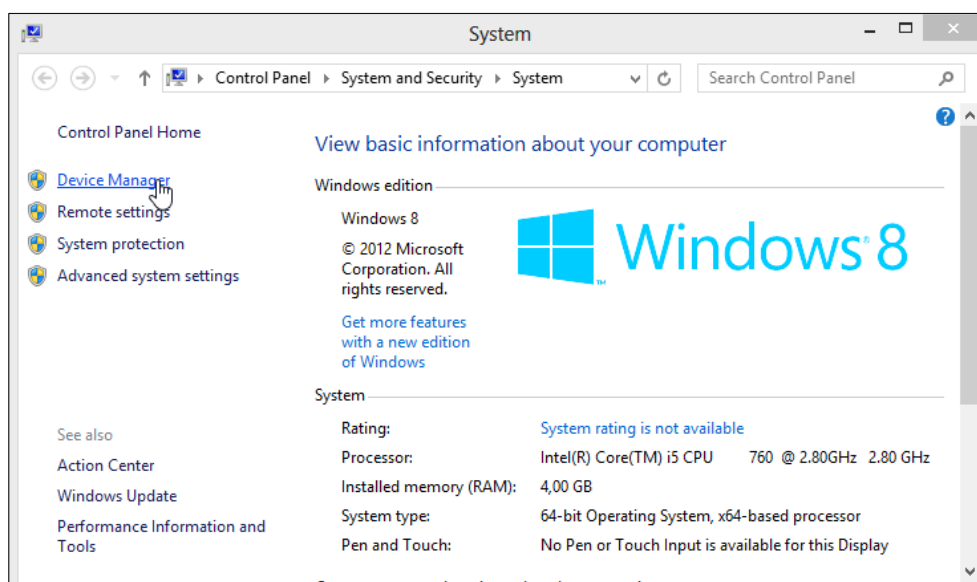
To initiate the device driver installation process, you must connect the CAN module to your system. Depending on the *Hot Plugging* capability of the hardware you might have to shut down Windows for this. Please refer to the CAN module specific hardware manual for advises.



Attention!

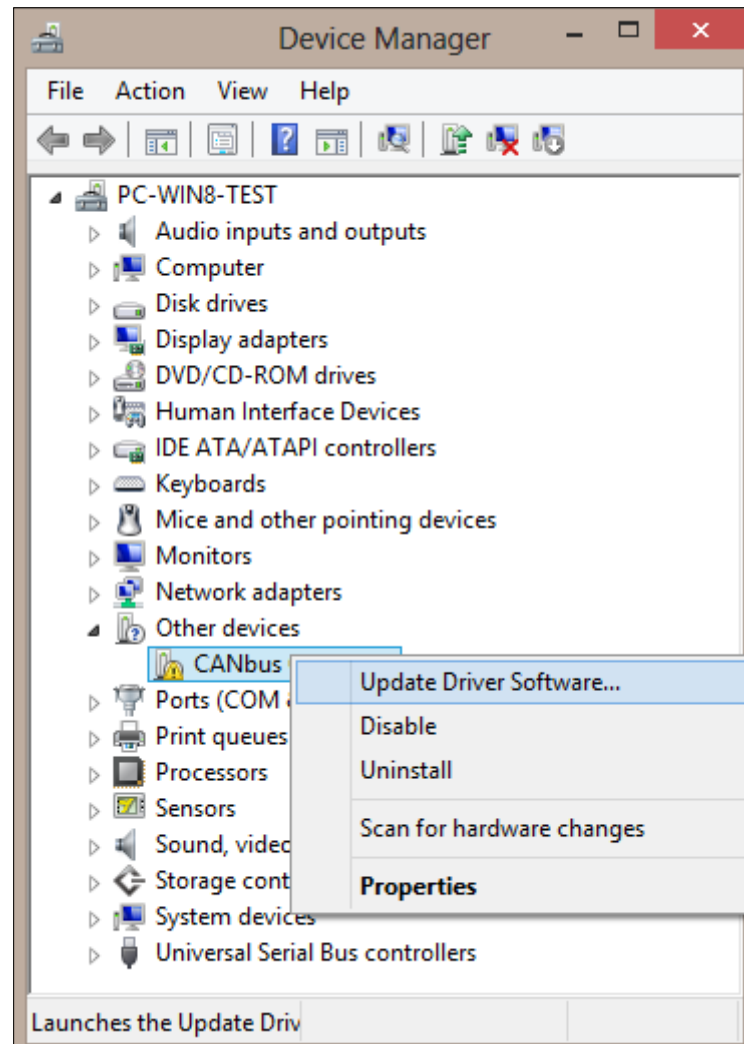
A user which wants to install a device driver must be member of the Administrators group.

Starting with Windows 7 the presence of a new hardware does not automatically start the *Found New Hardware Wizard* to locate and install a driver for the new device with user interaction as in previous versions of Windows. To initiate the interactive device driver installation, you now must open the *Device Manager*. One of the fastest ways to do this is to press **WinKey + Pause/Break** and to select *Device Manager* or your local translation of this tool in the newly opened dialogue box.

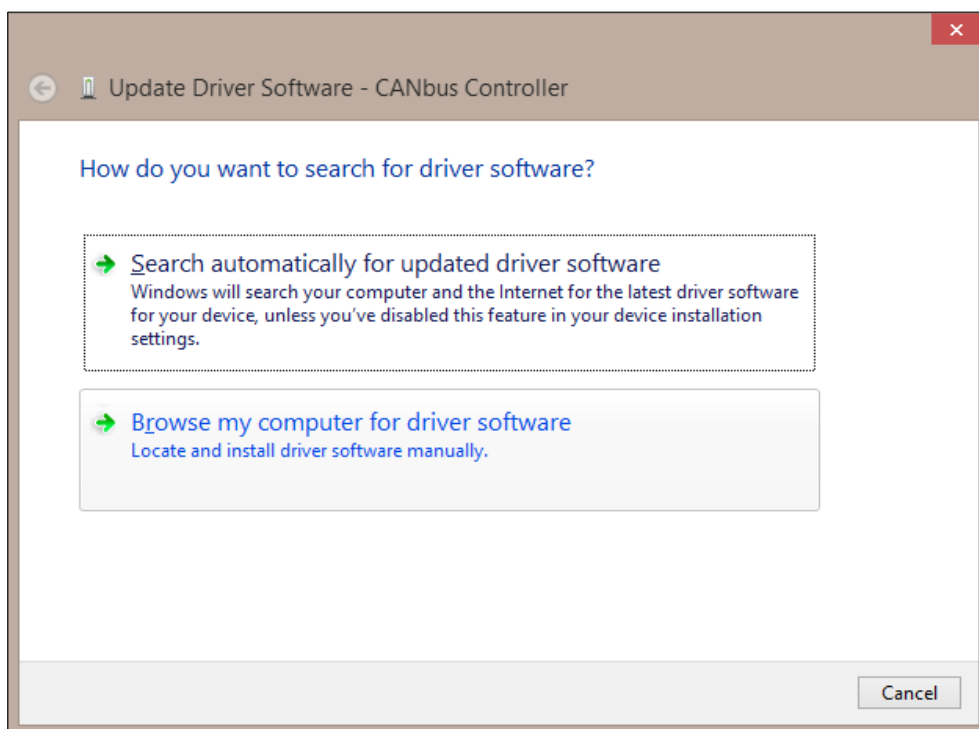


In the *Device Manager* windows there will be a device under *Other Devices* with a yellow exclamation point next to the icon to indicate that there is no device driver installed yet. The text next to the device will depend on the CAN module attached.

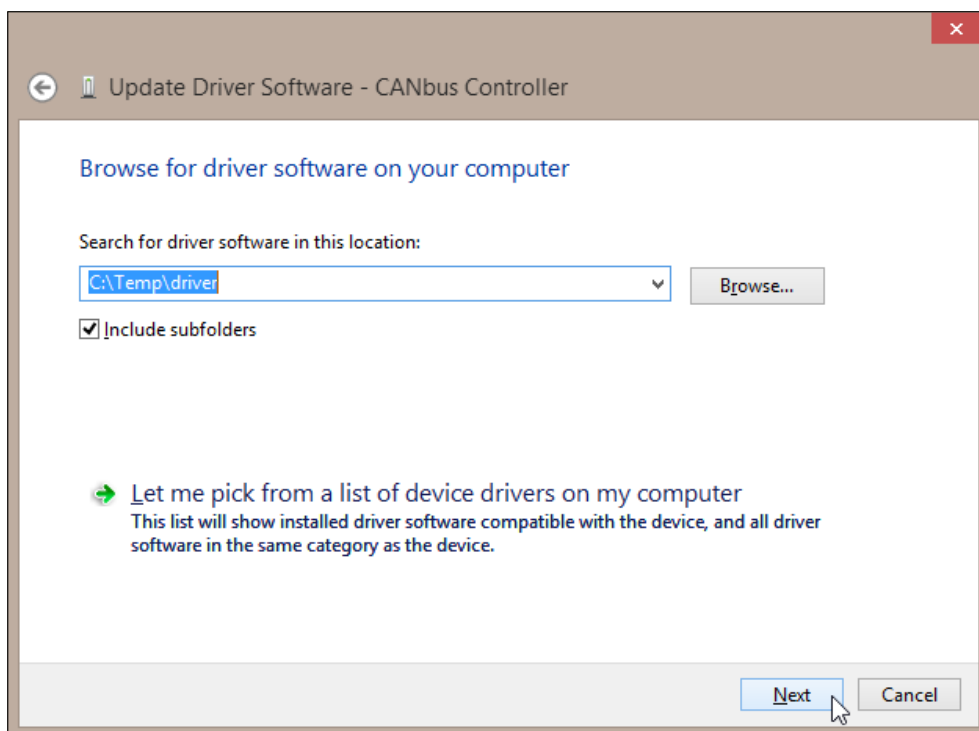
Right click on the device to bring up the context menu as shown above and select the menu item **Update Driver Software...** which opens the following dialogue box.



Right click on the device to bring up the context menu as shown above and select the menu item **Update Driver Software...** which opens the following dialogue box.

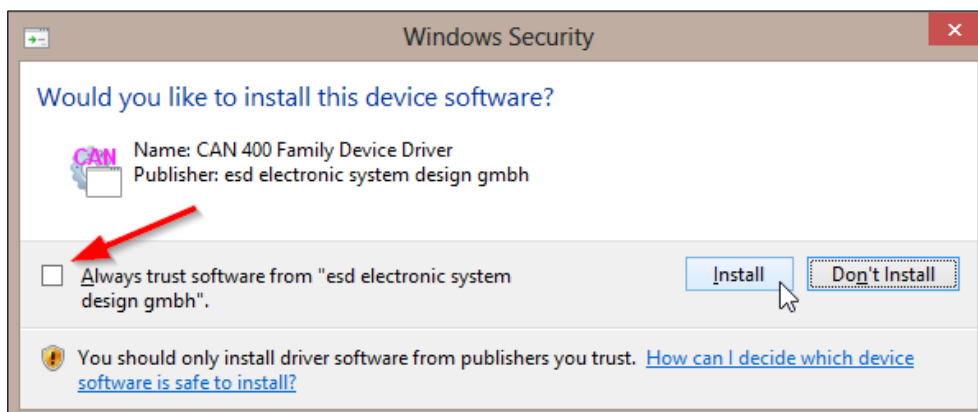


Select the second option to “Browse the computer for driver software” and the following dialogue box will appear.



Press the **Browse...** button to define the location of the driver files. This might either be the drive letter of your optical driver if you want to use the CD which accompanied the delivery of your CAN module or is the location on your hard disk where you have extracted a driver archive downloaded from the **esd** website (www.esd.eu). If the driver files are in a sub directory of the configured path do not forget to check the “Include subfolders” option in the dialogue before you press the **Next** button to start copying the files to your system which may take some time.

During driver installation you will see a security message similar to in the dialogue below.



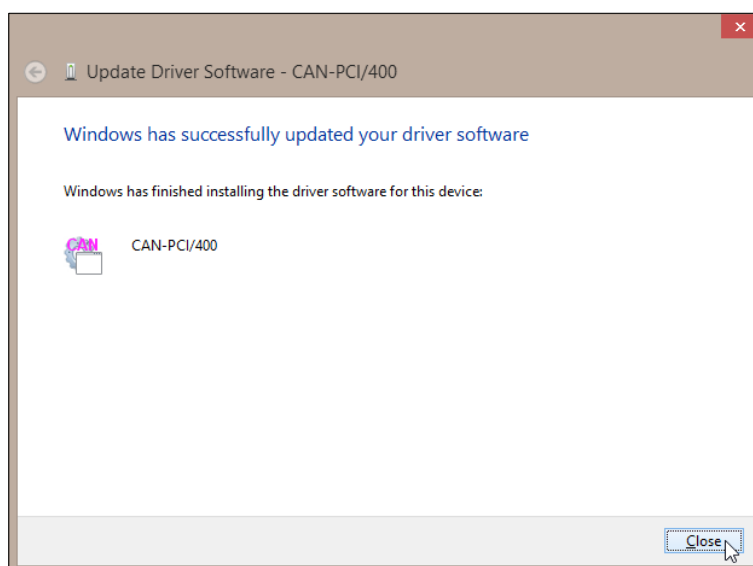
All CAN device drivers are digitally signed to give you as end user who is installing this software the possibility to verify that **esd** is really the publisher of this driver package and that the binaries are not tampered. Please refer to chapter 2.9 for more details about *Digital Signatures*.



Note:

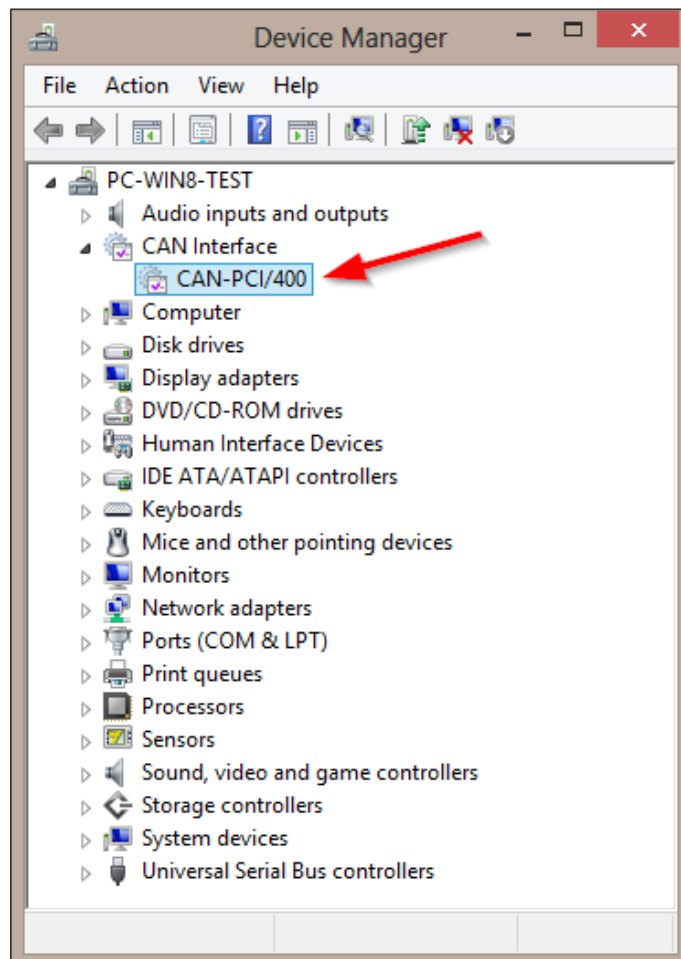
If you activate the check box "Always trust software from *esd electronic system design gmbh*" you will not have to confirm this dialogue in the future during the installation of another digitally signed driver for an **esd** device.

Press the **Install** button to continue.



When the installation is finished a completion dialogue as above is displayed and the driver is now started automatically with every Windows start-up. The displayed device name depends on the CAN module. Press the **Close** button to complete the installation.

If you return to the *Device Manager* window you will see that the CAN module is now listed below the new device class “CAN Interface”.



If you have installed several **esd** CAN modules attached to a local bus of your system you will find all of them here. By double clicking the device you will open the *Properties* dialogue where you can configure the device specific options described in chapter 2.3 via the *Settings* tab.



Note:

If you just want to run NTCAN based application on the system you are done.

If you intend to develop NTCAN based applications on this system you also must install the CAN SDK as described in chapter 2.8 which contains in addition to many tools the required header files, library files and/or wrapper for your development environment.

2.1.2 Software-First Driver Installation

The process for a software-first driver installation is similar on all versions of Windows since Vista and is covered in chapter 2.4.

2.1.3 Driver Lifecycle Management

The process of updating, rolling back or uninstalling a device driver package is very similar on all Windows versions and covered in chapter 2.6.

2.2 Windows 10 / 11 (Arm64)

This chapter covers the device driver installation for Windows 10 / 11 for Arm64.

**Note:**

The Windows on ARM64 device drivers require Windows 10 2004 (Build 19041) as the minimum supported Windows on ARM64 version.

Windows 11 on Arm supports emulation of both x86 and x64 apps. Performance is enhanced with the introduction of the new emulator *Prism* in Windows 11 24H2. Windows 10 on Arm also supports emulation, but only for x86 apps.

This transparent emulation makes it possible that you can run (existing) x86 or x64 NTCan based applications on Windows on Arm versions without any modifications.

For x86 apps, an WoW64 layer of Windows allows x86 code to run on the Arm64 version of Windows (like on the x64 versions of Windows) with filesystem and registry redirection.

For x64 apps, there's no WoW64 layer and no separate registry or folder of Windows system binaries. Instead, shared libraries are made available as Arm64X PE versions that can be loaded into both x64 and Arm64 processes from the same location without the need for filesystem redirection. This means that x64 applications can access the entire OS, both filesystem and registry, without the need for special code.

**Note:**

Please note the drivers delivered on CD are most likely outdated. We rather recommend checking the esd electronics gmbh website for newer driver releases and use these ones to circumvent any problems caused by known and fixed issues before they occur.

2.2.1 Installation

The support of esd CAN interfaces for Windows on ARM is installed with a setup application instead of using the Windows device manager. This setup application is based on the built-in Windows driver store and the mechanism of *Driver Staging* (see chapter 2.4.1).

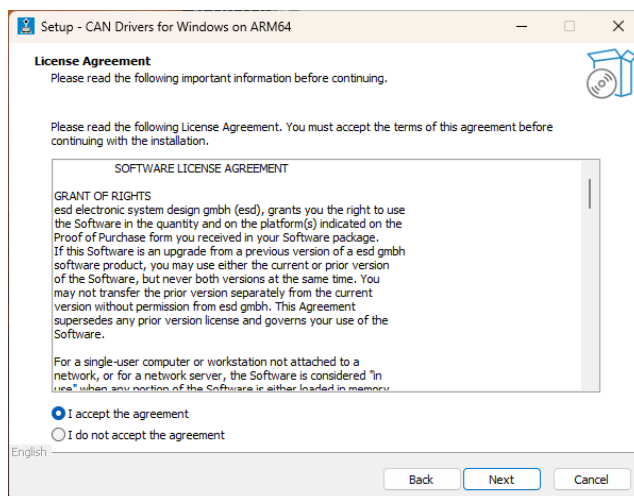
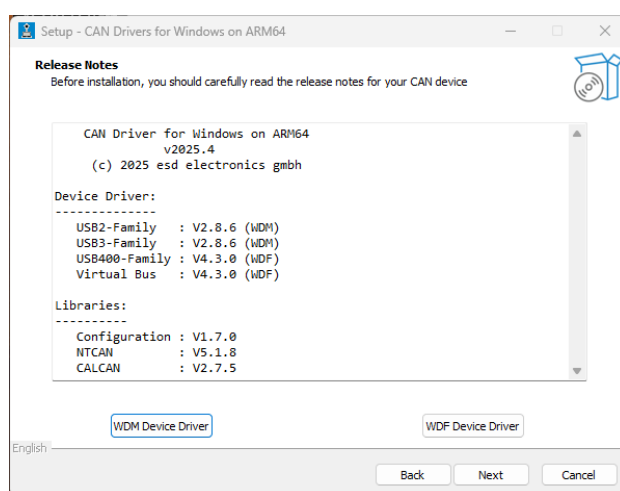
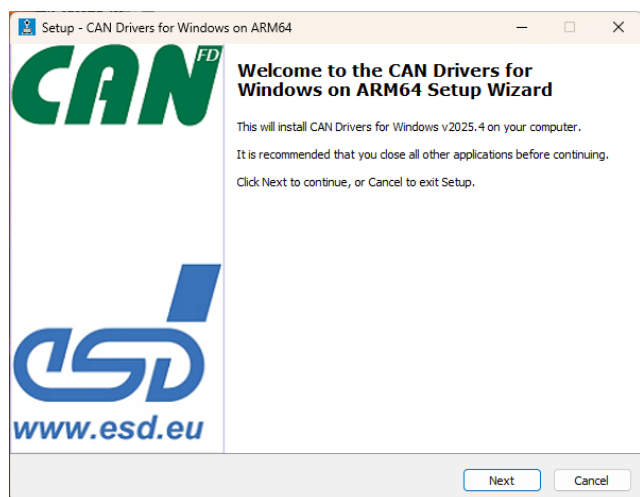
Note:



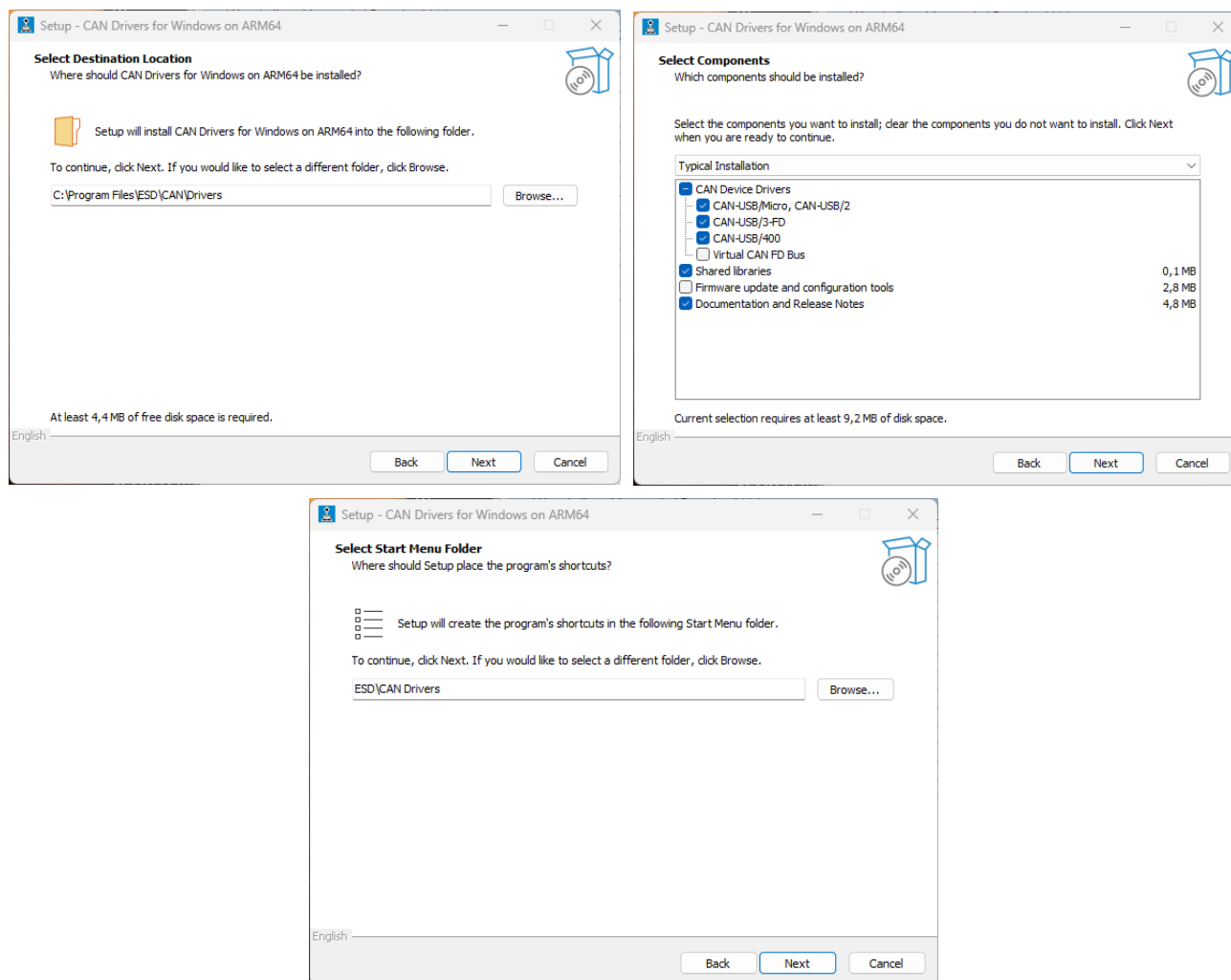
The setup application supports hardware first as well as software first installation which means a device driver gets installed and started for any supported CAN interface which is already attached to the target during installation as well as a driver gets automatically installed and started for any supported CAN interface which is attached at a later point in time.

After starting the installer, you may see a message from 'User Access Control' asking for permission to proceed (see chapter 2.9.3 ff. for details).

Follow the installation instructions shown in the pictures below, including acceptance of the license terms, to start installing device drivers, shared libraries, tools and documentation to the system.



You can select the destination folder, the components you want to install and the start menu entry that is created:



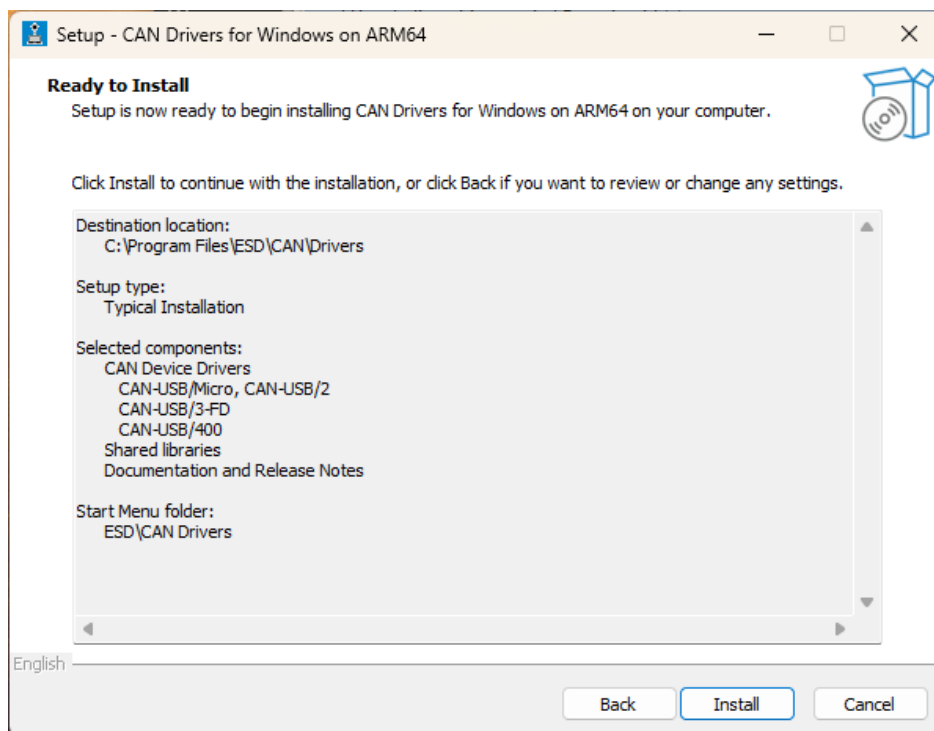
Note:



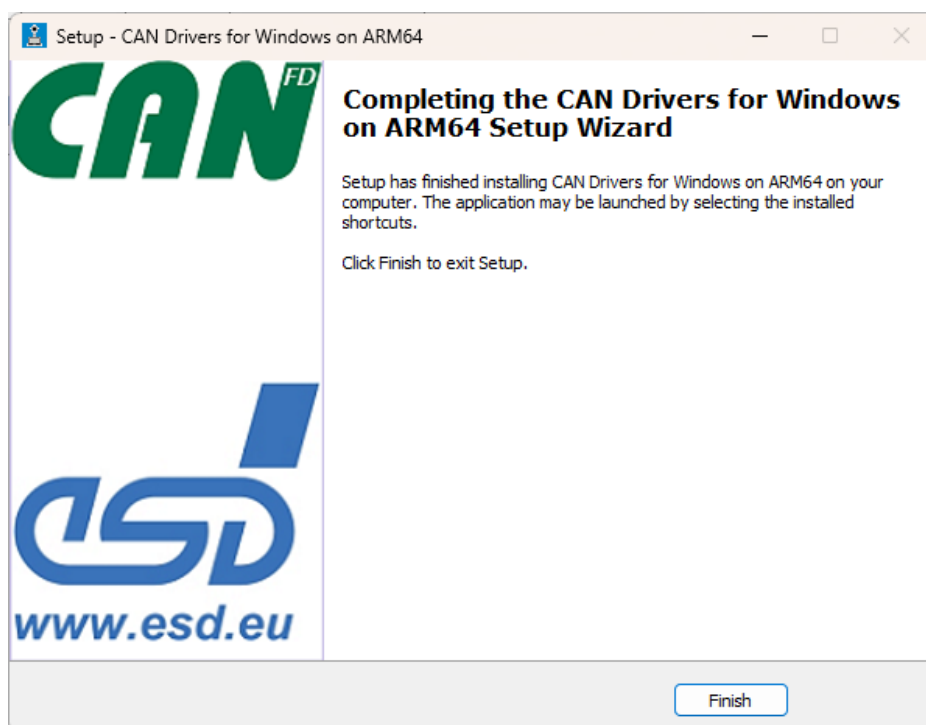
In the *Typical* (default) installation mode only the device drivers for physical esd CAN interfaces will be installed in the Windows *Driver Store*.

If you want to use the virtual CAN driver you must either select this explicitly or change to the *Full* installation mode.

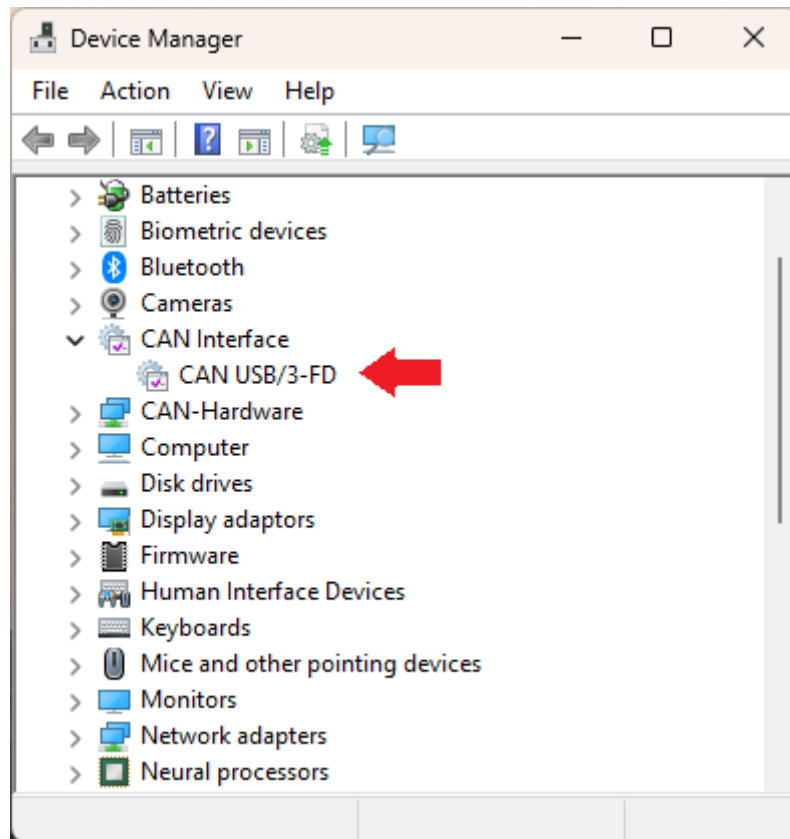
Before the installation process is started the setup application shows a summary of your choices. Press the **Install** button to proceed and wait for the end of the installation process.



After the installation, the completion dialogue below indicates the end of the installation. Press the **Finish** button to terminate the setup application.



If you open the *Device Manager* window, you will see the available CAN interfaces listed below the new device class *CAN Interface*.



By double clicking the device you will open the *Properties* dialogue to configure the device specific options described in chapter 2.3 via the *Settings* tab if necessary.

**Note:**

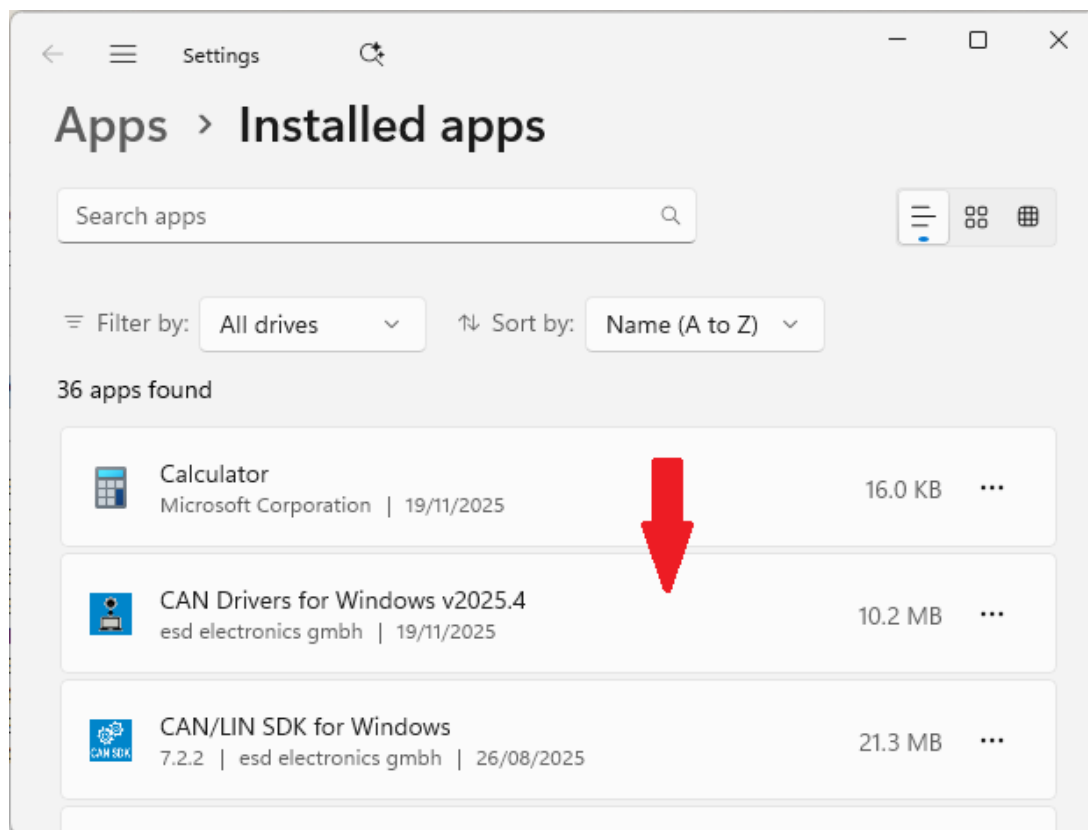
If you just want to run NTCAN based application on the system, you are done.

If you intend to develop NTCAN based applications on this system, you must also install the CAN SDK (see chapter 2.8) which contains in addition to many tools the required header files, library files and/or wrapper for your development environment.

If you select to install the (optional) *Tools* component firmware update tools with the latest firmwares for the different CAN interfaces are installed as well together with a start menu entry “CAN Firmware Update” which opens a console window with elevated rights to start the update tool for your CAN interface if necessary.

2.2.2 Uninstallation

You can uninstall the software via the Windows standard ways to uninstall installed applications.

**Note:**

The uninstall process will remove all shared libraries, tools and documentation from the system but will **NOT** remove the device driver files copied to the Windows *Driver Store*. Please refer to chapter 2.6 for instructions how to remove the device driver from the Windows *Driver Store*.

The process of updating, rolling back or uninstalling single device driver is very similar on all Windows versions and covered in chapter 2.6.

2.3 Configuration

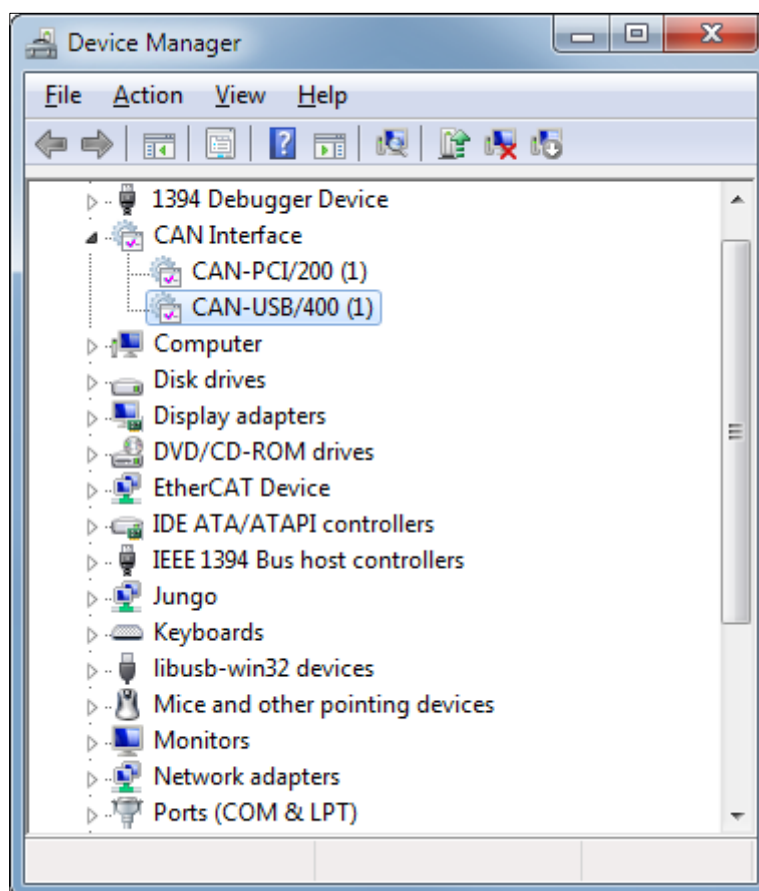
This chapter contains a description of configuration options of the device driver and Windows itself.

2.3.1 Device Driver

2.3.1.1 Standard Settings

This chapter covers the CAN device driver configuration for Windows.

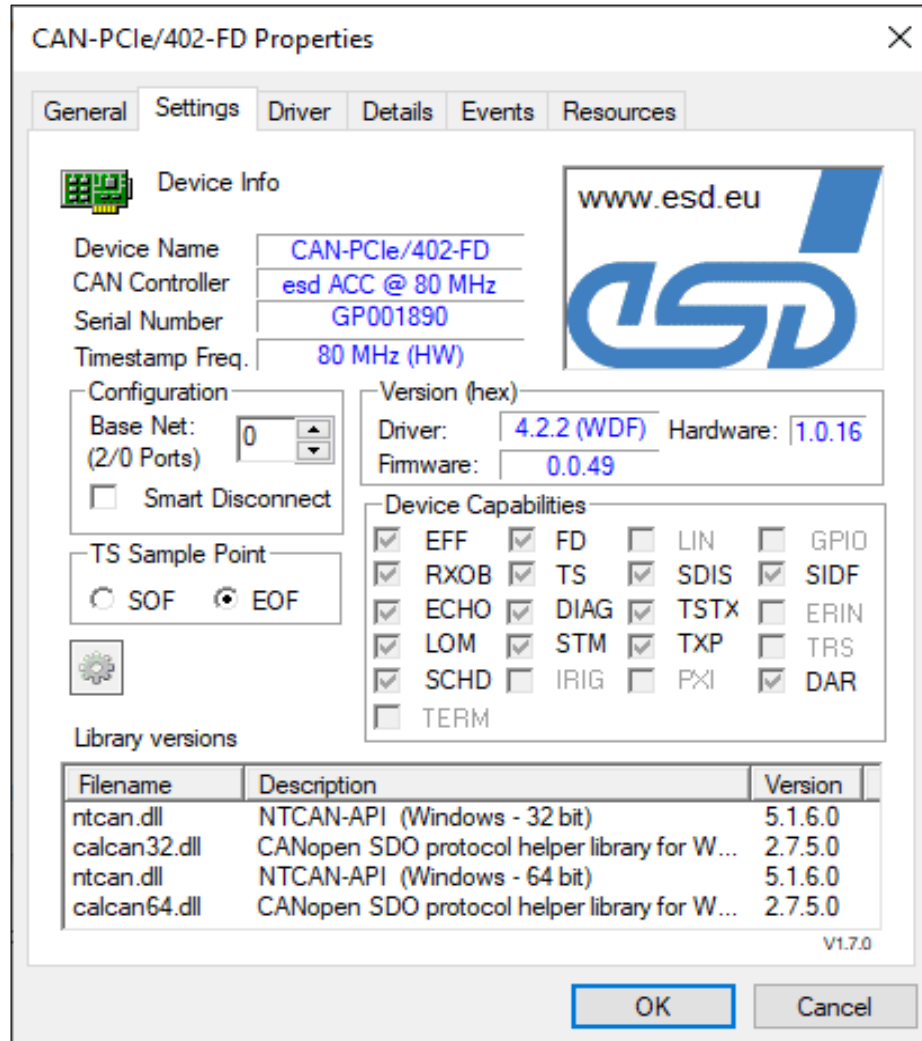
To configure several CAN driver related settings or to check the version of the installed components you must open the Windows Device Manager. All device drivers for **esd** CAN modules are installed as a **CAN Interface** class as shown below.



To change the device driver configuration parameter, double-click the device instance to open the *Properties* dialogue of the device and select the *Settings* tab.

The settings dialogue (Rev. 1.6.x or later) contains the following read only information:

- Device name with CAN controller type and CAN controller frequency.
- Device driver revision and type (WDM / WDF), the firmware revision(s) (if applicable) and the hardware revision.
- The serial number (if serial number access is supported by the hardware).
- The timestamp frequency (if timestamping is supported) and the information if this is a hardware (HW) or software (SW) timestamp.
- The CAN interface and driver related device capabilities.
- The version of the libraries which are installed together with the driver.



Nearly all elements in the configuration dialogues will present a context sensitive (English) help text if you move the mouse on them and wait for a while.

The text 'Device Info' at the top of the dialogue may be replaced by a problem notification with increased font size to ease troubleshooting.

If you click on the **esd** logo your default web browser is launched with the URL of the **esd** website where you can check if device driver updates for your CAN hardware are available for download.

The dialogue also allows to configure several driver parameters on a per device basis.

**Attention!**

Parameter changes are applied with the next start of the device driver and not immediately after the dialogue is closed.

Base Net

Via *Base Net* a logical net number is assigned to the CAN module which is used by NTCAN to distinguish between several physical CAN ports. The number of available physical ports is indicated here. If a CAN interface has more than one physical CAN port, the logical net number entered in *Base Net* is assigned to the first physical port and further ports will be assigned consecutive increasing net numbers. The default value for the first instance of a hardware is always 0.

**Attention!**

If there is more than one CAN module in the system, the user must make sure that the logical net numbers which are assigned to the physical ports do not overlap!

Smart Disconnect

The *Smart Disconnect Feature* to disable a port after the last handle is closed can be enabled or disabled, if supported by the CAN hardware. The default after installation is disabled.

Timestamp sample point

An ESDACC based CAN hardware (C400, C402 and U400 CAN device family) allows to configure if the timestamp of a CAN frame is captured at the Start of Frame (SOF) or at the End of Frame (EOF) which is the default after driver installation. For other CAN device families this configuration option is not available.

CPU Affinity

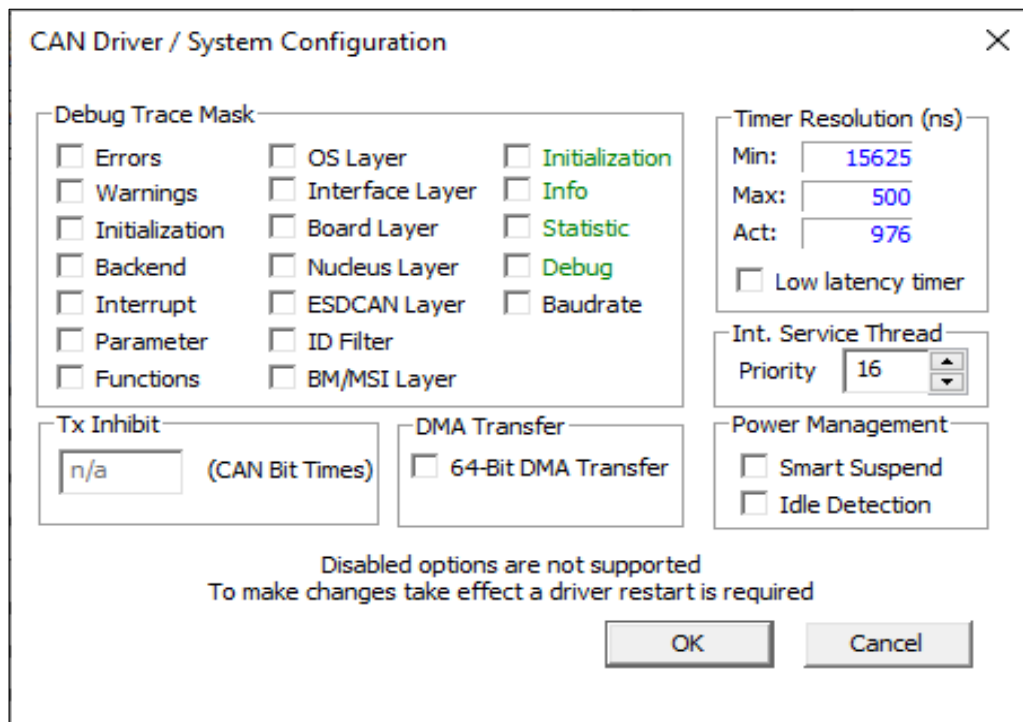
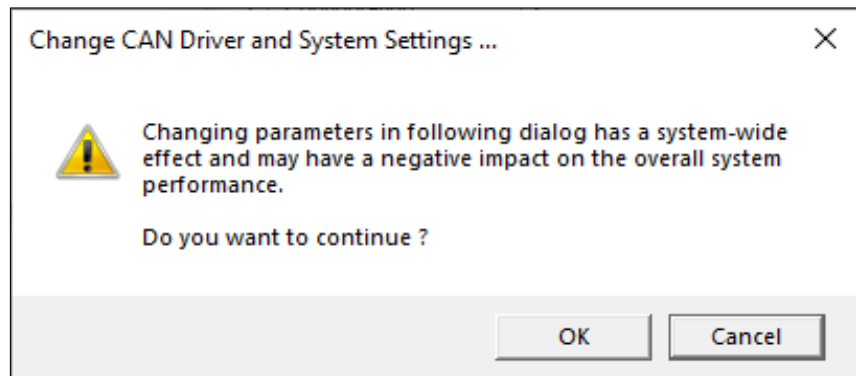
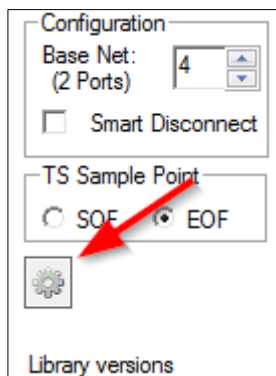
Earlier versions of this dialogue allowed to configure the *CPU Affinity* of the devices interrupt handler. This option has been removed (from the dialogue and the driver) as modifying the CPU affinity mask within the driver caused unwanted side effects if the interrupt was shared with another device. Please refer to chapter 2.3.2.2 for this topic.

Smart Suspend

Earlier versions of this dialogue allowed to configure the *Smart Suspend* option here which is now moved into the *Expert Settings* dialogue described in the next chapter.

2.3.1.2 Expert Settings

Via the cogwheel icon you can reach an expert settings dialogue after you have confirmed the warning that changing parameters in this dialogue may have (negative) system-wide effects.



Debug Trace Mask

For all device driver with a version greater than 3.10.x you can enable a trace mask which causes the driver to send trace messages via the Windows kernel debugger API. For the release build of the device driver only the options marked green have an effect. From Microsoft TechNet you can download the tool DebugView to capture these messages without setting up a kernel debugger.

Timer Resolution

In Windows the clock interrupt frequency can be changed within certain limits. The dialogue shows the actual active clock interrupt resolution in nanoseconds as well as the system specific minimum and maximum value. A higher value for the clock interrupt frequency decrease in some cases the I/O latency and improves the granularity of configured timeouts.

If you enable the **Low Latency Timer** option, the device driver will configure the interrupt resolution to 1 ms with the next driver start.

**Attention!**

Note that the result of changing the clock interrupt frequency is system-wide and can also have a severely negative effect on system performance. Also note that higher clock interrupt frequencies can shorten a system's battery life.

Service Thread Priority

All device driver V4.x.x and later process CAN messages on passive level instead on DPC level to reduce the overall system latency on multicore CPUs. The thread priority of the passive level worker thread which handles the CAN messages can be configured here.

Smart Suspend

If the device driver supports this feature a change into a lower power state (Sleep/Hibernate) is rejected by the device driver if there is an application with an open CAN handle.

**Note:**

A device driver is only able on Windows 2000/XP to prevent a state into a low power state completely. On Windows Vista and later a device driver cannot prevent such a change if it is explicitly requested by a user, but it can prevent changes if they result on configurations in the powerplan.

DMA Transfer

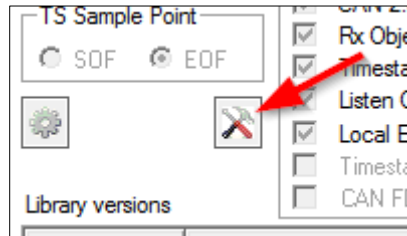
If the option '64-Bit DMA Transfer' is selectable and enabled, the device driver will indicate to Windows that the device supports single-packet DMA operations with 64-bit addressing. This allows Windows to assign DMA memory from the area above 4GB on 64-bit systems (but does not enforce this). If not selected the device driver will always request DMA memory from the area below 4GB.

**Note:**

The use of DMA memory in an area $\geq 4\text{GB}$ increases each PCIe packet by additional four bytes for the address, which may lead to a small drop in performance.

2.3.1.3 Device Specific Settings

If a device specific configuration tool is available for a CAN hardware (CAN-AIR/2, CBX-AIR/2, CBX-AIR/3) the common *Settings* dialogue will show a tools icon which opens this hardware specific configuration tool if clicked.

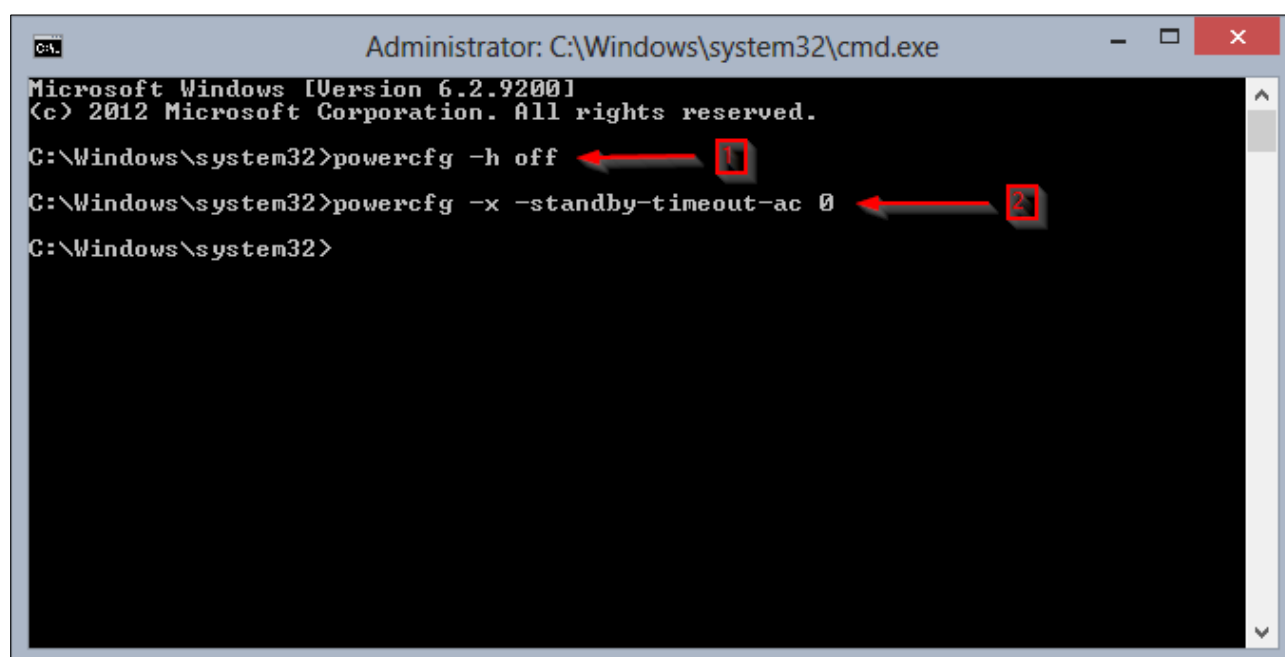


2.3.2 System

2.3.2.1 Power Management

The WDM device drivers (3.x.y) for the C400, C402 and C405 family (see 1.4) of CAN devices do not support a change to a low power state which includes the support of the *Hybrid Shutdown* feature introduced with Windows 8. With *Hybrid Shutdown* (which is enabled by default) the states of drivers and services are saved into the hiberfile on shutdown for a faster reboot.

To prevent problems using hardware from this driver families on Windows you must make sure that the change to a low power state is disabled on your system. This can be achieved via various power management related dialogues in Windows, but the easiest method is to use the *powercfg* command line utility which was introduced with Windows XP SP2. Powercfg must be run from an elevated command prompt and requires administrator rights.



Use the command `powercfg -h off` (1) to disable hibernation and implicitly the *Hybrid Shutdown* feature of Windows 8. If you want to restore hibernation you have to use 'on' instead of 'off'.

Use the command `powercfg -x -standby-timeout-ac 0` (2) to disable Windows standby mode. If you want to restore the standby behaviour, use a positive value in minutes instead of 0 as parameter. On Windows XP you must write `-change` instead of the abbreviation `-x`.

2.3.2.2 Interrupt Affinity

The interrupt affinity is the set of processors/cores that should service an interrupt in a multi-processor/multi-core architecture. After the device driver installation each PCI / PCIe / CPCI / PMC based CAN device has the default interrupt affinity policy that any processor/core can handle its interrupt which usually need not to be changed.

In cases you want to assign the affinity to dedicated (set) of processors or cores you should use the *intfiltr.exe* tool which is part of the *Windows 2003 Resource Kit* for Windows 2000/XP or the *Microsoft Interrupt-Affinity Policy Tool* for Windows Vista and later.

2.4 Device Driver Preinstallation

This chapter describes the process of (pre-)installing a device driver before the hardware is present (*software-first* installation) for Windows Vista and later so the device driver for **esd** CAN hardware is installed as soon as the PnP manager detects a (new) instance of the hardware without further user interaction like devices which are supported by Windows in-box drivers.

2.4.1 Driver Staging

Starting with Windows Vista, Microsoft introduced a repository for device drivers which is called the *Driver Store* and split up the device installation process into two steps:

- **Driver Staging:** The process of adding driver packages to the *Driver Store*.
- **Driver Installation:** The process of installing drivers from the *Driver Store*.

During a hardware-first installation the PnP Manager performs the step of driver staging implicitly so a device driver is always installed from the driver store, but the user experience remains similar to the Windows versions before Vista. You can find a detailed description of the steps during driver installation in this [Microsoft TechNet article](#).

The central repository which even keeps several versions of a device driver offers some advantages compared to earlier versions of Windows:

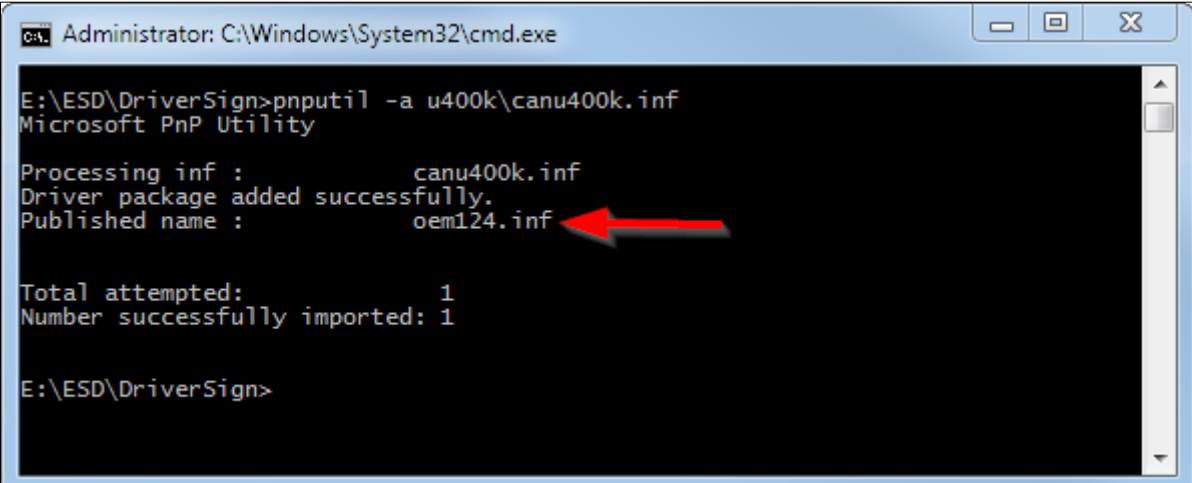
- Driver Repair or Re-installation no longer requires the source media.
- Managing a driver rollback (see chapter 2.6.2) is easily possible.
- The driver store can be preloaded with OEM drivers, and the device hardware does not need to be present during this staging process.

The option to stage a device driver is the precondition for a software-first installation process. The in-box (console) staging tool for Windows is [PnPUTil](#) which usage requires administrative privileges on the system. During driver staging, the driver files are verified and copied into the driver store. They are not installed on the system until the device hardware is detected by the PnP manager.

The staging process is triggered with the command

```
pnputil -a <Path\drvname.inf>
```

as shown in the picture below for a CAN-USB/400 device driver. The directory with the INF file must contain all driver files in the hierarchy of the distribution media.

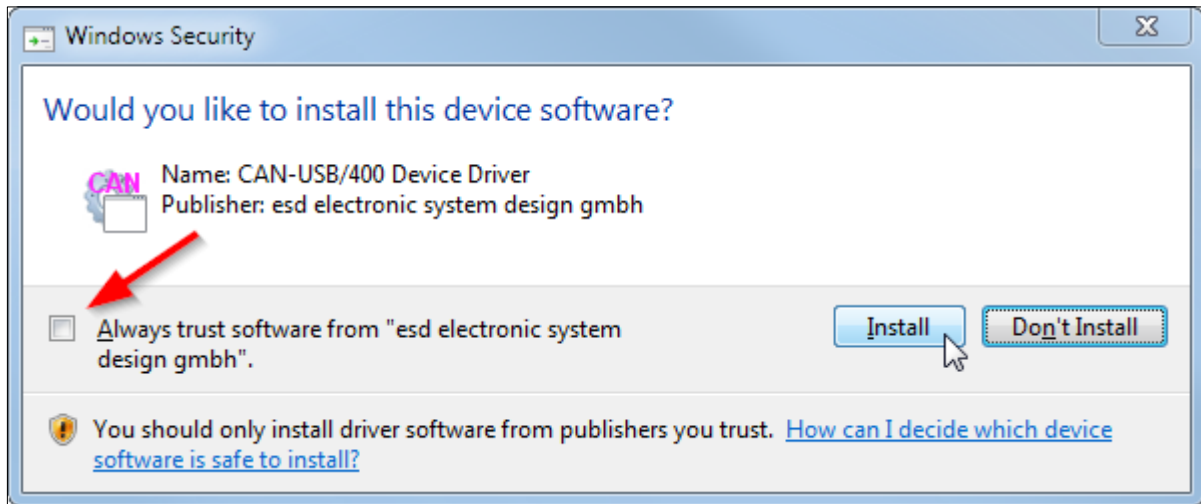


```
Administrator: C:\Windows\System32\cmd.exe
E:\ESD\DriverSign>pnputil -a u400k\canu400k.inf
Microsoft PnP Utility

Processing inf :          canu400k.inf
Driver package added successfully.
Published name :          oem124.inf
Total attempted:          1
Number successfully imported: 1

E:\ESD\DriverSign>
```

During the staging process you must complete the *Windows Security* dialogue with the **Install** button in the same way you have to do it for hardware-first installation process. Optionally you may choose to accept **esd** as trustworthy software publisher so this dialogue will not appear in further device driver staging or installation tasks (refer to chapter 2.9.2 for more information about digital signatures).



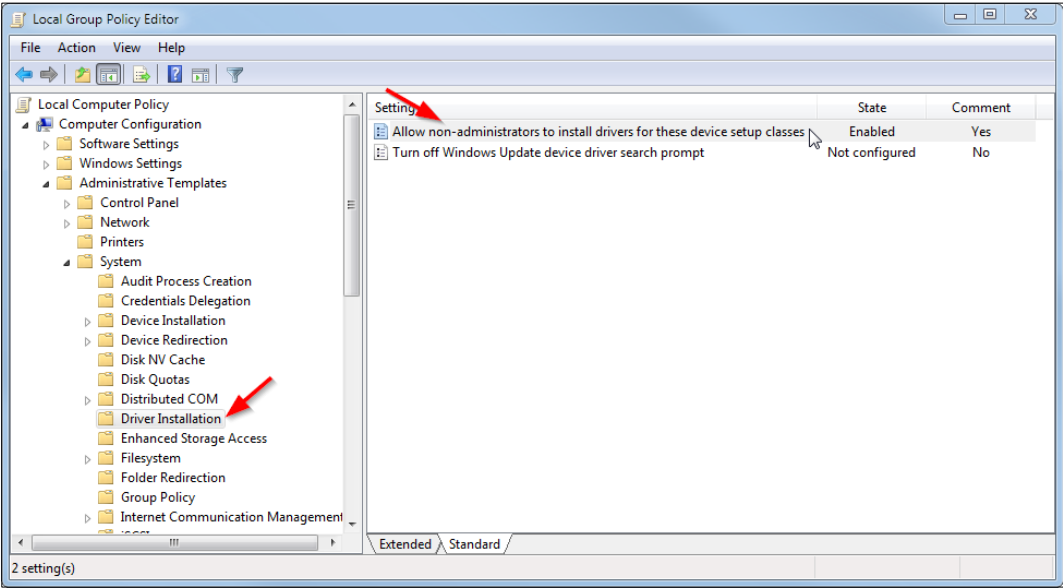
Successfully staged drivers can also be removed from the driver store. For this purpose, you have to note the *Published Name* which is indicated during the staging process in the console window. This name consists of the common prefix '**oem**' followed by a digital number and the suffix '**.inf**'. In the example above the name is **oem124.inf** and you can remove this staged driver with the command:

```
pnputil -d oem124.inf
```

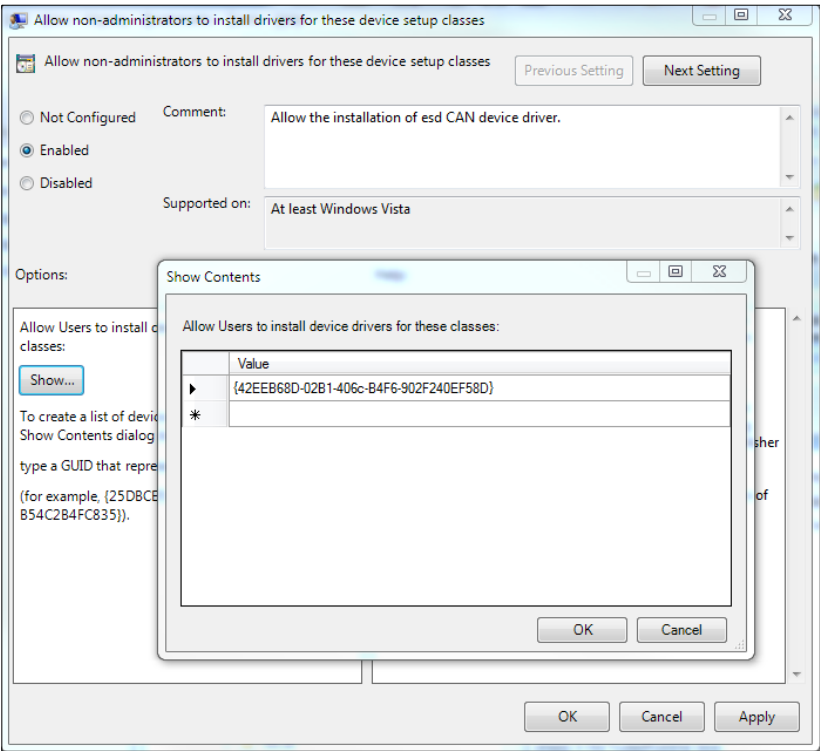
2.4.2 Driver Installation for Non-Administrators

Without further administrative action the installation of a device driver, which has been staged as described in the previous chapter, is only possible for users which belong to the local Administrators group.

This [Microsoft TechNet article](#) describes the required steps to configure a policy with the *Local Group Policy Editor* to allow users which do not belong to the local Administrators group to install previously staged device drivers. If you follow the steps of this article you have to double click the “**Allow non-administrators to install drivers for these device setup class**” rule as shown in the following picture:



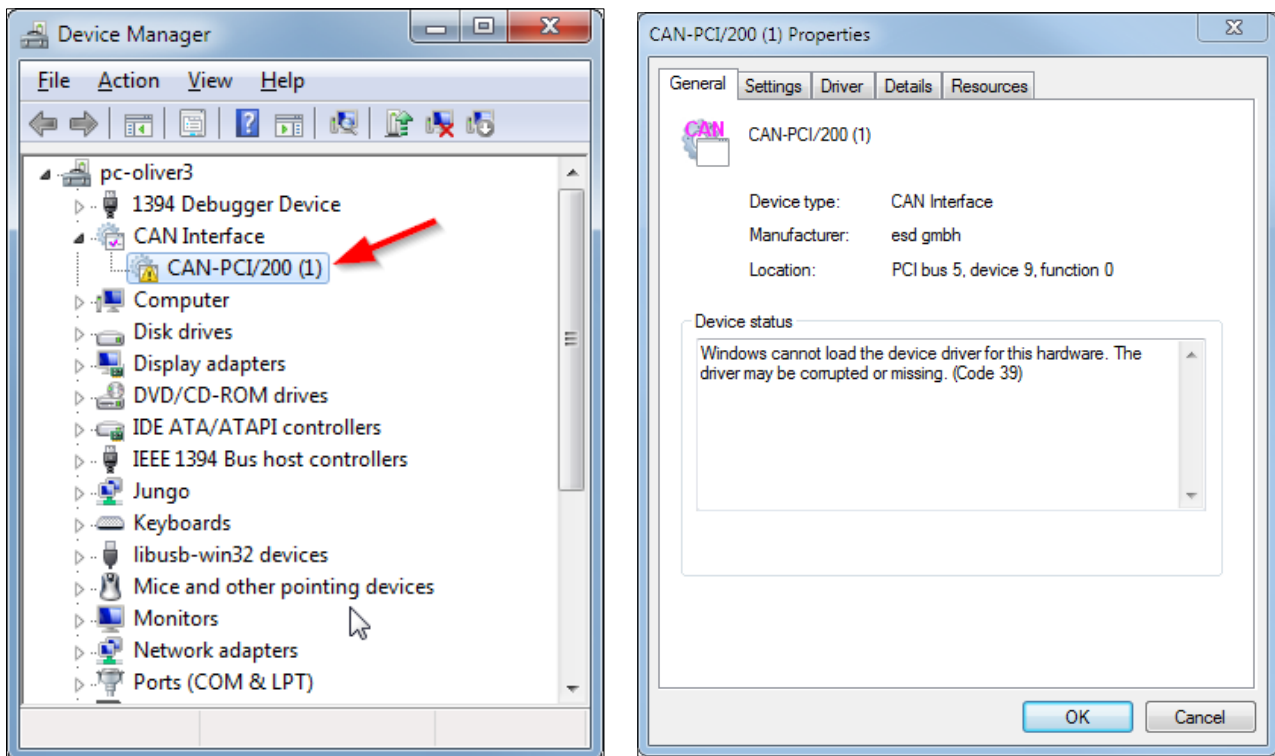
The device setup class for **esd** CAN devices is **{42EEB68D-02B1-406c-B4F6-902F240EF58D}** which has to be added in the dialogue which is opened if you click the “**Show**” button.



2.5 Troubleshooting Driver Installation

If the installation of a device driver fails, the Windows *Device Manager* is the central starting point for troubleshooting the problem. One of the fastest ways to open the *Device Manager* (which works on any supported Windows version) is pressing the key combination **WinKey + Pause/Break**. This opens the system settings dialogue where you can open the *Device Manager* with a further mouse click.

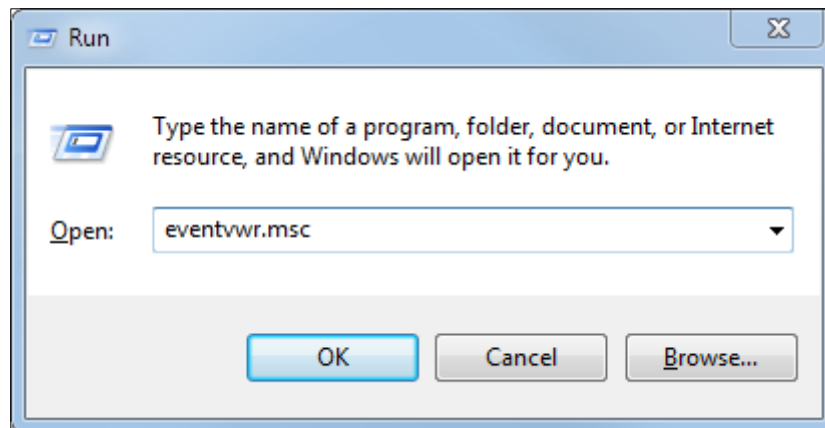
In case of a problem the CAN hardware is marked with a yellow exclamation point and you can double-click on the device to view more information about the problem.



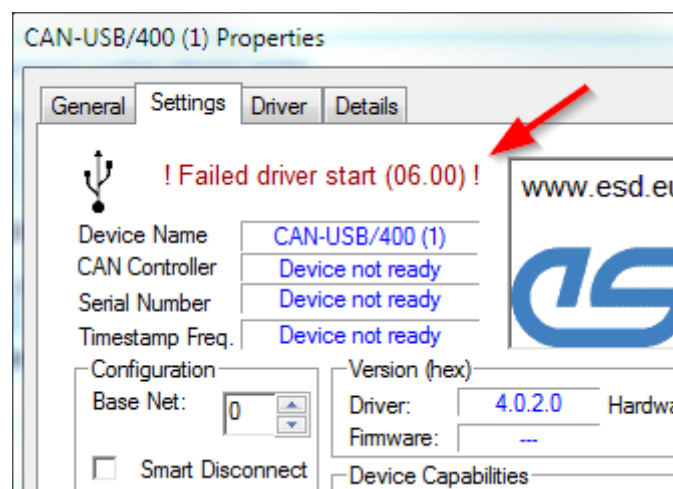
Follow this [link](#) for a list of possible error codes and reasons. The next two chapters deal with the two most common error situations:

2.5.1 Error Code 31

This is an indication that Windows starts the device driver, but something went wrong during the startup process. In this case in many error situations the CAN device driver will store additional information in the Windows *System Event Log*. You can open the Windows Event Viewer by pressing the key combination **WinKey + R** and starting *eventvwr.msc*.



WDM based device driver from the driver package 2.6.8 and later as well as all WDF based device driver can indicate the error reason in the upper left corner of the device's *Settings* tab.



The error reason is indicated as a CAN device driver specific major and minor error code separated by a colon. If you double click on this number the Windows *Event Viewer* will be opened with a filter configured for this device driver.

The following table contains a list of common major error codes which are reported during device initialization.

Error Code	Description
1	Failed to map physical address.
2	Failed to attach the interrupt handler.
3	Failed to verify the CAN hardware.
6	Failed to create the logical base net as it is already in use by another CAN device.
7	Out of resources.
8	No interrupt assigned to the device.
11	Firmware and device driver are incompatible. Update firmware or driver.
12	Hardware and device driver are incompatible. Update device driver.
13	Unrecoverable PCI bridge bug detected.
14	Failed to write into the PCI configuration space.
16	This is an engineering release of a debug driver without CAN I/O functionality.
19	Internal error during initialization.
20	Failed to attach to a lower-level device driver.
21	Failed to register CAN interface class.
22	Failed to configure an USB device.
25	Bootloader update required (CAN-PCI/405).
28	Error returning from standby or hibernate.
29	Failed to create a logical net as it is already in use by another CAN device.

Table 11: Windows Device Driver Installation Error Codes

2.5.2 Error Code 39

This is an indication that the device driver is missing, corrupted or especially on 64-bit system the validation of the digital signature has failed (see chapter 2.9.2).

If re-installing the device driver does not solve the problem you can check a plaint text log file where Windows Vista and later versions of Windows store information about device driver installation process especially if a signing problem exists. You will find this log file in

%SystemRoot%\inf\setupapi.dev.log

2.5.3 Error Code 52

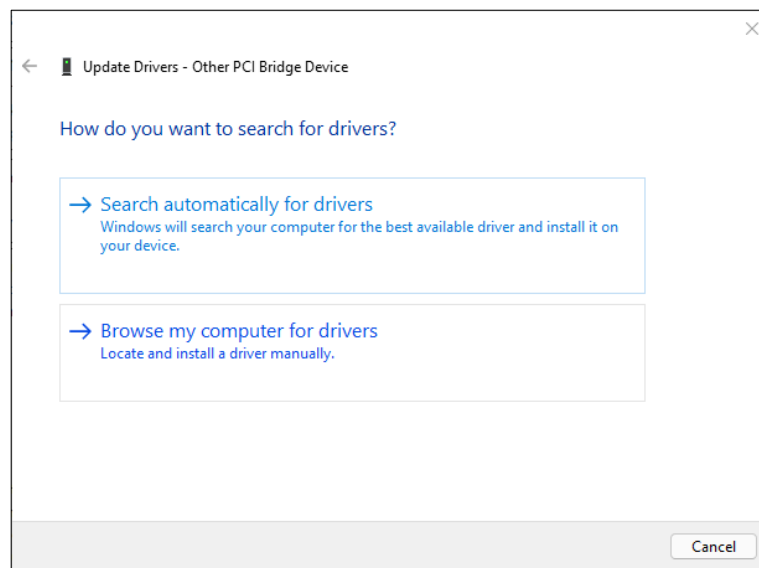
This is an indication that the device driver may be unsigned or corrupted. (see chapter 2.9.2).

2.5.4 Best Driver already Installed

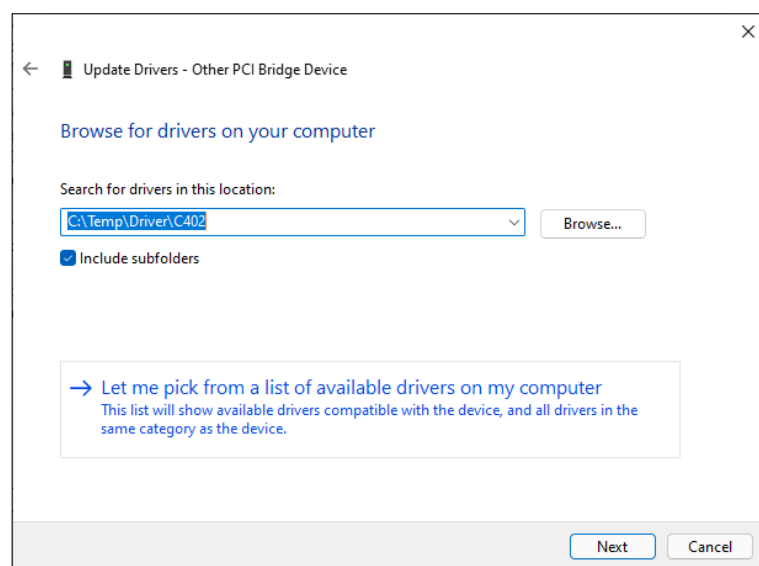
Windows implements an internal mechanism of driver ranking. If there is already a driver installed which is ranked higher than the one you want to install Windows will cancel the installation process with the message that the “Best Driver is already Installed”. If the driver has to be installed anyway, there are two possibilities to overcome this situation:

1. Uninstall existing drivers until Windows reports that there is no driver available for this device (see chapter 2.6.3).
2. Enforce the installation of the driver installation as follows:

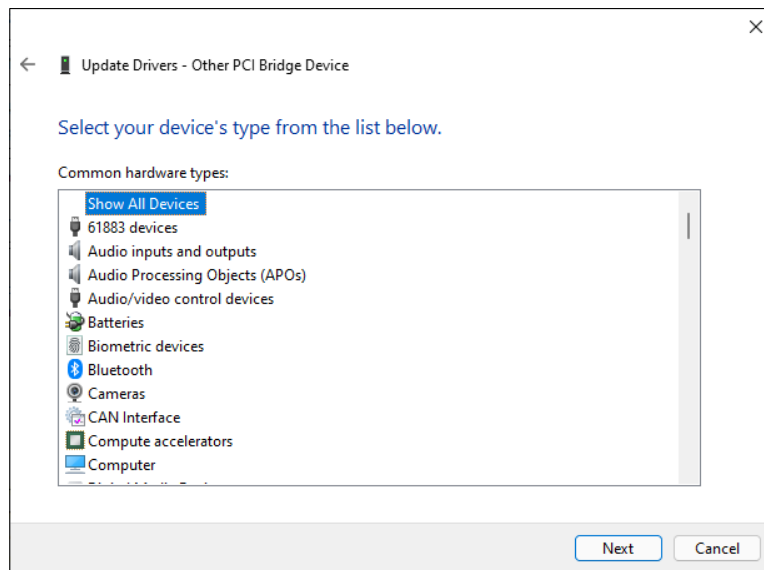
Open the Device Manager, select “Update Driver” in the context menu of the device (see chapter 2.6.1) and choose “Browse my computer for drivers” in the dialogue which is opened by Windows.



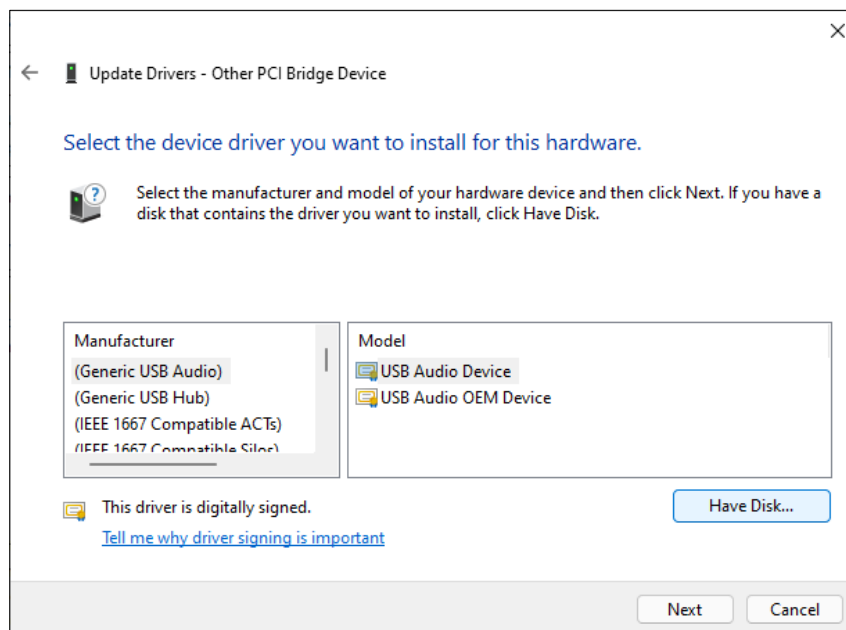
Choose “Let me pick...” in the following dialogue.



Select “All available devices” in the following dialogue:



Choose “Have Disk...”, change to the location of your driver installation package and start installing the driver.



2.6 Device Driver Lifecycle Management

This chapter describes the process of updating a device driver to a newer version, rolling back a device driver to a previous version or uninstall a device driver. This process is similar for all Windows versions.

**Attention!**

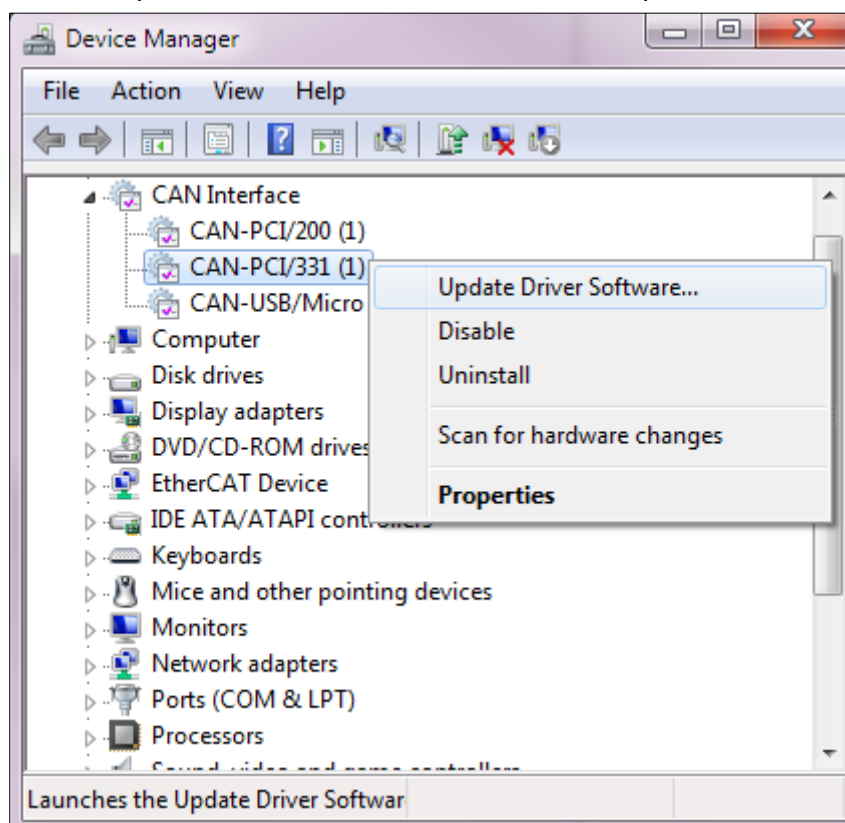
Close all applications which are using the NTCAN-API before you start updating or rolling back a device driver.

**Note:**

As most device driver packages are family device driver which support more than one CAN device type (see chapter 1.4) all devices covered by this driver are affected by any change.

2.6.1 Driver Update

To update the device driver to a newer version, open the *Device Manager* and select in the context menu of the CAN device *Update Driver Software* as shown in the picture below for Windows 7.



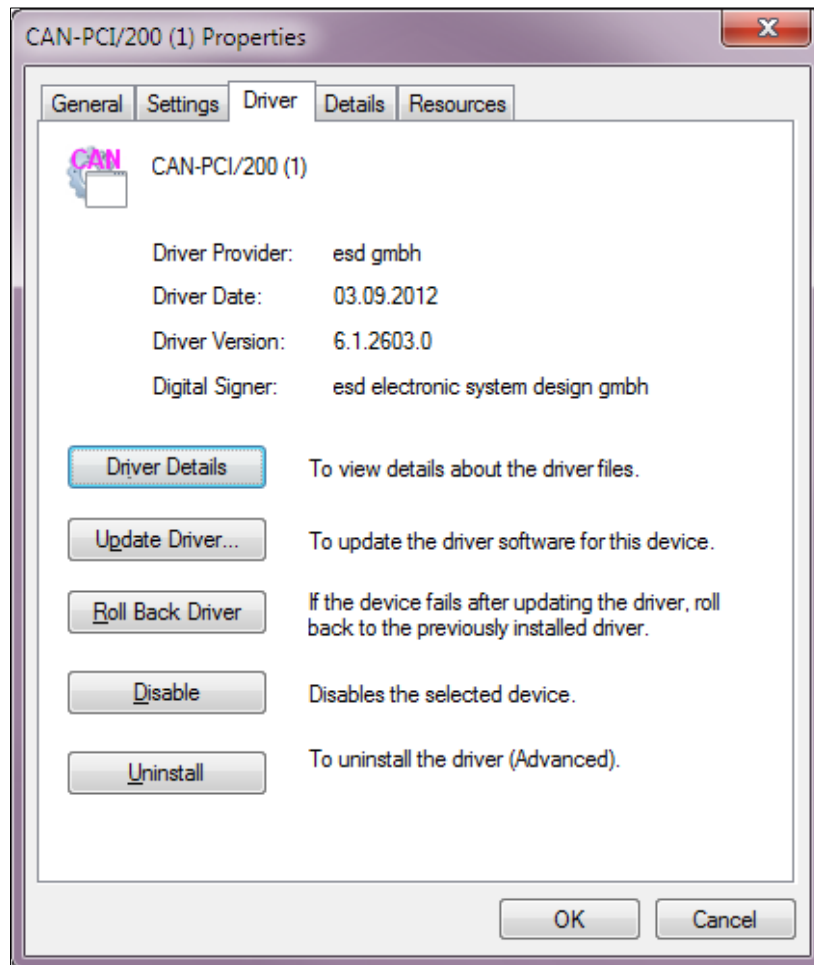
Follow the steps of the wizard which will guide you through the device driver update process which is like the initial installation of a device driver.

**Note:**

Depending on the files which are updated a restart of the system might be required. The configuration of the device driver is not affected by the driver update. If an update introduces new configuration parameter, they will have the default value.

2.6.2 Driver Rollback

If you have updated the initially installed driver to a newer one, starting with Windows XP you can roll back your driver to the previous version. Double click the CAN device in the *Device Manager* and select in the *Properties* dialogue the *Driver* tab as shown in the picture below for Windows 7.



To roll back to a previous version of the driver press the **Roll Back Driver** button and confirm the following security dialogue.

**Note:**

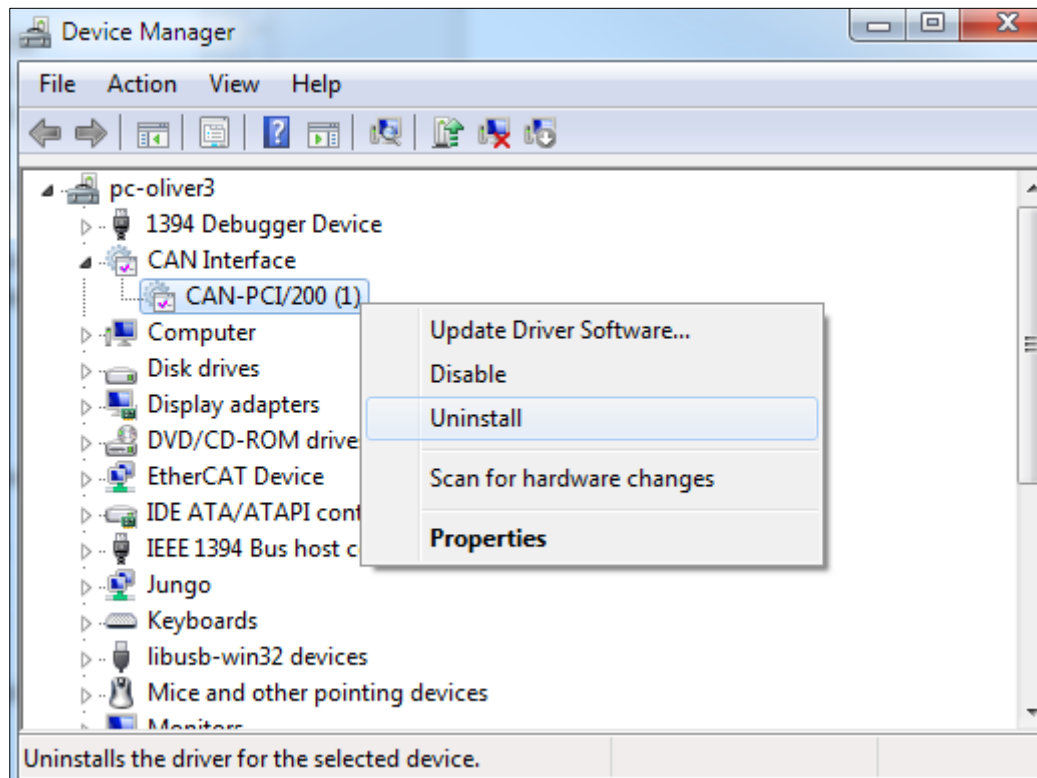
If the installed device driver is the only one in the Windows driver store for this hardware the **Roll Back Driver** button is not enabled.

**Attention!**

A newer version of a device driver often adds functionality and fixes problems that were discovered in earlier versions. Rolling back a driver can cause the loss of that new functionality and can reintroduce the problems that were addressed with the newer version. Furthermore, the DLLs are shared between device driver packages for different **esd** CAN device families so that a roll back of one driver might also affect files of another driver.

2.6.3 Driver Uninstall

Starting with Windows Vista you can completely remove an installed driver package from the driver store if you start a device **Uninstall** operation in the device manager



and check the **Delete the driver software for this device** in the confirmation dialogue box.



If an older version of the device driver is available in the driver store Windows will choose this one the next time the device is enumerated so basically this driver uninstall operation is a driver rollback as described in the previous chapter without and implicit re-enumeration.

2.7 EtherCAN Driver

In comparison to CAN interfaces connected to a local PC bus (PCI, USB, ...) supported with a Windows kernel mode device driver the EtherCAN, EtherCAN/2 and EtherCAN/3-FD interfaces are supported with a user mode device driver which integrates this remote CAN hardware into the NTCAN architecture in the same way as a local interface. This user mode device driver supports the EtherCAN/3-FD and EtherCAN/2 as well as the legacy EtherCAN hardware.

The required user mode driver on Windows is part of the digitally signed (see chapter 2.9.3) *EtherCAN SDK* available for Windows 7 (32-/64-bit) and later.

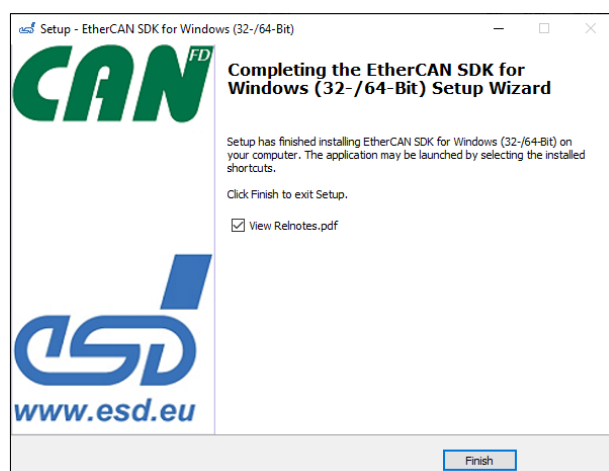
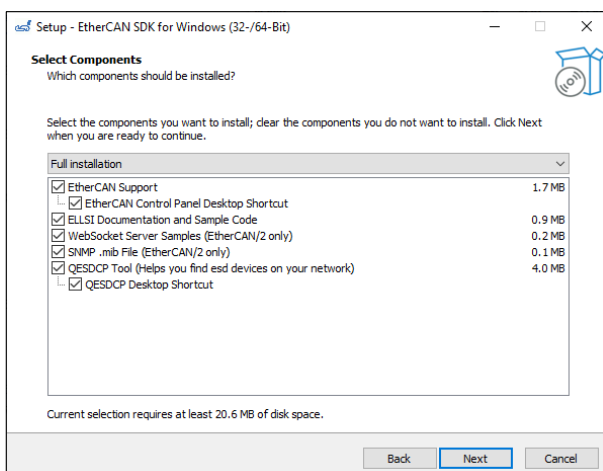
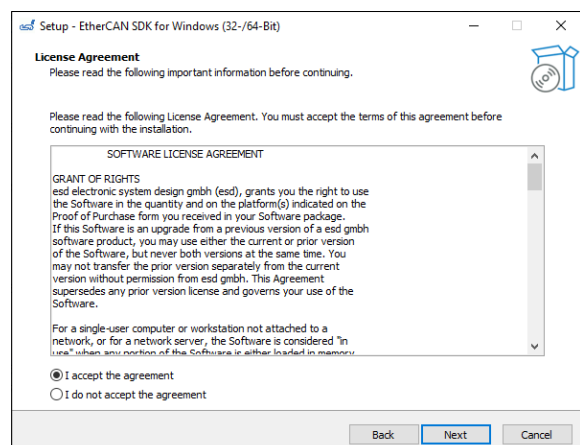
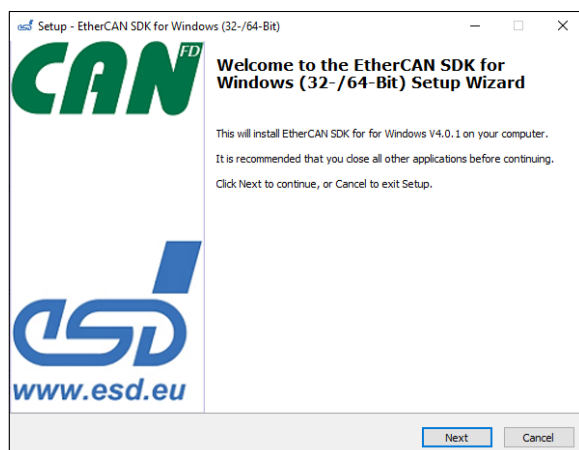


Attention!

A user which wants to install/uninstall the software must be member of the Administrators group.

2.7.1 Installation

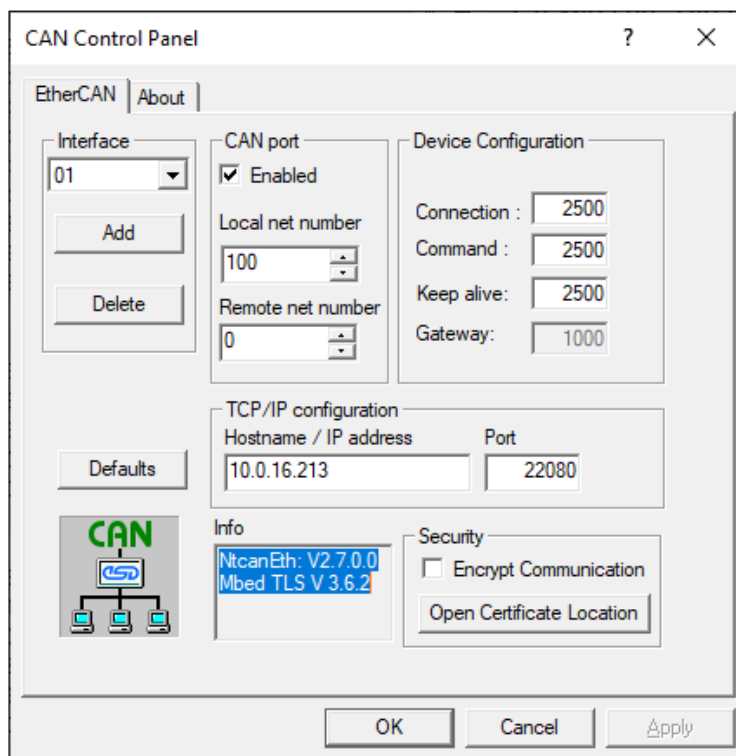
Start the installer application received on CD/DVD or downloaded from the **esd** website and follow the dialogue-based setup process shown in the picture below.



2.7.2 Configuration

For the configuration of the EtherCAN/2 or EtherCAN/3-FD please refer to the EtherCAN/2 hardware manual /3/ or EtherCAN/3-FD hardware manual /5/.

The local driver is configured with the help of the *CAN Control* application that can be started by clicking on “**esd/EtherCAN/CAN Control Panel**” in the Windows Start Menu which opens a dialogue like the picture below:



The following driver configuration options are available:

Interface

In the drop-down box you can choose the EtherCAN/2 interface instance. With the Add button you can create additional instances. With the Delete button you can remove the currently selected instance. The default after installation are 5 EtherCAN/2 instances which can not be deleted.

CAN port

You must assign a CAN network number between 0 and 255 and must enable it before this net number is available in the NTCAN environment to be used by your application.

As the EtherCAN/3-FD supports two physical CAN ports you must also define the remote net number (0 or 1). This is ignored by the EtherCAN/2.



Attention!

Please make sure that the assigned logical net number is not already in use by another EtherCAN or another CAN interface attached to a local bus.

<i>TCP/IP Configuration</i>	Configure the IP address or hostname (if registered in the DNS server) of the EtherCAN/2 interface. A change of the default port 22080 is not supported at the moment. The IP address must be identical with the one that is assigned to the EtherCAN/2 or EtherCAN/3-FD (please refer to /3/ or /5/ for details).
<i>Timeouts</i>	Currently three separate timeouts can be configured. The connection timeout defines the time the EtherCAN/2 driver waits for a response during the initial connection before the client software returns with a timeout. The command timeout defines the time after which a request to the EtherCAN/2 interface must be replied before the client software returns with a timeout. The keep alive timeout defines the time after which a keep alive request must be replied by the EtherCAN/2 interface before the host driver tries to reset and re-establish the connection.
<i>Security</i>	The communication between the host and EtherCAN/3-FD can optionally be TLS encrypted which has to be enabled explicitly and for which you must create and roll-out appropriate certificates (see /5/ for more details).
<i>Defaults</i>	The Default button restores all driver defaults for the timeout parameters.

**Note:**

If you just want to run NTCAN based application on the system, you are done.

If you intend to develop NTCAN based applications on this system, you must also install the CAN SDK as described in chapter 2.8 which contains the required header files, library files and/or wrapper for your development environment in addition to many tools which support test and debugging.

2.7.3 Uninstall

To uninstall the *EtherCAN/2 driver* from your computer you must open the **Add/Remove Programs** dialogue and uninstall the *EtherCAN SDK*.

The process of uninstallation is described in more detail for the *CAN SDK* in chapter 2.8.3.

2.8 CAN SDK

After driver installation you can proceed with the installation of the **esd CAN SDK** which supports software development on 32- and 64-bit Windows versions. It contains the necessary files, documentation, and tools to develop, debug and test NTCAN-API based applications.

Please refer to the release notes which gets installed with the CAN SDK for a complete list of supported programming languages and development environments.

**Note:**

If you have installed an older version (before V 2.x) of the CAN SDK uninstall this version before you install the new version.

If you have already installed a CAN SDK revision 2.x or newer you can overwrite the older installation or the uninstall process of a previous version is triggered implicitly if necessary.

The CAN SDK is deployed either on the CAN driver CD you receive with your CAN hardware or can be downloaded as an archive from the **esd** website. To install the **CAN SDK**, start **CAN_SDK.exe** located in the directory **CAN_SDK** of the CAN driver CD or start the installer after unpacking the downloaded archive. The installer is digitally signed so you can verify its integrity (see chapter 2.9.4) before you start installation.

**Attention!**

A user which wants to install/uninstall the software must be member of the Administrators group.

Follow the steps of the setup wizard to complete the installation.

2.8.1 Setup Command Line Parameter

The following table contains the most important command line parameters which are supported by the setup application to automate the installation process:

Parameter	Description
/HELP, /?	Shows a summary of all available parameters.
/SP	Disables the This will install... Do you wish to continue? prompt at the beginning of Setup.
/LOG	Causes Setup to create a log file in the user's TEMP directory detailing actions taken during the installation process. The log file is created with a unique name based on the current date (It will not overwrite or append to existing files). The information contained in the log file is technical in nature and therefore not intended to be understandable by end users.
/LOG="filename"	Same as /LOG, except it allows you to specify a fixed path/filename to use for the log file. If a file with the specified name already exists it will be overwritten. If the file cannot be created, Setup will abort with an error message
/LANG=language	Specifies the language to use (en or de). When a valid /LANG parameter is used, the Select Language dialog will be suppressed.

2.8.2 Installation Options

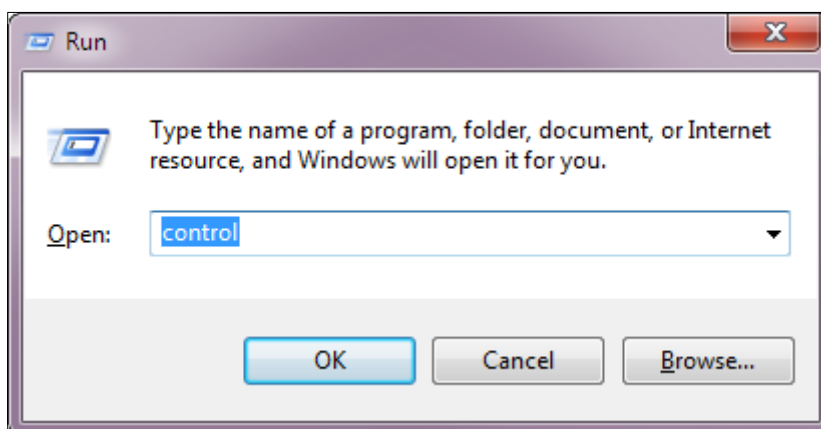
At the start of the installation, you are asked for the installation language. Currently an installation in English and German is supported. The installation language also defines the language of the installed documentation (if available in both languages).

The installer allows to choose between a *full installation*, a *compact installation* and a *custom installation*. The *full installation* installs everything. The *compact installation* installs only the files which are necessary for software development. The *custom installation* lets you choose which components of the SDK are to be installed. The categories currently available are *Tools*, *Documentation*, *Sample Code* and *Software Development* files.

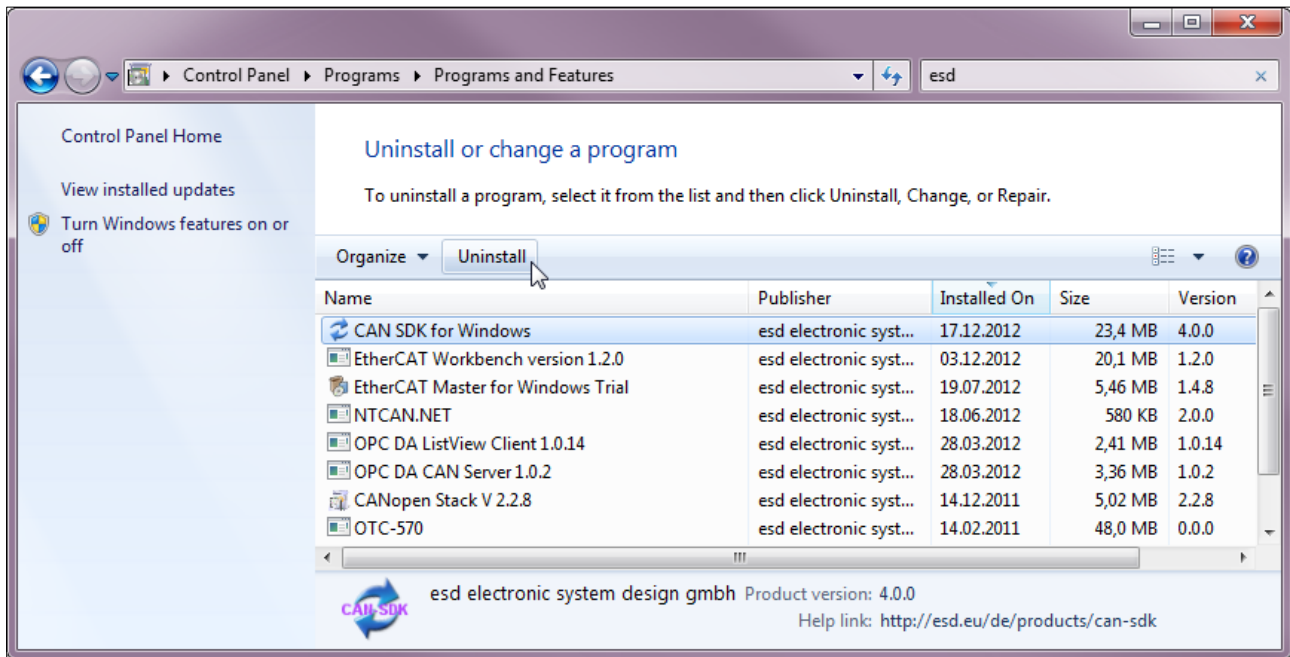
2.8.3 Uninstall

To uninstall the *CAN SDK* from your computer you must open the **Add/Remove Programs** (Windows 2000/XP) or **Programs and Features** (Windows Vista and later) dialogue via the **Control Panel**.

To open the **Control Panel** in all versions of Windows open the run dialogue by pressing the keys **WinKey + R** and type *control* as shown below for Windows 7.



Choose in the **Programs and Features** dialogue as shown below for Windows 7 the *CAN SDK*, click **Uninstall** and follow the steps of the wizard.



2.8.4 IDE Integration

Many Integrated Development Environments allow to define paths relative to an environment variable. For this reason, during the installation the environment variable `CanSdkDir` is created which is set to the installation directory of the CAN SDK. Using this environment variable in paths makes a project independent of the installation directory of the CAN SDK.

2.9 Digital Signatures

2.9.1 Overview

All binaries (device drivers, libraries (DLLs) and installers) of the current **esd** CAN device driver packages and the package itself are digitally signed. A digital signature is an electronic security mark that can indicate the publisher of the software, as well as whether someone has changed the original contents of the files. The code-signing technology built in Microsoft Windows operating system for this is called *Authenticode*.

Certificates issued by certification authorities (CA) trusted by Windows for the initial implementation of *Authenticode* used SHA-1 cryptographic hash functions. These have been deprecated by the National Institute of Standards and Technology (NIST) in 2011 because of significant mathematical weaknesses according to the collision resistance which allows brute force attacks to circumvent the security. For this reason, in 2015 Microsoft published a SHA-1 code signing certificate deprecation policy with the aim to migrate to SHA-2 code signing certificates in future and at this time still supported Windows versions. This SHA-1 deprecation policy distinguishes in the level of support between kernel mode code (device driver) and user mode code for the different major versions of Windows.

The table below gives an overview about the SHA-2 support for the different major Windows desktop versions. This table and everything said below also applies for the respective server versions.

OS	Windows XP	Windows Vista	Windows 7	Windows 8	Windows 10/11
User Mode	Yes	Yes	Yes	Yes	Yes
Kernel Mode	No	No	Yes	Yes	Yes

Table 12: Windows Support for SHA-2 Code Signing Certificates



Note:

SHA-2 user mode code signing support for Windows XP requires the installation of SP3. Windows 7 requires the KB3033929 update to be installed for SHA-2 signed kernel driver support. SHA-2 signed kernel driver support is not published by Microsoft for earlier versions of Windows.

Before Windows 10, device driver packages are signed with a code signing certificate and the cross-certificate of a CA which issued the certificate used for signing. This CA must belong to the group of CAs which are trusted by Microsoft. The mechanism of signing a device driver that way is referred to as **Cross Signing**.

Starting with Windows 10, Microsoft changed its general policy for kernel mode code signing by making it mandatory that all Windows kernel mode drivers must be submitted to and digitally signed by the Windows Hardware Developer Centre Dashboard Portal instead of performing an (in-house) cross signing. This portal only accepts (device driver) submissions with a valid EV Signing Certificate which are, since 2016, only submitted by the CAs based on SHA-2. This mechanism of signing device driver (which is only applicable for Windows 10 and later) is referred to as **Attestation Signing**.

Windows 10 enforces the policy of Attestation Signing after a grace period which ended with the release of Windows 10 1607 (aka *Anniversary Update*) and if the following two conditions are met:

- Fresh installation of Windows (no upgrade from a version before 1607)
- Secure Boot enabled

All cross-certificates trusted by Microsoft expired in July 2021 so Cross Signing device driver is no longer possible.

For CAN device driver released by **esd** the Microsoft kernel mode code signing policy has the following impact:

- All device drivers released before March 6, 2017 are “cross-signed” with an SHA-1 certificate issued to *esd electronic system design gmbh* and they should be accepted by all versions of Windows XP and later.
- All device drivers released after March 6, 2017 are “cross-signed” with an SHA-2 EV certificate issued to *esd electronic system design gmbh* and they should be accepted by all versions of Windows 7 with installed KB3033929 and later. **esd** can no longer provide device driver which install flawlessly on Windows versions before Vista.
- All device drivers released between May 29, 2019 and March 19, 2021 are “attestation-signed” with an SHA-2 EV certificate issued to *Microsoft Windows Hardware Compatibility Publisher* as well as “cross-signed” with an SHA-2 EV certificate issued to *esd electronics gmbh* and should be accepted by all versions of Windows 7 with installed KB3033929 and later.
- All device driver released after March 19, 2021 are “attestation-signed” with an SHA-2 EV certificate issued to *Microsoft Windows Hardware Compatibility Publisher* and should be accepted by all versions of Windows 10 and later.



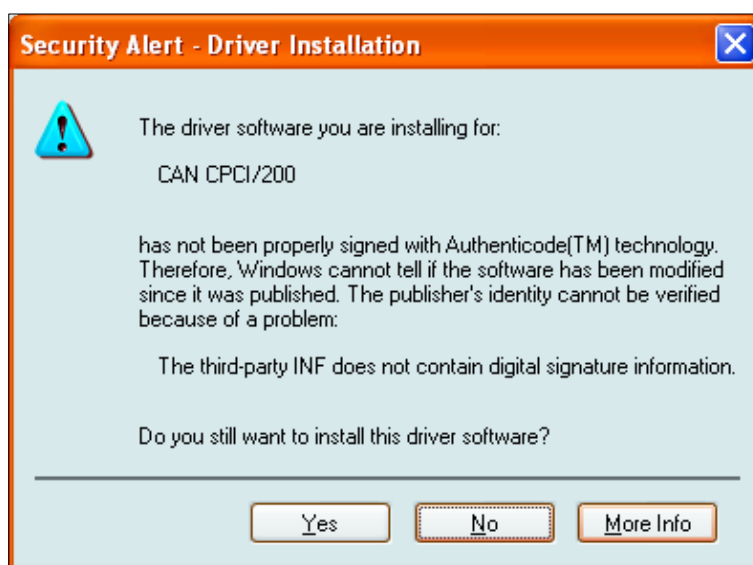
Note:

As Microsoft ended the support to allow in-house cross-signing of device driver code, it is technically no longer possible for esd to release new or updated device drivers for Windows versions before Windows 10.

2.9.2 Driver Installation

Windows indicates an invalid signature of a driver package in the following ways:

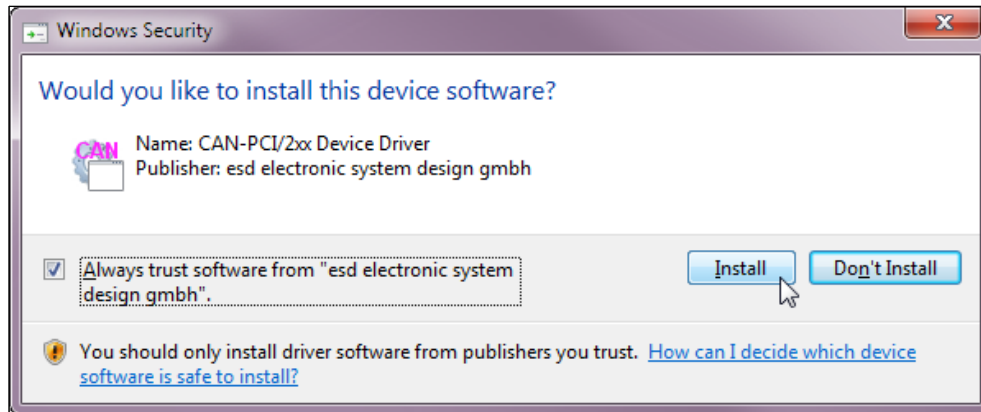
- **Windows rejects to install a device driver.** This can occur if one of the following conditions is met:
 - **Any 64-Bit Windows version:** The operating system fails to validate the digital signature of the driver package because for example one or more files were altered after the driver package has been digitally signed.
 - **Windows 10 1607 (32-/64-Bit) or later:** The device driver was not signed by the Windows Hardware Developer Centre Dashboard Portal on fresh installations of the operating system (no upgrade from a version before 1607) with enabled secure boot.
 - **Windows XP/Vista (64-Bit):** Device driver released after March 6, 2017 which are signed with an SHA-2 certificate which is not supported.
- **Windows can't verify the publisher of the device driver.** This occurs if a device driver either has no digital signature or it has been signed with a digital signature that could not be verified by a certification authority. On Windows versions before Vista during the process of driver installation, depending on the system configuration, a dialogue box may indicate that the *Authenticode* signature is invalid.



Windows versions before Windows Vista cannot validate a correct SHA-1 signature because the trusted chain of the certificates is not completely stored in them and they accept only a signature created by Microsoft during a WHQL certification process. In this special case you can ignore the warning and continue with the installation.

- **The device driver has been altered.** This occurs if files in the driver package are altered on 32-bit Windows versions after it was digitally signed by **esd**.

On Windows Vista and later after a successful validation of the certificate a message shows the name of **esd** as vendor that has signed the driver package.



You can optionally decide to add the **esd** certificate in the *Trusted Publishers* certificate store which will prevent this dialogue for other digitally signed **esd** software and for device driver updates

If you get an indication about an invalid signature (with the exception described for Windows versions before Vista) please contact **esd**.

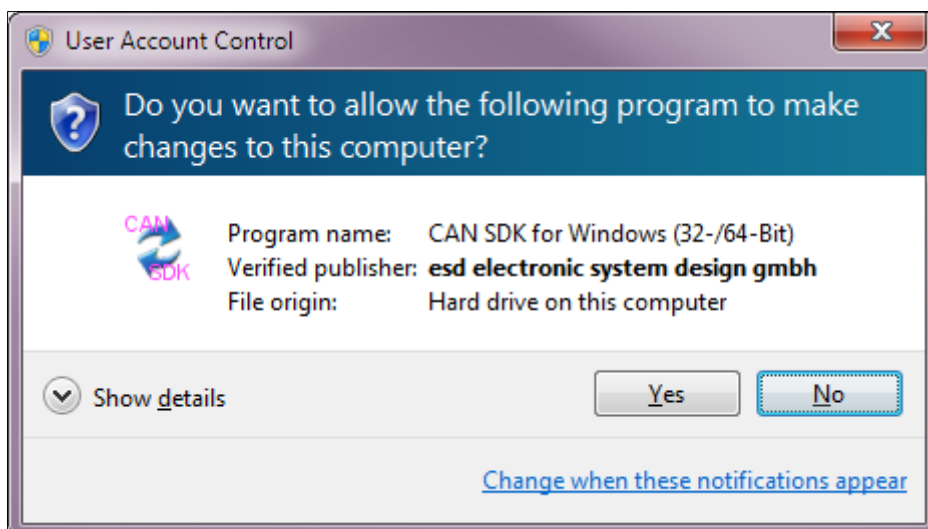


Note:

The 64 bit versions of Windows do not allow to install and use device drivers which are not digitally signed by the vendor which means that it is also not possible to just replace the driver .sys file with a newer version which is still possible on the 32 bit versions of Windows.

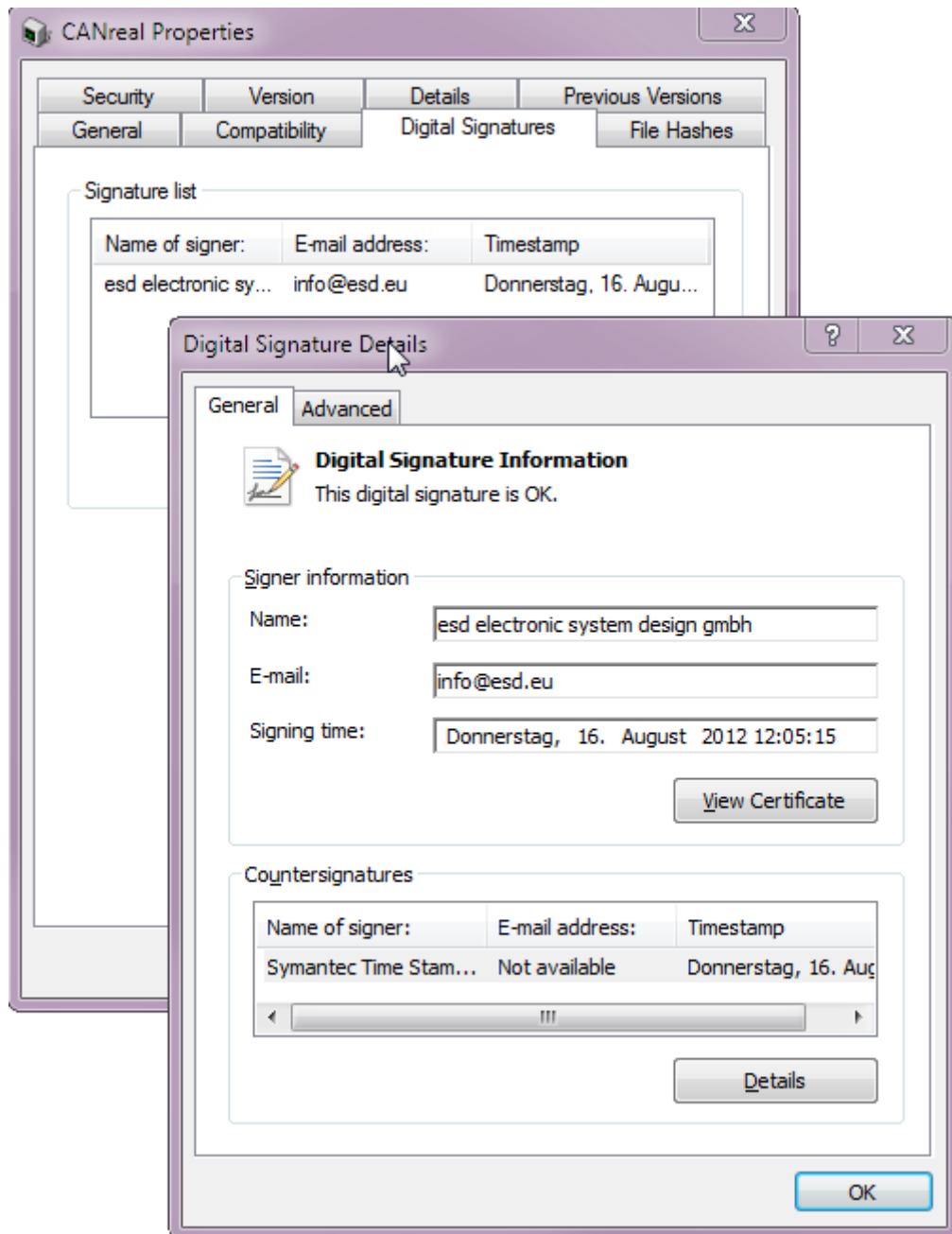
2.9.3 Software Installation

Windows installer based software by **esd** is also digitally signed and on UAC enabled systems (Windows Vista and later) you will see an UAC dialogue similar to the picture below which indicates that the **esd** is the publisher of the software and the binary was not altered by third parties since it was signed.



2.9.4 Digital Signature Verification

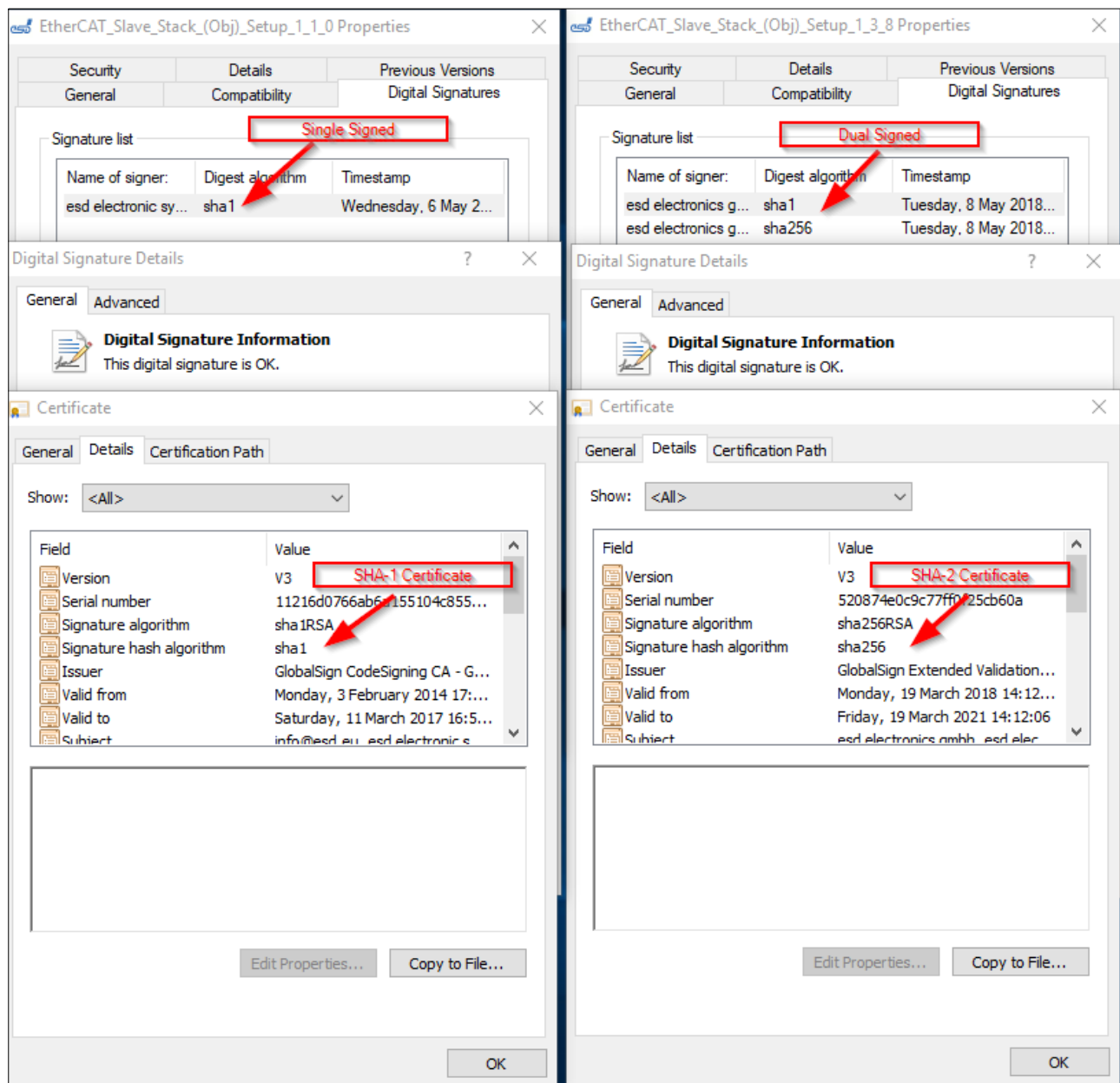
To view digital signatures of a binary, open the context menu of the file and select **Properties**, then go to the **Digital Signatures** tab.



The dialogue box provides information that **esd** is the software publisher, about the certification authority that issued the certificate and the date the code was signed (timestamp). Finally it indicates that the digital signature for the file is valid.

As described in chapter 2.9.1 the Windows XP SP3 with KB3033929 and Windows Vista support SHA-2 certificates at least for user mode code but they do not support the improved cryptographic hash functions which come with SHA-2 for the digest algorithm. For this reason, all binaries released by **esd** after March 6, 2017 are dual signed with the SHA1 and the SHA256 algorithm.

The picture below shows the difference between a single signed binary with SHA-1 certificate and a dual signed binary with SHA-2 certificate.



3 Unix® Operating Systems

This chapter covers the necessary steps to install, configure and start the device drivers for **esd** CAN interfaces available for UNIX® operating systems.

3.1 Linux®



Note:

For current information on the installation please check the readme in the according installation directory. For the changes introduced with the last service pack, please check the release notes.

Depending on the CAN hardware NTCAN support is either realized by device driver packages provided by **esd** (see chapter 3.1.2) or by the standard framework for CAN driver support in Linux (aka *SocketCAN*) together with a wrapper library (see chapter 3.1.3). The EtherCAN interface family requires a user mode driver which installation and configuration is described in chapter 3.1.4.



Linux CAN support (aka *SocketCAN*) is a set of *Open Source* drivers and a network stack which extends the BSD socket API in Linux by introducing the new protocol family **PF_CAN**. Since version 2.6.25 this framework is part of the vanilla (mainline) Linux kernel and can be included by compiling the kernel with **CONFIG_CAN**.

An introduction and overview about the Linux CAN implementation can be found in a presentation during the International CAN Conference (iCC) 2012 with the title “The CAN networking subsystem of the Linux kernel”.

Refer to Table 13 which esd CAN interfaces come with a native SocketCAN support.

The picture below gives an overview about using the **esd** NTCAN architecture compared to using the SocketCAN architecture together with the NTCAN wrapper.

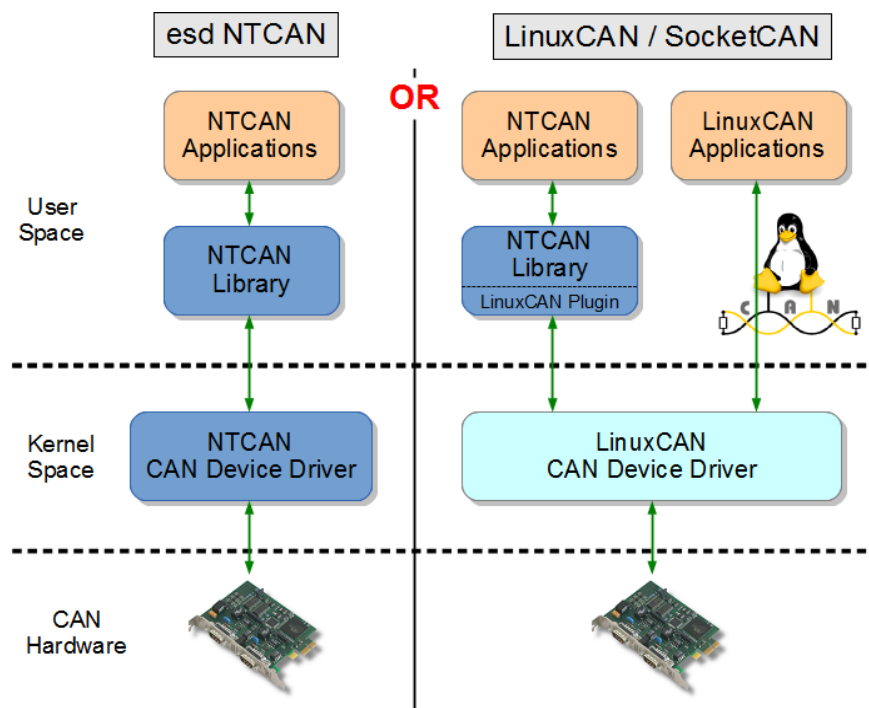


Figure 1: Native NTCAN Architecture vs. Linux CAN



Using the SocketCAN NTCAN wrapper requires that the SocketCAN driver for the **esd** CAN interface is working.

As this driver is maintained by the Linux (CAN) community the appropriate community support mechanisms like mailing lists, etc. should be consulted in case of installation problems for the SocketCAN part.

Problems with the NTCAN driver or the SocketCAN wrapper are handled by the **esd** support.

The CAN drivers provided by **esd** are intended for the standard Linux (vanilla) kernel. Linux real-time extensions (apart from the real-time preempt patches aka **PREEMPT_RT**) like RTAI, Xenomai, etc. are not officially supported.



Xenomai is a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications.

Part of the framework is the RT-Socket-CAN environment which supports hard real-time for CAN communication with **esd** CAN interfaces of the C200 CAN interface family. Please refer to the project's homepage for details about installation and API.

!! There is no NTCAN API and installation support for Xenomai by esd !!

Please contact **esd** for options to support further CAN interface families.

3.1.1 CAN Board Support Overview

Device drivers for **esd** CAN interfaces are available for x86 (32-Bit) and x64 (64-Bit) target architectures. The CAN interfaces are supported either by **esd** NTCAN driver or by LinuxCAN / SocketCAN and in some cases even by both architectures. The table below gives an overview which CAN interface family is supported by which driver architecture.

CAN Family	Driver Architecture	
	esd NTCAN	SocketCAN
C200I	✓	-
C331I	✓	-
C200	✓	✓
C331	✓	-
C360	✓	-
C400	✓	-
C402	✓	-
C405	✓	-
USB1*	✓	-
USB2***	-	✓
USB3†	-	✓
U400	-	-
EtherCAN	✓	-
AMC4	✓	-

Table 13: SocketCAN support for esd interfaces



Attention!

If a CAN interface is supported by LinuxCAN / SocketCAN and you want to use the **esd** NTCAN driver you must make sure that the device is not already used by the LinuxCAN / SocketCAN driver as otherwise starting the **esd** CAN driver will fail.

*Support ends with kernel version 2.6.24 for legal reasons.

**No support for CAN-USB/AIR2.

† Support is integrated in Linux kernel since kernel version 6.6.

In addition to the CAN interfaces **esd** also provides Linux BSPs for several embedded boards with on-board CAN interfaces. These boards are supported by NTCAN driver architecture which is part of the BSP. The table below gives an overview about the availability for different Linux versions and the included CAN driver version.

<i>Board</i>	<i>Linux</i>	<i>CAN Driver Version</i>
PMC-CPU/405(-DE)/440	>=3.2.0	3.x
CPCI-CPU/750	>=2.6.36	3.x
CPCI-405/EPPC-405	>=3.2.0	3.x
EPPC-405-UC	>=3.2.0	3.x
CAN-CBX-CPU52xx	>=3.2.0	3.x
CPCI-CPU/5201	>=3.2.0	3.x

3.1.2 NTCAN Driver

**Note:**

On most Linux installations the driver installation is only possible with superuser rights.

Please read the current `README` file that comes with the software!

Please note the drivers delivered on CD are most likely outdated. Increasing speed in Linux kernel development makes it almost impossible for us to provide you with drivers on CD, which work with all Linux versions and distributions. We rather recommend checking our website for the latest Linux drivers. To circumvent any problems before they occur, we advise you to visit this site before installing a driver from this CD.

The latest driver archives can be downloaded from <https://www.esd.eu>

Please note: Drivers released before July 2012 still come as encrypted ZIP archives. Newer drivers are released in the more commonly used TGZ format.

3.1.2.1 Files of the Linux Package

The software drivers for Linux are distributed on CD-ROM or delivered as archive via e-mail. The following files are contained:

File	Description
README	current notes and information
Makefile	driver compilation rules
config.mk	configuration file for the compile process It may be necessary to edit this file, in order to suit any peculiarities of the current system (mainly the <i>KERNELPATH</i> , see page 79)
libntcan.a	static CAN-API library (located in directory <i>./lib</i>)
libntcan.so	dynamic CAN-API library (located in directory <i>./lib</i>)
ntcan.h	header of the NTCAN-API/Library (located in directory <i>./lib</i>) This is the only header that has to be include in your application. Please do not use any defines located in any of the other headers, in order to keep your applications working with future version of the driver!
cantest.c	source code of the example-application 'cantest' (located in subdirectory <i>./example</i>) (see <i>/1/</i>)
cantest	binary of example-program 'cantest' (located in subdirectory <i>./bin</i>)
xxxx.o xxxx.c xxxx.h	source- and object-files (located in subdirectory <i>./src</i>) This driver is released as a combination of binary-objects (*.o) and source-files (*.c and *.h). This way esd can provide a CAN-driver working with many different Linux-kernels. The source files are NOT under the GPL (GNU Public Licence)! You are not allowed to modify, redistribute or sell the files! They are intellectual property of esd electronics gmbh. Attention! Do not try to use any defines or data-structures located in these files in your own sources. This will lead to non-working applications in the future.
updcrd	This tool is only delivered with CAN modules that are equipped with a local processor (e.g. CAN-PCI/331). It is located in subdirectory <i>./bin</i> . This tool can be used to switch the firmware of such a card between CAN-2.0A-firmware (used for reception of CAN-messages with 11-bit-identifier) and CAN-2.0B-firmware (used for additional reception of CAN-messages with 29-bit identifier). Syntax: <code>updcrd -tx net</code> Parameter: <code>crd:</code> CAN module ID, e.g. pci331, usb331 (see table on page76) <code>x:</code> 'a', if CAN 2.0A firmware 'b', if CAN 2.0B firmware <code>net:</code> Net number of the CAN interface in the system (0, 1, 2, ...)



Note:

In driver archives for x86_64-Linux the path for libraries and binaries exists twice: Once for 32-bit (*./lib32* and *./bin32*) and once for 64-bit (*./lib64* and *./bin64*).

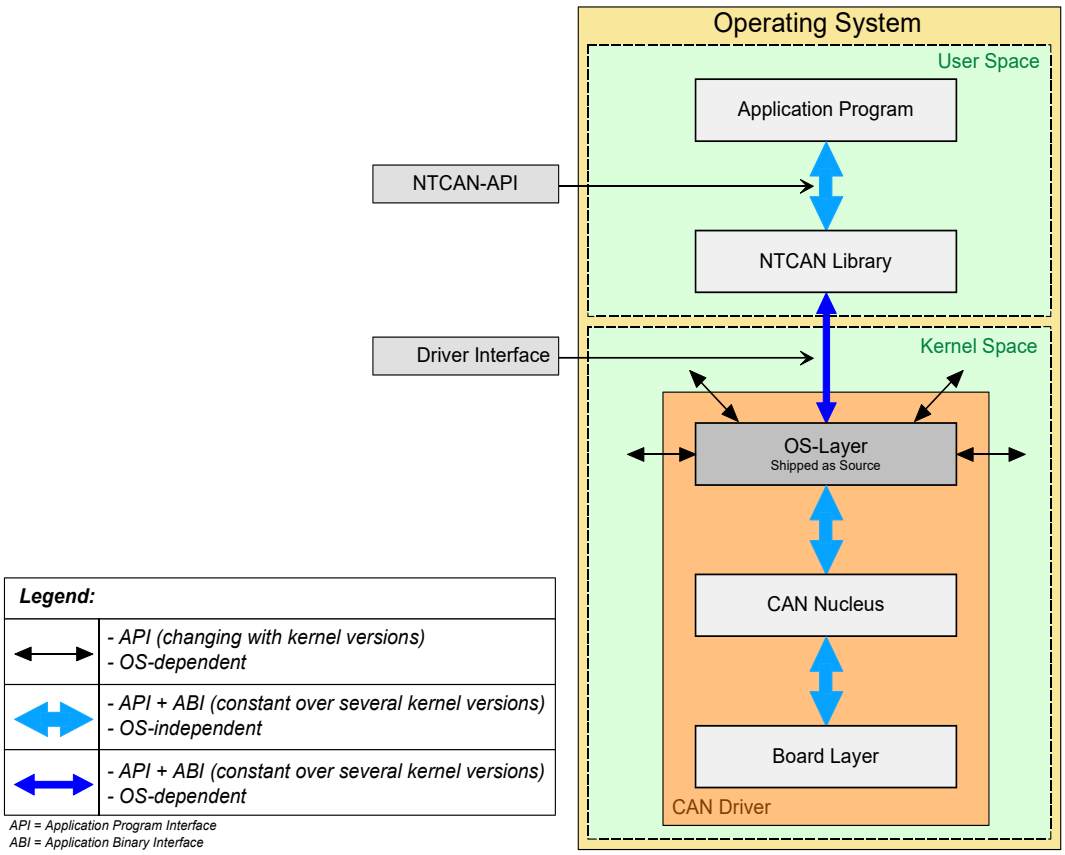


Figure 2: Linux driver architecture

3.1.2.2 CAN-Module-ID and Default Parameters of the Driver

CAN Module	Module ID <i>crd</i>	Default Values*		
		<i>major</i>	Address <i>io</i>	Interrupt <i>irq</i>
AMC-CAN4	amc4	54	–	–
CAN-ISA/200	isa200	53	0x1E8	7
CAN-PC104/200 (SJA1000 version)				
CAN-ISA/331 CAN-PC104/331	isa331	52	0x1E0	5
CAN-PCI104/200 CAN-PCI/200 CAN-PCle/200 CPCI-CAN/200 CAN-PCI/266 PMC-CAN/266	pci200**	54	–	–
CAN-PCI/331 CPCI-CAN/331 PMC-CAN/331	pci331	50	–	–
CAN-PCI/360 CPCI-CAN/360	pci360	51	–	–
CAN-PCI/400 CAN-PCle/400 CPCI-CAN/400 PMC-CAN/400	esdaccbm	55	–	–
CAN-PCI/405	pci405	53	–	–
CPCI-405 (local driver)	cpci405	53	–	–
CPCI-CPU/750 (local driver)	cpci750	53	–	–
CAN-USB/Mini	usb331	50	–	–

*The default values can be overwritten by the command `insmod` (see following chapter).

**Before installing the NTCAN driver for `pci200` please check, that the CAN interface is not yet used by the SocketCAN driver.

3.1.2.3 Installation



Note:

It is necessary to install the kernel sources and configure them to comply with the running kernel, before installing the CAN-driver!

Unpacking the Archive

Unpacking the TGZ archive (for drivers released after July 2012) with

```
tar -xzf esdcan-crd-os-arch-ver-ext.tgz
```

crd = card-id (e.g.: *pci200* or *cpci405*, see table in chapter 3.1.2.2)

os = host-operating-system (e.g.: *linux_2.4.x*)

arch = host-architecture (e.g.: *x86* or *x86_64*)

ver = driver version (e.g.: *3.7.2*)

ext = extension (applicable to certain cards only, e.g.: *gcc2*)

You'll end up with a directory named as the archive.

Unpacking the ZIP archive (for drivers released prior July 2012) with

```
unzip esdcan-crd-os-arch-ver-ext.zip
```

crd = card-id (e.g.: *pci200* or *cpci405*, see table chapter 3.1.2.2)

os = host-operating-system (e.g.: *linux_2_4_x*)

arch = host-architecture (e.g.: *x86* or *x86_64*)

ver = driver version (e.g.: *3.7.2*)

ext = extension (applicable to certain cards only, e.g.: *gcc2*)



Note:

You will be prompted for a password. The password can be found in the accompanying file `README_FIRST` (delivered on the CAN-CD or per e-mail).

Resulting file: `esdcan-crd-os-arch-ver-ext.tar`

Untar the Driver Directory

```
tar -xvf esdcan-crd-os-arch-ver-ext.tar
```

crd = card-id (e.g.: *pci200* or *cpci405*, see table chapter 3.1.2.2)
os = host-operating-system (e.g.: *linux_2_4_x*)
arch = host-architecture (e.g.: *x86* or *x86_64*)
ver = driver version (e.g.: *3.7.2*)
ext = extension (applicable to certain cards only, e.g.: *gcc2*)

You will end up with a directory named as the archive.

Unzip and Untar in a Single Step

Alternatively, the unzipping and untaring of the driver directory can be accomplished in a single step with:

```
unzip -p esdcan-crd-os-arch-ver-ext.zip | tar -xv
```

The unpacked files will be stored in a directory that carries the same name as the archive file.

Compiling the Driver

```
cd ./esdcan-crd-os-arch-ver-ext
```

In some cases, you need to edit a configuration file for the compilation: In `config.mk` you need to set the variable `KERNELPATH` correctly. Normally the default path should be correct. If your Linux configuration differs from standard, correct the following line accordingly:

```
KERNELPATH = <your-path-to-the-kernel-source>
```



Note:

For Linux kernel > 2.6.0:

On some systems you might need to be "root" to compile the driver.

Compilation of the driver is simply started by typing:

```
make
```

For some cards there are warnings like `COMPILING FOR xxx`. These can be ignored and will be removed in future versions.

Now, you have a file called as described below, which is the actual driver-module in the same directory:

```
esdcan-crd-os-arch-kver
```

Dynamically loadable driver-file with:

`crd` = card-id (e.g.: `pci200` or `cpci405`, see table chapter 3.1.2.2)

`os` = host operating system (e.g.: `linux...`)

`arch` = host architecture (e.g.: `x86`)

`kver` = target-version information (e.g.: `2.4.18`)

For Linux the kernel version is coded here, because the compiled version is kernel specific!

Example:

For a CAN-PCI/331 for 32-Bit-Linux on x86 with 2.4.21-99-smp kernel the driver with version 3.6.1 is called as following:

```
esdcan-pci331-linux-x86-3.6.1-2.4.21-99-smp
```



Note:

For Linux kernel > 2.6.0:

The driver file is called `esdcan-crd.ko` and is generated inside of the `src-` subdirectory.

Example: (for Linux kernel > 2.6.0)

For the above mentioned CAN-PCI/331 the driver file is called:

`esdcan-pci331.ko` (the file is located in `./src` subdirectory)

1. File Locations

It is recommended (though not necessary) to store the driver module in the following directory:

```
/lib/modules/kernelversion/
```

The variable `kernelversion` has to be replaced by the according string of the system. The string (on kernels 2.4.x it should be equivalent to the `os`-string in the driver's name (see above)) is returned, if the following command is called:

```
uname -r
```

The dynamic shared library `libntcan.so` should be placed in the directory `/usr/local/lib/` or an equivalent path, which is contained in the `LD_LIBRARY_PATH` environment variable.



Note:

On 64-bit systems, there are two versions of `libntcan.so`. One in `./lib32` and one in `./lib64`. The first belongs into `/usr/local/lib` on most Linux distributions. The later should be kept together with other 64-bit libraries, e.g. in `/usr/local/lib64`.

The static version of the library `libntcan.a` can be kept wherever you want. Here at **esd** we prefer to keep it with the sources of a project, on the other hand, one might like to install it with the shared-lib at `/usr/local/lib/`.



Note:

esd discourages the use of statically linked libraries. We rather recommend to make use of the dynamically linked libraries. Your advantages will be much easier backtracking of involved versions and much simpler update procedures.



Installation Note:

The shared library should belong to user and group "root" with the following file access permissions: `u=rwx, g=rx, o=rx`

After installation of the library, the root-user should call:

```
ldconfig -n /usr/local/lib (if installed to this directory)
```

Afterwards there is a link `libntcan.so.v --> libntcan.so.v.mv.r`.

For your own convenience it is advised to generate another link in your library-directory:

```
libntcan.so --> libntcan.so.v
```

The static-library, if installed in `/usr/local/lib/`, should also belong to user/group root, but it does not need (and should not have) the executable-flag. Leading to the following file access permissions: `u=rw, g=r, o=r`

2. Load the Driver File (as Superuser)

Syntax:

```
insmod ./esdcan-crd-os-arch-kver [major=m]
```

with the following optional parameter: *m* = non-default major

The naming of the kernel module is equivalent to the naming of the driver archive as printed on page 76 (exception on kernels > 2.6.x).



Note:

With the module CAN-PCI/405 this call returns after approx. 5 seconds!

3. Make the 'inodes' (as Superuser)



Note for Systems with Kernel 2.6.x:

On 2.4.x systems this step has to be executed just once. On 2.6.x systems the inodes might vanish after reboot. If this is the case on your system, please do the following:

Instead of `/dev` rather change into `/lib/udev/devices` directory and create the inodes there. In this way they will be automatically recreated on every reboot.

```
cd /dev
```

```
mknod --mode=a+rw can0 c xx 0
```

```
mknod --mode=a+rw can1 c xx 1    ==>  as many as physical CAN nets provided
                                     by the modules
```

with

xx = major number of the driver (see table on **page 76**)



Note:

For your own convenience as soon as the driver has been loaded (Step 4 complete) there is a script in proc-filesystem (`/proc/bus/can/XXX/inodes`, where *XXX* is a subdirectory depending on your CAN device), which relieves you of this step. The script also has a parameter to specify the starting net number. It will handle this step for multiple CAN devices as well.

6. Checking the Installation

Whether the installation has been successful or not, can be checked in the following file:

`/var/log/messages`

or by calling `dmesg`

Here is an example for a successfully loaded driver for CAN-PCI/405:

```
kernel: esd CAN driver: pci405
kernel: esd CAN driver: baudrate not set
kernel: esd CAN driver: mode=0x00000000, major=53, 4 nodes on 1 cards
kernel: esd CAN driver: version 0.3.1 14:36:35 Feb 13 2003: successfully loaded
```

After a successful installation, the CAN bus can be accessed by means of the NTCAN-API. The application has to be linked to the library `libntcan.a` (static) or `libntcan.so` (shared).



Note:

esd strongly recommends usage of dynamically linked libraries (aka shared objects)!

If the example application `cantest` is called without parameters, the available CAN nets are displayed.

7. Unload the Driver File (as Superuser)

Kernel 2.4.x:

```
rmmod esdcan-crd-os-arch-kver
```

Kernel 2.6.x:

```
rmmod esdcan-crd.ko
```

3.1.3 Linux CAN Driver (aka SocketCAN)

3.1.3.1 Integration

The NTCAN library plugin capability is used to provide NTCAN support for **esd** CAN interfaces that are supported natively with the CAN driver implementation which is part of the Linux kernel usually referred to as *SocketCAN*. The plugin approach provides support for most basic features of the NTCAN API (see chapter 3.1.3.4 for limitations and differences) so NTCAN based applications can run in parallel with SocketCAN based applications and the hardware can be used in parallel with other **esd** CAN interfaces which are supported with **esd** NTCAN drivers. Refer to Table 13 which esd CAN interfaces come with a native SocketCAN support.

The following additional requirements have to be met:

1. Since version 2.6.25 the CAN support is part of the Linux kernel and can be enabled if compiled with `CONFIG_CAN`. Support for previous kernel versions might be available via the no longer updated web site of the SocketCAN project.
2. "ip"-tool supporting can
 The install script tries to detect/build this automatically. If that help fails please go to `iproute2` directory and check **READMEs** there to build it manually.
 If `make` is successful the file `.../ip/ip` should be built. Copy it to a binary path, e.g. `/usr/local/bin` then. All usages of just `ip` here might need that path then, too. (So `/usr/local/bin/ip` instead if just `ip` must be typed)

3.1.3.2 Installation

1. Make sure SocketCAN is available, and a device driver is loaded, etc., e.g. by
`ls /sys/class/net/`. When there is an entry `can0` everything should be fine, else don't try to install this NTCAN plugin before it exists.
2. Extract the archive, e.g. by
`tar xvfz ntcansckplugin32-2.0.7-ntcan-3.3.6.tgz`
3. Change into the newly created directory and run the install script, e.g. by
`sudo ./install.sh`

3.1.3.3 Configuration

The configuration is done with the file `/etc/esd-plugin`

By default 3 nets are configured: SocketCAN device `can0` is configured as NTCAN net 60, `can1` is configured as NTCAN net 61 and so on.

To change the logical net numbers just edit the corresponding lines in the file:

- `libntcanSckPlugin.0.Net=60`
- `libntcanSckPlugin.1.Net=61`
- `libntcanSckPlugin.2.Net=62`

Other settings in that file:

- Verbosity level (Text output in the console):
 - `libntcanSckPlugin.verbose=0` Prints nothing
 - `libntcanSckPlugin.verbose=1` Only errors
 - `libntcanSckPlugin.verbose=2` Errors and warnings
 - `libntcanSckPlugin.verbose=3` Errors, warnings and Infos
- Automatically adapting Socket-CAN settings:
 - `libntcanSckPlugin.noscc=1` To disable that

Use the *cantest* application described in [11](#).which binary is extracted to the sub folder `cantest` to verify the correct installation. Run it without parameters to see a short help and a list of available CAN nets.

3.1.3.4 Restrictions

Due to the different driver architecture the SocketCAN wrapper can not map all capabilities of the NTCAN API because this feature is either not supported (e.g. error injection) or is implemented in a way that it can not be easily mapped (e.g. count of lost frames). This chapter contains (a maybe incomplete) list of differences between the SocketCAN wrapper and native NTCAN drivers. The NTCAN API is described in */1/*.

- Lost counter in `CMSG/CMSG_T` structs is not used (always zero).
Use e.g. `sys/class/net/can0/statistics` for information about lost frames.
- Events: `NTCAN_EV_CAN_ERROR` is the only event supported. And within that event's data byte one (*error*) is the only supported byte.
- Only these I/O controls are supported (See NTCAN docs **canIoctl**):
 - `NTCAN_IOCTL_GET_BAUDRATE`
 - `NTCAN_IOCTL_SET_BAUDRATE`
 - `NTCAN_IOCTL_FLUSH_RX_FIFO`
 - `NTCAN_IOCTL_SET_20B_HND_FILTER`
 - `NTCAN_IOCTL_GET_TIMESTAMP`
 - `NTCAN_IOCTL_GET_TIMESTAMP_FREQ`
 - `NTCAN_IOCTL_GET_RX_TIMEOUT`
 - `NTCAN_IOCTL_GET_TX_TIMEOUT`
 - `NTCAN_IOCTL_SET_TX_TIMEOUT`
 - `NTCAN_IOCTL_SET_RX_TIMEOUT` (While `TX_TIMEOUT` is ignored, see **canWrite()**)

- Bus-OFF handling
Current CAN's default is to stay off bus then. The application is usually helpless in that case. To avoid this, set Socket-CAN to automatically restart the device then. Currently this can be done with the `ip` command:
e.g.:

```
ip link set can0 type can restart-ms 1000
```

(Manual restart possible by `ip link set can0 type can restart`)

This is also done automatically. See chapter 3.1.3.3 for infos about how to avoid that.

- **canStatus()**
Resulting `CAN_IF_STATUS` struct members:
 - *hardware* and *firmware* as described or 0 when they could not be determined.
 - *driver* is the wrapper plugin version number.
 - *dll* is set by NTCAN library.
 - *boardstatus* is one of enum `can_state` defined in `netlink.h` (or `0xffffffff` if it could not be determined)
 - *boardid* is "SocketCAN".
 - *features* is `NTCAN_FEATURE_BASIC_20B | NTCAN_FEATURE_TIMESTAMP`
Where it is not guaranteed that the underlying hardware really supports CAN 2.0B or hardware timestamps. (Else timestamps will be set by software.)

- **canOpen()**

The flags, *txqueuesize* and *txtout* parameters are ignored. Socket-CAN's TX queue size can be set with the `ip` tool, e.g.:

```
ip link set can0 txqueuelen 1000
```

The receive buffer size depends on the given *rxqueuesize* value. But does not match exactly that number of frames. Also it is limited by *rmem_max* value, which can be increased with e.g.

```
echo 1048576 >/proc/sys/net/core/rmem_max
```

These example values are also written automatically. See chapter “3.1.3.3 Configuration” for infos about how to avoid that.

- **canSetBaudrate()**

Only *Pre-defined bit rate table* and *User Bit Rate Numerical* are allowed. Usually only possible as "superuser". (e.g. when application is started with `sudo`) Also (re)starts the SocketCAN net interface if it was down or controller was Bus-OFF.

Stops the Socket-CAN net interface with `baud param NTCAN_NO_BAUDRATE`.

- **canGetBaudrate()**

Returns `NTCAN_NO_BAUDRATE` when controller is Bus-OFF or Socket-CAN net interface is down.

- **canRead()/canReadT()**

When function is waiting for a message, it is not interrupted when another thread closes the same NTCAN handle.

- **canWrite()/canWriteT()**

The TX timeout is ignored. It is blocking until message is written to socket. While this is fine for safely sending more messages than buffer could hold, it is still no guarantee they really went on the bus. (As we're only waiting till they're in the buffer/queue)

Does not return `NTCAN_CONTR_OFF_BUS/NTCAN_CONTR_WARN`. It will return `NTCAN_SUCCESS` even when controller is Bus-OFF! To avoid this use the error events and stop sending when controller goes off bus.

- CAN frame data length counter > 8

SocketCAN does not allow this, so when `CMSG/CMSG_T` member *len* is set to a value between 9 and 15 it is treated as 8.

3.1.4 EtherCAN and EtherCAN/2

In comparison to CAN interfaces connected to a local PC bus (PCI, USB, ...) supported with a Linux kernel mode device driver the EtherCAN and EtherCAN/2 interfaces are supported with a user mode device driver which integrates this remote CAN hardware into the NTCAN architecture in the same way as a local interface. This user mode device driver supports the EtherCAN/2 as well as the legacy EtherCAN hardware but for reasons of simplicity this chapter only refers to the EtherCAN/2.

The EtherCAN/2 software package is available for Linux (x86/x64) and contains the following files:

<i>File</i>	<i>Description</i>
README.x.x	Current notes and information
cantest.c	Source code of the example-application 'cantest' (located in subdirectory ./example) (see /1/.)
cantest	Binary of program 'cantest' (located in subdirectory ./bin)
makefile.ethercan	Example makefile for compiling the file cantest.c
installEthercanLibs	Bash script for installation of libraries and include files
etc/esd-plugin	Example config file for ntcaneThPlugin (to be stored as /etc/esd-plugin)
pdf/*.pdf	NTCAN API documentation (Part 1) and Installation (Part 2) and documentation of esd system abstraction layer API
include/ntcan.h	Header of the NTCAN API (to be stored e.g. at /usr/local/include). This is the only header that has to be include in the application. Please do not use any defines located in any of the other headers, in order to keep your applications working with future version of the driver!
lib/libntcan.so.x.x.x	Shared library containing the NTCAN-API (to be stored e.g. at /usr/local/lib)
lib/ntcaneThPlugin.so.x.x.x	Dynamically loadable plugin for libntcan.a or libntcan.so (to be stored at e.g. /usr/local/lib)
psys_linux/include/psys.h psys_linux/include/ psyslinux.h	Psys header files, esd System Abstraction Layer API (to be stored e.g. at /usr/local/include)
psys_linux/lib32/ libpsys.so.x.y.z	32-bit shared library, containing esd System Abstraction Layer (to be stored at e.g. under /usr/local/lib)
psys_linux/lib64/ libpsys.so.x.y.z	64-bit shared library, containing esd System Abstraction Layer (to be stored e.g. at /usr/local/lib64)
psys_linux/src/psysdrv.c	Source of the psys-driver
psys_linux/src/psysdrv.h	Psys-driver header
psys_linux/src/Makefile	KBuild-Makefile needed for the module generation with kernel 2.6.x
psys_linux/Makefie	makefile for PSYS-driver for kernel 2.4.x or 2.6.x
psys_linux/README	Release notes and installation hints for psys / psys-driver
psys_linux/LICENSE	License covering PSYS driver and library

3.1.4.1 Installation

**Note:**

The installation is only possible with superuser rights (user: root). Please read the current `README` file that comes with the software!

1. Unpacking the Archive

In order to unpack the zipped tar-archive file the following command has to be called:

```
tar xvfz ethercan-lx-2.0.10-ntcan-3.0.6- psys-1.3.0-gcc-3.3.1-glibc-2.3.2.tgz
```

The unpacked files will be stored in a directory that carries the same name as the archive file.

2. Compile and Load Psys-Driver

Therefore follow the installation instructions in *psys_linux/README*.

3. Installing the Libraries and Header Files

In order to install the libraries and header files the following command has to be called (with supervisor rights):

```
./installEthercanLibs
```

All library files will be stored in `/usr/local/lib`
and all include files will be stored in `/usr/local/include`.

4. Adapt `/etc/esd-plugin`

See chapter “3.1.4.2 Configuration”.

5. Compiling the Example Program ‘cantest’

With the call

```
make -f makefile.ethercan
```

the test- and example program `cantest` will be compiled.

After successful installation you can access the CAN-Bus via the esd NTCAN API (link `libntcan` with your application). For a complete NTCAN API documentation refer to `/1/`.

3.1.4.2 Configuration

All user configurable stuff concerning EtherCAN can be found in the file `/etc/esd-plugin`.

List of available keywords in `/etc/esd-plugin` (with $0 < x < 4$):

Keyword	Description	Default Value
PeerName[x]	host name or IP-address of EtherCAN server	-
Net[x]	CAN net number assigned to EtherCAN server with above PeerName[x]	50 + x
ConnTimeout[x]	Time to wait until connection to the EtherCAN server is established. If timeout exceeds, <code>canOpen()</code> returns <code>NTCAN_SOCK_CONN_TIMEOUT</code> .	2500 ms
CmdTimeout[x]	Timeout for special commands send from client to EtherCAN-server.	2500 ms
KeepAliveTime[x]	If there is no CAN traffic, client sends a keep-alive message to server. If sending the keep-alive message fails, the connection to the server is disconnected and the EtherCAN client will try to reconnect the server.	2500 ms
TCPNoDelay[x]	0: Nagle algorithm active (i.e.: Coalesce a number of TCP messages and send them all at once) 1: Nagle algorithm off (i.e.: Immediately send TCP messages without any inhibit)	1

Example 1: EtherCAN configured as CAN net 30 and all other parameters in default setting.

```
PeerName[1]=      "10.0.16.58"
Net[1]=           30
```

Example 2: EtherCAN configured as CAN net 20 and, with increased timeouts, because it is located outside the company network.

```
PeerName[0]=      "134.66.177.1"
Net[0]=           20
KeepAliveTime[0]= 10000 # increase keep-alive timeout
ConnTimeout[0]=   25000 # increase connection timeout
CmdTimeout[0]=    5000  # increase command timeout
```

3.1.4.3 Miscellaneous

This chapter covers several topics which should preclude problems using the EtherCAN/2 on Linux in your applications.

3.1.4.3.1 Linking Against `libntcan` (gcc-Option `-rdynamic`)



Attention!

It is mandatory to use the option `-rdynamic` when linking against `libntcan`, because the dynamically loaded library `ntcanEthPlugin.so` (beside delivering some new functionality) itself needs some symbols from within `libntcan`. Without `-rdynamic` this does not work!

If your application (on runtime) complains about `'ntcanEthPlugin.so: undefined symbol: openRegistryCanIf Ether'` the option `-rdynamic` is still missing in your makefile.

4 Real-Time Operating Systems

This chapter covers the necessary steps to install, configure and start the device drivers for **esd** CAN interfaces available for the real time operating systems described in this chapter. In comparison to Windows or Unix operating systems, described in the previous two chapters, the application is usually developed on a host system which is different from the (embedded) target system the device driver and the application runs on.

In comparison to the previous chapters which described the driver installation and configuration for **esd CAN Interfaces** this chapter covers **esd CAN Boards** (**esd** CAN Interfaces and embedded systems) as well as CAN driver developed for customer hardware.

4.1 VxWorks®



Note:

For the changes introduced with the last service release, please check the readme of the current release.

This chapter covers the necessary steps to install, configure and start the device drivers for **esd** CAN boards supporting the real time operating system Wind River VxWorks.

CAN device drivers are available for VxWorks 5.x, 6.x and 7.x for different CPU target architectures.

The architectural differences between VxWorks 5.x and VxWorks 6.x with and without VxBus support are covered in separate chapters.



Note:

Wind River offers the middleware component '*Wind River CAN for VxWorks 6.x*' as part of the VxWorks 6.x platforms '*Wind River Platform for Automotive Devices*' and '*Wind River Platform for Industrial Devices*'. Only the **esd** CAN interfaces CAN-PCI/200 and CAN-PC104/200 are supported directly by this Wind River implementation.

The esd NTCAN drivers do not rely on 'Wind River CAN' and can be used with any VxWorks platform. These device drivers are required if you want to use the optionally available higher layer CAN protocol stacks (CANopen, ARINC825, J1939, etc.) by **esd**.

A VxWorks CAN driver package supports one or more families of **esd** CAN Interfaces for a certain target architecture. The package contains a file `relnotes.htm` in HTML format which contains the revision history of the drivers and late-breaking information which did not make into one of the manuals. Please also read this file before installing the driver.



Note:

The CAN device driver for **esd** embedded boards with support for VxWorks are part of the BSP and are not deployed as a separate CAN driver package.

4.1.1 CAN Board Support Overview

Device drivers for **esd** CAN interfaces are available for different VxWorks versions and CPU architectures. A single device driver often supports more than one CAN interface type (refer to chapter 1.4 for the CAN interfaces which belong to the same interface family). For VxWorks 6.x device drivers with and without VxBus support are available.



Note:

If a combination of CAN interface, VxWorks version and/or CPU architecture is currently marked as not supported in the tables in this chapter please contact the **esd** support for help.

All VxWorks 5.x and 6.x non-VxBus device drivers are based on the 2.x version of the **esd** NTCAN driver architecture (see chapter *Driver History* in *I1I*). For VxWorks 6.x revision 6.6 or later is required.

VxWorks	5.4.x			5.5.x		6.x (UP, no VxBus)		
Architecture	386/486	Pentium	PPC	Pentium	PPC	Pentium	Pentium4	PPC
CAN Family								
C200I	✓	-	-	✓	✓	✓	-	-
C331I	✓	✓	-	✓	-	✓	-	-
C200	✓	✓	-	✓	-	✓	✓	✓
C331	✓	-	✓	✓	✓	✓	✓	✓
C360	✓	-	-	-	-	-	-	-
ICAN4	-	-	✓	-	✓	-	-	✓

The VxWorks VxBus-enabled CAN device driver support uniprocessor (UP) and symmetric multiprocessor (SMP) versions of VxWorks. Minimum requirement is a VxWorks version which supports at least VxBus 4 which was introduced with VxWorks 6.7. All drivers are based on the 3.x version of the **esd** NTCAN driver architecture (see chapter *Driver History* in *I1I*).



Attention!

For VxWorks 6.x UP the C200/C331 CAN interface family is supported by a VxBus and a non-VxBus (legacy) driver. Make sure you never include both device driver types in the same VxWorks image.

<i>VxWorks</i>	6.x (UP, VxBus)					6.x (SMP, VxBus)		
Architecture	Pentium	Pentium4	Core	Nehalem	PPC	Pentium4	Core	Nehalem
CAN Family								
C200	✓	✓	✓	✓	✓	✓	✓	✓
C331	✓	✓	✓	✓	✓	✓	✓	✓
C400	✓	✓	✓	✓	✓	✓	✓	✓
C405	✓	✓	✓	✓	✓	✓	✓	✓

<i>VxWorks</i>	7.x (UP, VxBus)					7.x (SMP, VxBus)		
Architecture	Pentium	Pentium4	Core	Nehalem	PPC	Pentium4	Core	Nehalem
CAN Family								
C331	✓	✓	✓	✓	✓	✓	✓	✓
C402	✓	✓	✓	✓	✓	✓	✓	✓

In addition to the CAN interfaces **esd** also provides VxWorks BSPs for several embedded boards with on-board CAN interfaces. These boards are also supported with the NTCAN driver architecture but the CAN device driver itself is part of the BSP. The table below gives an overview about the availability for different VxWorks versions and the included CAN driver version.

Board	VxWorks	BSP version	CAN Driver Version
PMC-CPU/405	5.5	1.2/15	2.x
PMC-CPU/440	>= 6.8	2.0/5	2.x
CPCI-CPU/750	5.5, >= 6.7	1.2/02, 1.2/02	3.x, 3.x
EPPC-405	5.4, 5.5, >=6.5	1.2/8, 1.2/11,2.0/10	2.x, 2.x, 2.x

4.1.2 Driver Integration

This chapter describes how the **esd** CAN device drivers can be included into a VxWorks project. The integration and installation of the drivers and libraries is different between VxWorks 5.x and VxWorks 6.x/7.x.

For VxWorks 5.x the driver and libraries are deployed as binaries which have to be linked to your application or BSP together with a configuration file which has to be adapted to the target hardware.

For VxWorks 6.x/7.x the driver installation and configuration is integrated into the Wind River Workbench.

4.1.2.1 VxWorks 5.x

The driver software for VxWorks 5.4.x and 5.5.x is deployed as Downloadable Kernel Modules (DKM) which have to be linked to the VxWorks image or can be loaded on application startup. An integration into Tornado is not supported. The driver comes as a CAN Interface specific package with the directory structure `/vw5x/CPU-Architecture/`.

A CAN device driver package for VxWorks 5.x contains the following files where `<drvname>` is the device family specific driver name following driver naming convention I (see chapter 1.4).

<i>File</i>	<i>Description</i>
<code>Ldcan</code>	Script to load driver and NTCAN library.
<code><drvname>.sys</code>	CAN driver (binary).
<code><drvname>.ini</code>	CAN driver configuration and start code (binary)
<code>caninit.c</code>	CAN driver configuration and start code (source of <code><drvname>.ini</code> binary)
<code>ntcan.o</code>	NTCAN library (binary)
<code>ntcan.h</code>	Header of the NTCAN-API for application development
<code>cantest.c</code>	Source code of example program <i>canTest</i>
<code>cantest</code>	Binary file of example program <i>canTest</i>

Depending on the released version of the driver there may be some postfixes on the `CPU-Architecture` part of directory names that denote build variants of the same driver for a certain `CPU-Architecture`. Here are some examples:

- `_SHRD`: The driver was built to use ***pcilntConnect()*** instead of ***intConnect()***. An example is `PENTIUM_SHRD`. The other version that uses ***intConnect()*** will be named `PENTIUM`. This is valid only if both variants for one `CPU-Architecture` exist. In any other cases the default value for the architecture is used.
- `_LONG`: The driver was built with the “-mlongcall” option for the C-Compiler that circumvents the 32MB branch distance limitation on the PowerPC architecture. An example is `PPC604_LONG`.

4.1.2.2 VxWorks 6.x

The driver software for VxWorks 6.x with and without VxBus support comes as a package for all supported host CPU architectures and CAN hardware with the following directory layout structure:

Directory	Description
doc\	Documentation of driver installation and NTCAN-API
src\	Example code (<code>cantest.c</code>) also included in binary format which can be optionally added to your project.
target\	This directory contains all necessary files to integrate and configure the driver. For driver installation and integration in the Wind River Workbench the complete folder has to be copied into the target directory of your VxWorks 6.x installation keeping the directory hierarchy.

The support to integrate the CAN driver in the VxWorks image will be available with the next start of the Workbench in the '*Kernel Configuration*' view below the new category '*esd gmbh Driver, Protocol Stacks and Software*'.



Note:

Some versions of the Wind River Workbench do not update their Components with each restart. To solve this issue it might help to remove the CDF cache file

`\target\config\comps\vxWorks\CxrCat.txt`

before restarting the Workbench.

To uninstall the driver package you just have to remove all files copied into the target architecture of your VxWorks installation.

4.1.2.3 VxWorks 7.x

The driver software for VxWorks 7.x with VxBus GEN2 support comes as a package for all supported host CPU architectures and CAN hardware. Once the RPM files are copied onto the directory <local_dir> on the host, the following commands are used to integrate the software to be used with the Workbench (the version numbers of the RPM files depends on the current distribution):

Linux:

```
<vw7_install_dir>/maintenance/wrInstaller/x86-linux2/wrInstaller -silent -nosplash -yum localinstall  
<local_dir>esd_CAN_PCI331-4.0.0.0-1.noarch.rpm -y
```

```
<vw7_install_dir>/maintenance/wrInstaller/x86-linux2/wrInstaller -silent -nosplash -yum localinstall  
<local_dir>esd_CAN_NTCAN-3.7.4.0-1.noarch.rpm -y
```

Windows:

```
<installation>\maintenance\wrInstaller\x86-win32\wrInstallerc.exe -silent -nosplash -yum localinstall  
<local_dir>\esd_CAN_PCI331-4.0.0.0-1.noarch.rpm -y
```

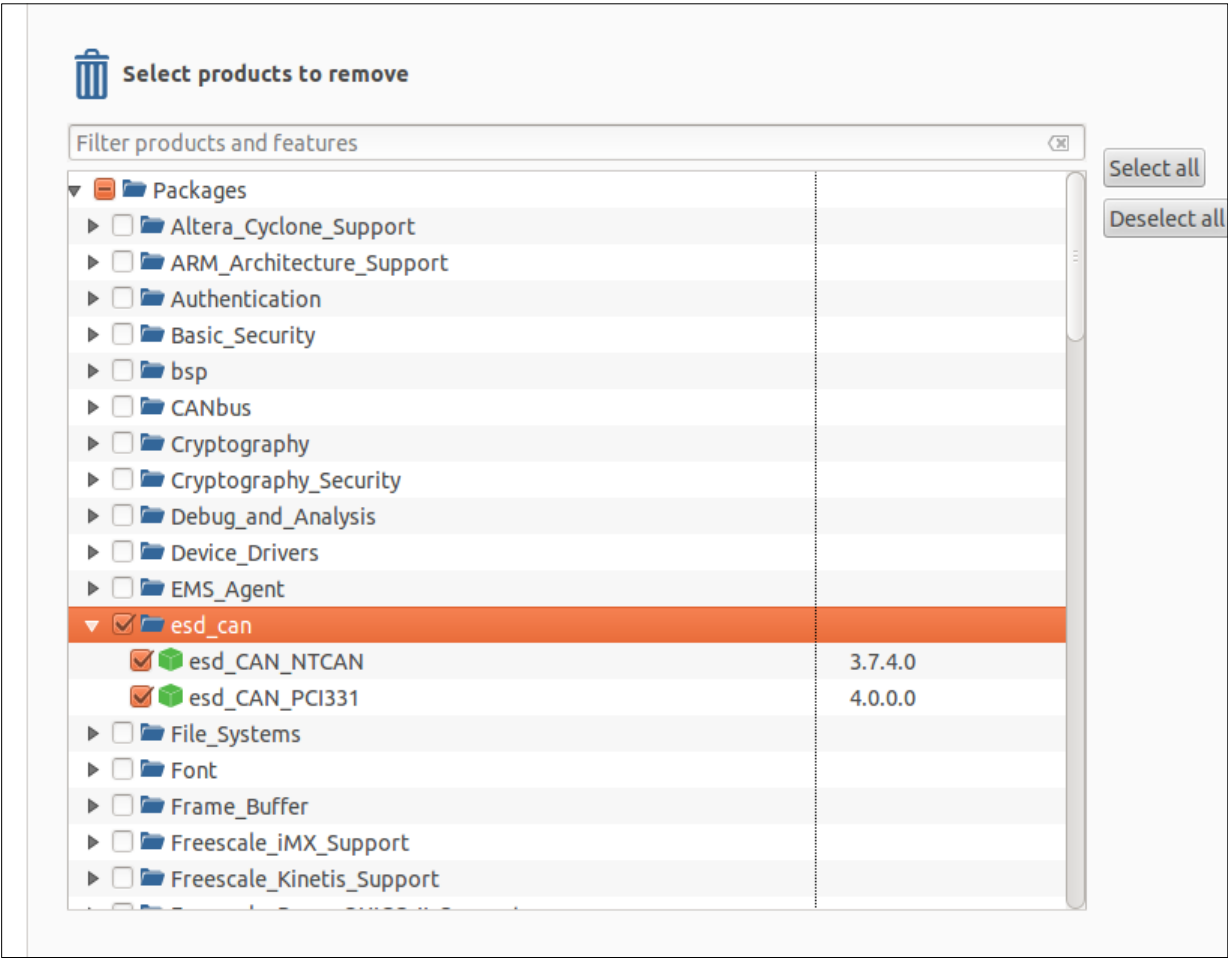
```
<installation>\maintenance\wrInstaller\x86-win32\wrInstallerc.exe -silent -nosplash -yum localinstall  
<local_dir>\esd_CAN_NTCAN-3.7.4.0-1.noarch.rpm -y
```

The following output should be seen (e.g. when installing the PCI331 driver):

```
--> Finished Dependency Resolution  
Dependencies Resolved  
#####  
Package Arch Version Repository Size  
#####  
Installing:  
esd_CAN_PCI331 noarch 4.0.0.0-1 local rpms 1009.9 KB  
Transaction Summary  
#####  
Install 1 Package(s) (+0 Dependent packages)  
Total download size: 1009.9 KB  
Is this ok [y/N]: y  
Downloading packages:  
esd_CAN_PCI331-4.0.0.0-1.rpm | 1009.9 KB < 1 min  
Running Transaction Check  
Running Transaction Test  
Transaction Test Succeeded  
Running Transaction  
Installing : esd_CAN_PCI331-4.0.0.0-1.noarch 1/1  
Configuring installation...  
Installed:  
esd_CAN_PCI331.noarch 4.0.0.0-1  
Complete!
```

When starting the Workbench the CAN driver, the NTCAN library and the cantest utility is available and integrated by default.

To **uninstall** the driver package, just start the wrInstaller, click on "Remove" and select the package to remove: esd_can.



4.1.3 Driver Configuration

This chapter describes the CAN device driver configuration for the various versions of VxWorks.

4.1.3.1 VxWorks 5.x

Driver for VxWorks 5.x are configured calling the driver start routine `<drvname>_install()` with a pointer to an initialized array of structures of the type `<drvsig>_CAN_INFO` (one array entry for each CAN interface). This can either be performed by modifying the code of the example start-up file `caninit.c` (recommended) or by calling the driver start routine `<drvname>_install()` from within your own application. In either case please look into the `caninit.c` to see configuration examples for several target architectures.

The varying name parts `<drvname>` and `<drvsig>` can be derived from Table 2 in chapter 1.4 where `<drvname>` is the device family specific driver name following driver naming convention I and `<drvsig>` is the signature.

The following section shows the members of a configuration structure, which is valid for most of the CAN Interface Families but NOT for all. To be confident about the structure's layout refer to the `caninit.c` source from the driver package.

```
struct <drvsig>_CAN_INFO
{
    unsigned long base;
    unsigned char net[4];
    unsigned char prio;
    unsigned char irq;
    unsigned long timestampFreq;
    unsigned char flags;
    unsigned char reserved;
};
```

The table below describes the members of the configuration structure:

Member	Description
base	<u>ISA and PC104 CAN Interfaces:</u> I/O base address of the module. <code>base</code> has to be set to the value configured on the hardware via jumpers or coding switches. If <code>base</code> is set to '0', the driver terminates the search for further CAN interfaces. <u>PCI, CPCI, PMC and PCIe CAN Interfaces:</u> For x86 architectures the PCI resources of the CAN hardware are configured usually by a kind of boot loader (usually the PC BIOS). In order to tell the device driver to use these addresses you have to set the parameter <code>base</code> to <code>0xFFFFFFFF</code>. You have to make sure that your BSP has got enough available entries in the MMU Memory Descriptor table (dummy entries) to register the PCI addresses of the CAN interface.

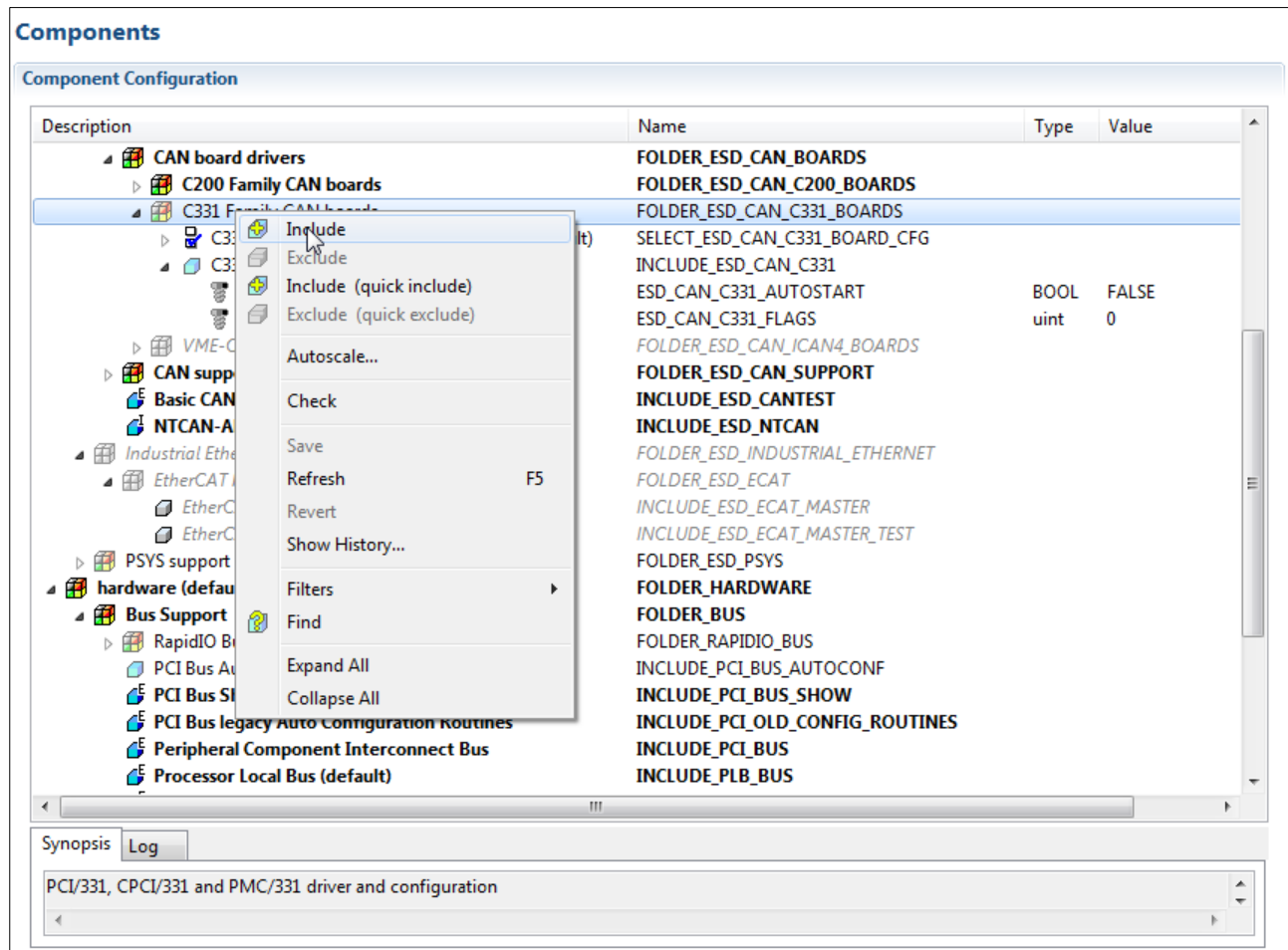
Member	Description
	For PPC architectures where the BSP or a boot loader performs the PCI bus enumeration and configuration you have to set the parameter <i>base</i> to 0xFFFFFFFF. For PPC architectures without any kind of boot loader you have to find an unused physical PCI address room (at least 3 MB, page aligned) which is used by the driver if configured in the parameter <i>base</i>. For debug level information about the correct setup of <i>base</i> have a look into the troubleshooting chapter 4.1.6.4.2.
<code>net[0]</code>	The parameter <code>net[0]</code> is the logical base net number that should be assigned to the first physical CAN port on the CAN board. If the hardware has more than one physical CAN port these ports will get consecutive logical net numbers starting with the base number. The values in <code>net[1]</code> to <code>net[3]</code> are ignored by the driver. The user has to make sure that all assigned logical net numbers are unique (especially if more than one esd CAN driver is active) as driver initialization otherwise will fail.
<code>prio</code>	Priority of back end task(s) which handle post processing of CAN messages. This back end task is responsible for the distribution of received CAN messages to all open handles. Therefore its priority must be better than the priority of all tasks that are doing CAN I/O via the driver to make it all work as expected.
<code>irq</code>	<u>ISA and PC104 CAN Interfaces:</u> Interrupt vector which should be configured and used by this CAN interface. You have to make sure that this interrupt is not used by any other hardware. <u>PCI, CPCI, PMC and PCIe CAN Interfaces:</u> For x86 architectures the PCI resources of the CAN hardware are configured usually by a kind of boot loader (usually the PC BIOS). In order to tell the device driver to use the configured resources you have to set the parameter <code>irq</code> to 0xFF. In this case the IRQ value configured in the configuration space of the PCI bridge is used and the default offset of 0x20 for a PIC system is added. For PPC architectures where the BSP or a boot loader performs the PCI bus enumeration and configuration you have to set the parameter <code>irq</code> to 0xFF. For PPC architectures without any kind of boot loader you have to find out the interrupt used by your target for the PCI slot in your hardware manual and set the parameter <code>irq</code> to this value regarding any BSP specific interrupt vector offsets. This manual setup may also be used to overcome issues that arise from wrong IRQ number translation, see chapter 4.1.6.5.2.
<code>timestampFreq</code>	The parameter <i>timestampFreq</i> defines the frequency of the timestamp in kHz for CAN drivers supporting software timestamps which are derived by a high resolution timer of the target CPU. If this parameter is set to 0 the driver probes this frequency, which causes a delay of driver startup where all interrupts and the scheduler are disabled. For x86 architectures the high resolution timestamp runs at processor frequency for most PPC targets it runs at 1/4 processor frequency. If a driver or the hardware does not support software timestamps the parameter should be set to 0. Note: This structure member does not exist for the VME-CAN4 driver. Instead the driver exports <i>can4_tickFreq</i> because of parameter structure size restrictions.

Member	Description						
flags	The parameter <i>flags</i> defines device flags supported by the driver: <table><tr><td>0x01</td><td>Suppress driver start-up banner.</td></tr><tr><td>0x02</td><td>Devices of the C331 and C200 family (Driver rev. 2.5.8 and later): Enable 'Delayed Read' support as specified in PCI specification v2.1</td></tr><tr><td>0x04</td><td>Devices of the C331 and C200 family (Driver rev. 2.7.0 and later): Use alternate connection method to attach the interrupt handler. x86 architecture: Use <i>intConnect()</i> instead of <i>pcilntConnect()</i>. PPC architecture: Use <i>pcilntConnect()</i> instead of <i>intConnect()</i>.</td></tr></table> Note: This structure member does not exist for the VME-CAN4 driver. Instead the driver exports <i>can4_flags</i> because of parameter structure size restrictions.	0x01	Suppress driver start-up banner.	0x02	Devices of the C331 and C200 family (Driver rev. 2.5.8 and later): Enable 'Delayed Read' support as specified in PCI specification v2.1	0x04	Devices of the C331 and C200 family (Driver rev. 2.7.0 and later): Use alternate connection method to attach the interrupt handler. x86 architecture: Use <i>intConnect()</i> instead of <i>pcilntConnect()</i> . PPC architecture: Use <i>pcilntConnect()</i> instead of <i>intConnect()</i> .
0x01	Suppress driver start-up banner.						
0x02	Devices of the C331 and C200 family (Driver rev. 2.5.8 and later): Enable 'Delayed Read' support as specified in PCI specification v2.1						
0x04	Devices of the C331 and C200 family (Driver rev. 2.7.0 and later): Use alternate connection method to attach the interrupt handler. x86 architecture: Use <i>intConnect()</i> instead of <i>pcilntConnect()</i> . PPC architecture: Use <i>pcilntConnect()</i> instead of <i>intConnect()</i> .						
reserved	Reserved for future use. Set to 0.						

4.1.3.2 VxWorks 6.x (Non-VxBus)

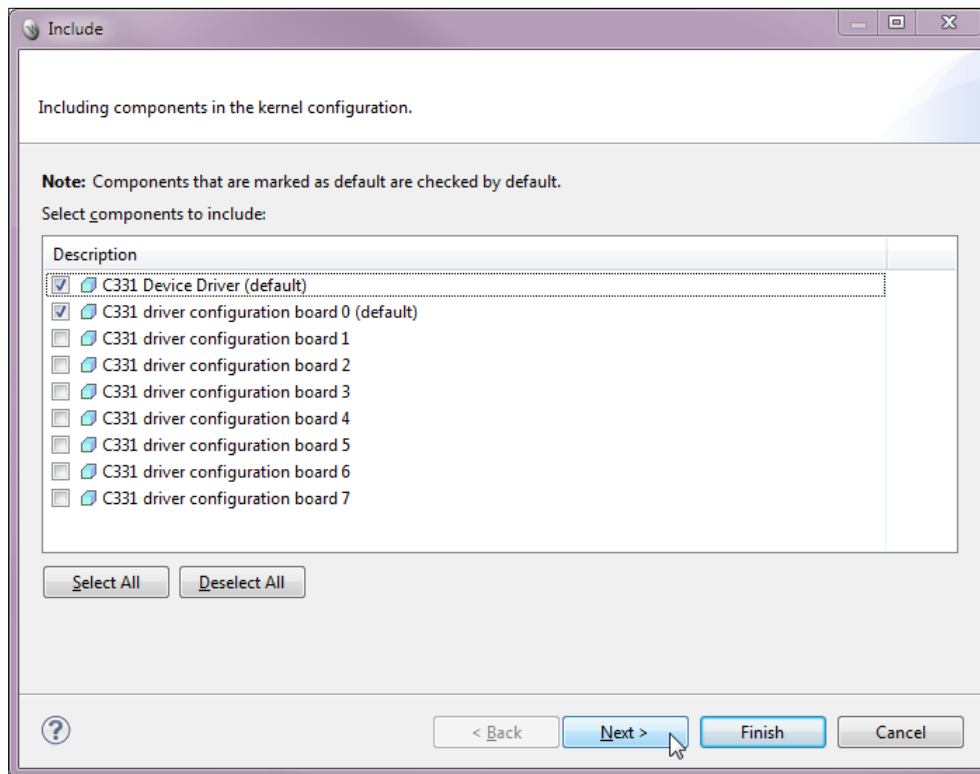
The integration of the VxBus CAN device driver into your VxWorks image and the configuration of the driver parameter is done in the Wind River Workbench via the *Kernel Configuration*. After the driver installation described in the previous chapter you will have a new node in your Workbench VxWorks kernel parameter tree with the name *esd gmbh Driver, Protocol Stacks and Software*.

To integrate the device driver for your CAN Interface include the device driver for this CAN Interface Family (see chapter 1.4) by opening the context menu and choose the option *Include* as shown in the picture below for the C331 Family device driver.

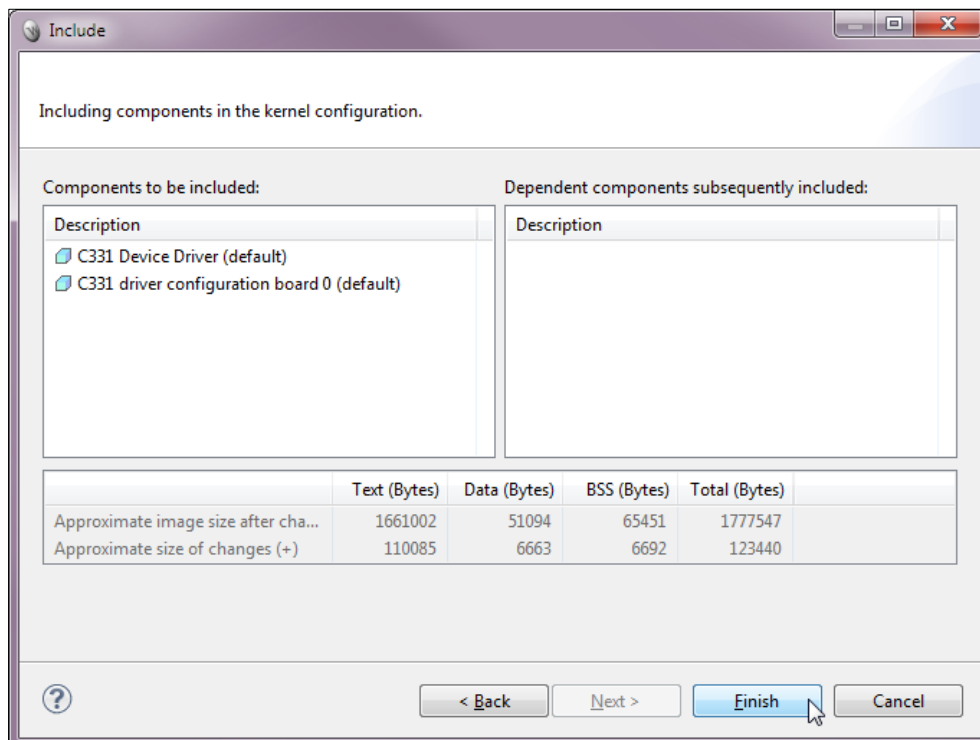


In the next step you will have the opportunity to configure how many boards you want to use.

In the following dialogue box, you have to define the number of CAN Interfaces of this CAN Interface Family you want to support before you click *Next*.



Complete the integration of the CAN device driver by clicking *Finish* in the next dialogue.



Continue configuring some additional options.

Description	Name	Type	Value
esd gmbh Driver, Protocol Stacks and Software	FOLDER_ESD_ROOT		
CAN Protocol Components	FOLDER_ESD_CAN_PROTOCOLS		
CAN fieldbus	FOLDER_ESD_CAN_NETWORK		
CAN board VxBus drivers	FOLDER_ESD_CAN_BOARDS_VXBUS		
CAN board drivers	FOLDER_ESD_CAN_BOARDS		
C200 Family CAN boards	FOLDER_ESD_CAN_C200_BOARDS		
C200 Family CAN Board Configurations (default)	SELECT_ESD_CAN_C200_BOARD_CFG		
C200 driver configuration board 0 (default)	INCLUDE_ESD_CAN_C200_BRD_0		
Backend priority of 1st board	ESD_CAN_C200_BACKEND_0	uint	50
Base net of 1st board	ESD_CAN_C200_BASENET_0	uint	0
PCI IRQ vector of 1st board	ESD_CAN_C200_IRQ_VECTOR_0	uint	0xFF
PCI memory base address of 1st board	ESD_CAN_C200_BASEMEM_0	uint	0xFFFFFFFF
C200 driver configuration board 1	INCLUDE_ESD_CAN_C200_BRD_1		
C200 driver configuration board 2	INCLUDE_ESD_CAN_C200_BRD_2		
C200 driver configuration board 3	INCLUDE_ESD_CAN_C200_BRD_3		
C200 driver configuration board 4	INCLUDE_ESD_CAN_C200_BRD_4		
C200 driver configuration board 5	INCLUDE_ESD_CAN_C200_BRD_5		
C200 driver configuration board 6	INCLUDE_ESD_CAN_C200_BRD_6		
C200 driver configuration board 7	INCLUDE_ESD_CAN_C200_BRD_7		
C200 Device Driver (default)	INCLUDE_ESD_CAN_C200		
Autostart behaviour of driver	ESD_CAN_C200_AUTOSTART	BOOL	TRUE
Driver flags	ESD_CAN_C200_FLAGS	uint	1
Target timestamp counter frequency in kHz	ESD_CAN_C200_TIMESTAMP_FREQUENCY	uint	0
C331 Family CAN boards	FOLDER_ESD_CAN_C331_BOARDS		
VME-CAN4 boards	FOLDER_ESD_CAN_ICAN4_BOARDS		
CAN support components (private)	FOLDER_ESD_CAN_SUPPORT		
BSP PCI PnP quirk for PCI CAN boards	FOLDER_ESD_CAN_BSP_PNP_QUIRK		
Adapt IRQ and/or PCI BARx addresses to BSP	SELECT_PNP_QUIRK_TYPE		
PNP BSP quirk (type 1)	INCLUDE_ESD_CAN_BSP_PNP_QUIRK		
PCI IRQ offset	ESD_CAN_PCI_IRQ_OFFSET	uint	0
PCI address offset	ESD_CAN_PCI_ADR_OFFSET	uint	0x0
PNP BSP quirk (type 2) (default)	INCLUDE_ESD_CAN_BSP_PNP_QUIRK_2		
PCI IRQ to Vector Mapping	ESD_CAN_PCI_IRQ_ROUTINE	string	irq = INT_NUM_GET(irq);
PCI address offset	ESD_CAN_PCI_ADR_OFFSET	uint	0x0
esd PCI CAN board initialization	INCLUDE_ESD_CAN_PCI_CONFIG		
Basic CAN test application	INCLUDE_ESD_CANTEST		
NTCAN-API support	INCLUDE_ESD_NTCAN		

Synopsis Log

0x01 = No banner, 0x02 = Enable PCI delayed read 0x04 = Non default IRQ connector

Defined at:

[E:\vworks-6.9\target\config\comps\vxWorks\52esd_can_pci.cdf](#)

Optionally you can adapt the *Backend priority of driver* option (2) which defines the priority of the task for CAN messages post processing to your requirements. Refer to the description of the parameter *prio* in chapter 4.1.3.1 for more details.

Adapt the logical base net of the CAN interface with the option *Base net of board x* (1) to your requirements. If you use just the device driver for one CAN Interface Family leave the default values.

Usually you can leave the *PCI IRQ Vector* (3) set to the default of 0xFF and the *PCI memory base address* set to the default of 0xFFFFFFFF (4). Refer to the description of the parameter *irq* and *base* in chapter 4.1.3.1 for more details.

Set the *Autostart behaviour of driver* option (5) to `FALSE` for a first test and start the driver manually as described in the next chapter. If everything works you can change this parameter to `TRUE` to start the driver with your VxWorks.

Reset bit 0 of the *Driver Flags* option (6) for a first test to start the driver with a start-up banner. If everything works you can set this bit to suppress this banner. Refer to the *Synopsis* box for further driver specific flags.

CAN Interfaces with software timestamp support have the additional option *Target timestamp*

counter frequency (7) which defines the frequency of the counter used for software timestamping in kHz. If this parameter is set to 0 the driver probes this frequency (which causes a delay of driver initialization where all interrupts and the scheduler are disabled) otherwise it uses the configured value. For **x86** architectures the high resolution timestamp used runs at processor frequency for most **PPC** targets it runs at 1/4 of the processor frequency.

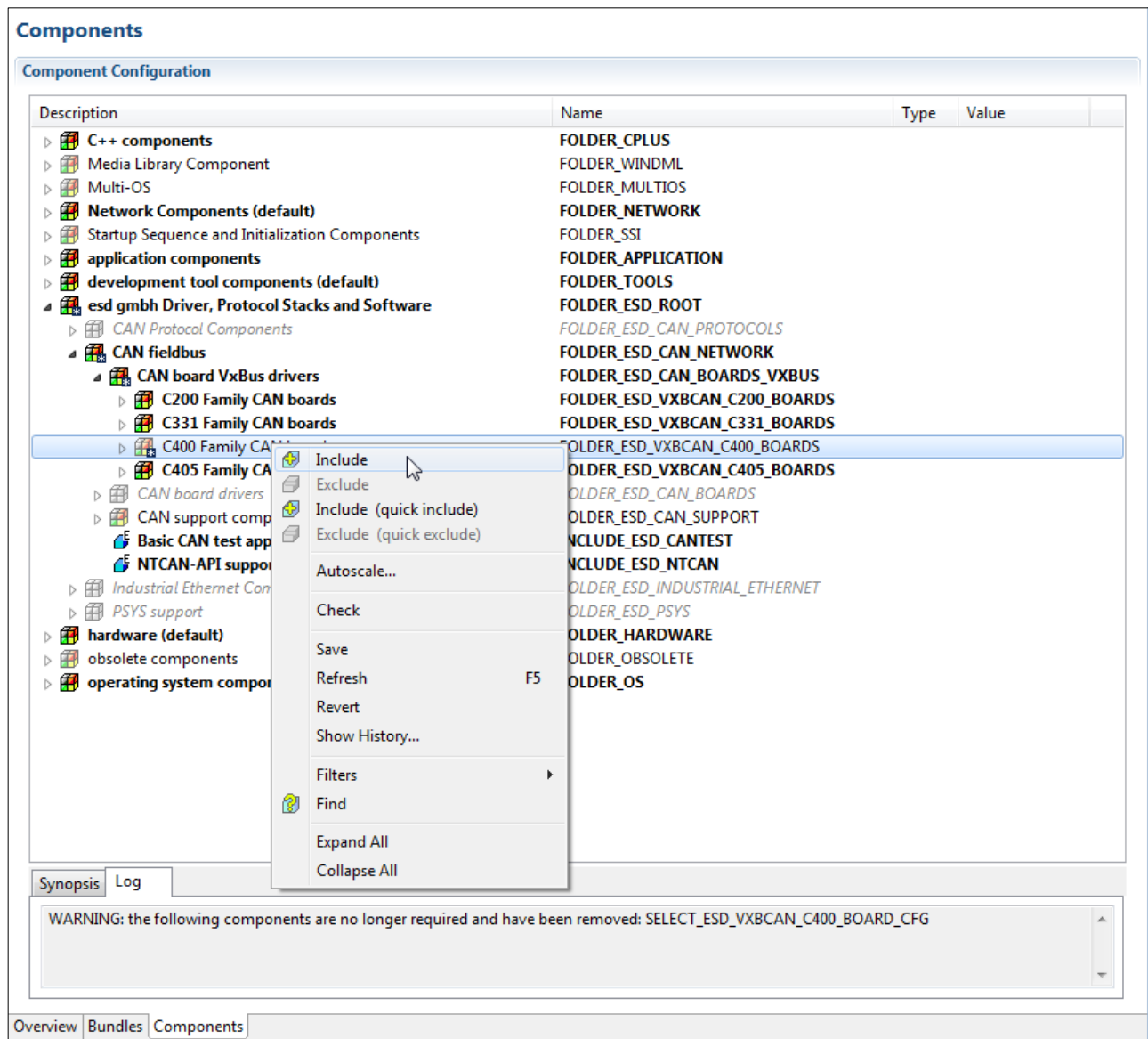
Include *NTCAN-API support* component (12) as this is the common API /1/ for all device drivers you use by your application. This component is automatically included if you include the *Basic CAN Test Application* component (13) which is the *canTest* application described in chapter 4.1.5.3.

Now you can rebuild your VxWorks image.

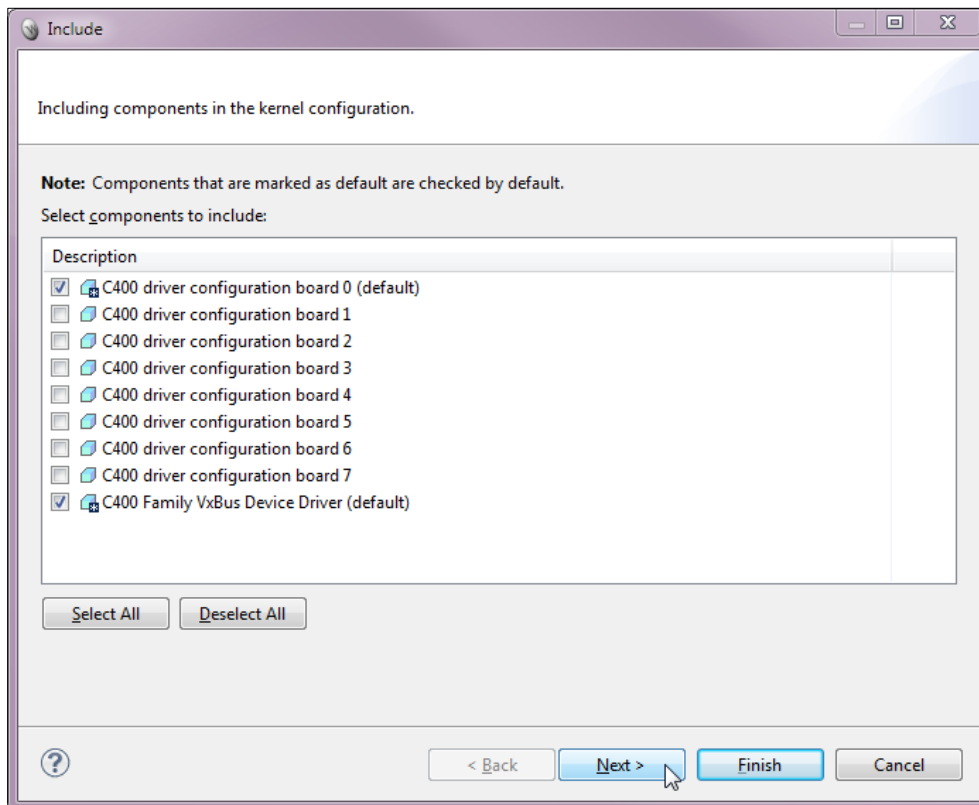
4.1.3.3 VxWorks 6.x (VxBus)

The integration of the VxBus CAN device driver into your VxWorks image and the configuration of the driver parameter is done in the Wind River Workbench via the *Kernel Configuration*. After the driver installation described in the previous chapter you will have a new node in your Workbench VxWorks kernel parameter tree with the name *esd gmbh Driver, Protocol Stacks and Software*.

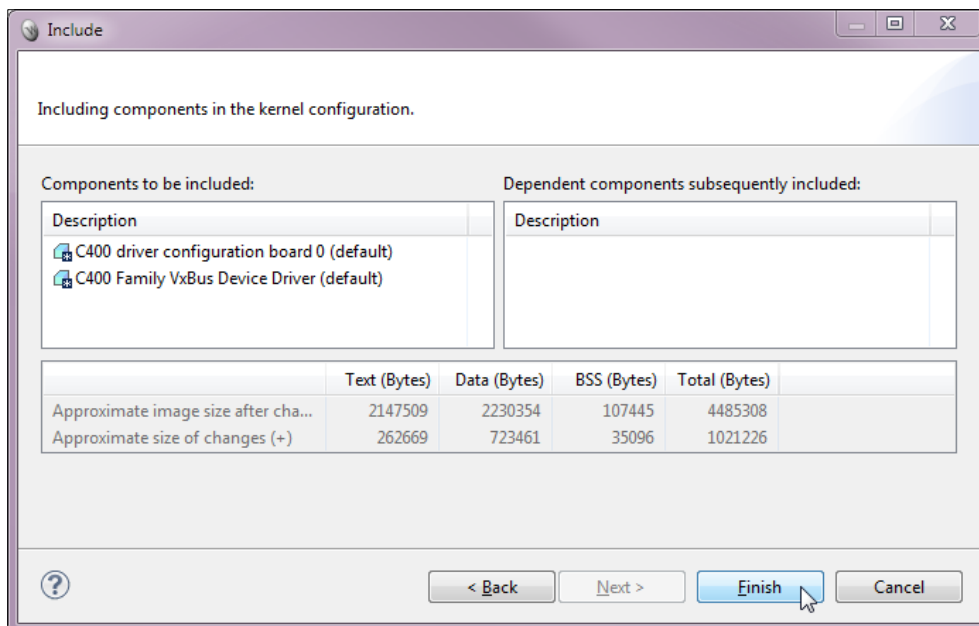
To integrate the device driver for your CAN Interface include the device driver for this CAN Interface Family (see chapter 1.4) by opening the context menu and choose the option *Include* as shown in the picture below for the C400 Family VxBus device driver.



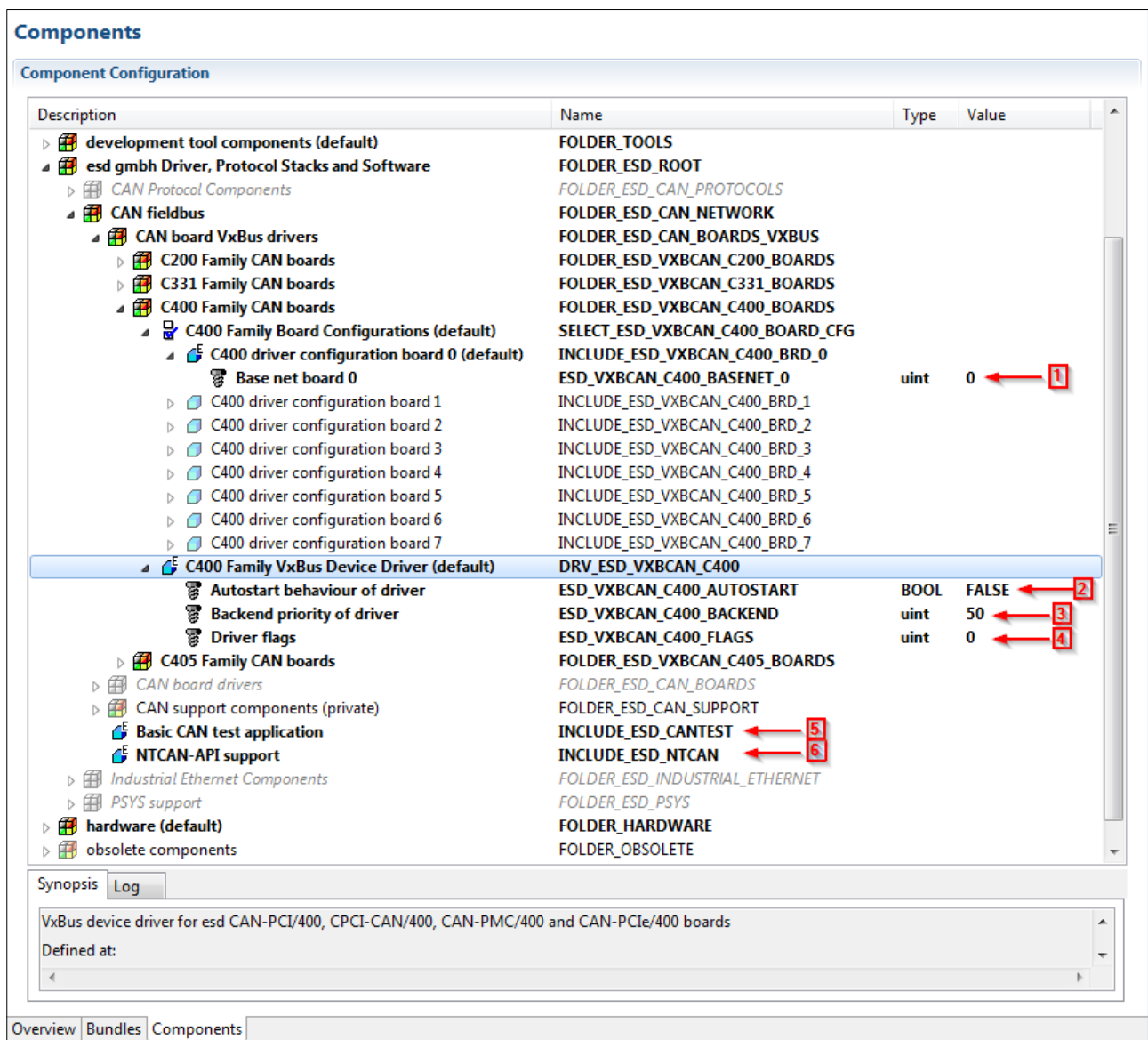
In the following dialogue box you have to define the number of CAN Interfaces of this CAN Interface Family you want to support before you click *Next*.



Complete the integration of the CAN device driver by clicking *Finish* in the next dialogue.



Continue configuring some additional options.



Adapt the logical base net of the CAN interface with the option *Base net of board x* (1) to your requirements. If you use just the device driver for one CAN Interface Family leave the default values.

Set the *Autostart behaviour of driver* option (2) to `FALSE` for a first test and start the driver manually as described in the next chapter. If everything works you can change this parameter to `TRUE` to start the driver with your VxWorks.

Optionally you can adapt the *Backend priority of driver* option (3) which defines the priority of the task for CAN messages post processing to your requirements. Refer to the description of the parameter *prio* in chapter 4.1.3.1 for more details.

Reset bit 0 of the *Driver Flags* option (4) for a first test to start the driver with a start-up banner. If everything works you can set this bit to suppress this banner. Refer to the *Synopsis* box for further driver specific flags.

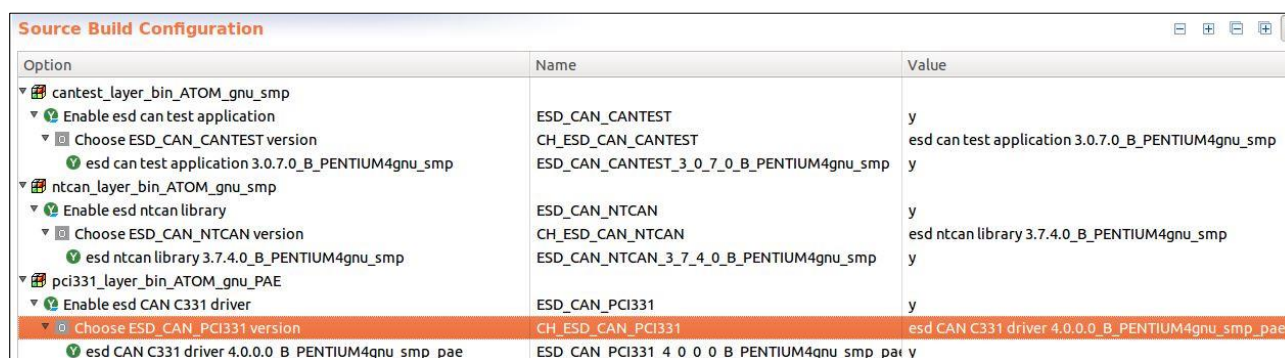
CAN Interfaces with software timestamp support have the additional option *Target timestamp counter frequency* (not in the picture above) which defines the frequency of the counter used for software timestamping in kHz. If this parameter is set to 0 the driver probes this frequency (which causes a delay of driver initialization where all interrupts and the scheduler are disabled) otherwise it uses the configured value. For **x86** architectures the high resolution timestamp used runs at processor frequency for most **PPC** targets it runs at 1/4 of the processor frequency.

Include *NTCAN-API support* component (6) as this is the common API /1/ for all device drivers you use by your application. This component is automatically included if you include the *Basic CAN Test Application* component (5) which is the *canTest* application described in chapter 4.1.5.3.

Now you can rebuild your VxWorks image.

4.1.3.4 VxWorks 7.x (VxBus GEN2)

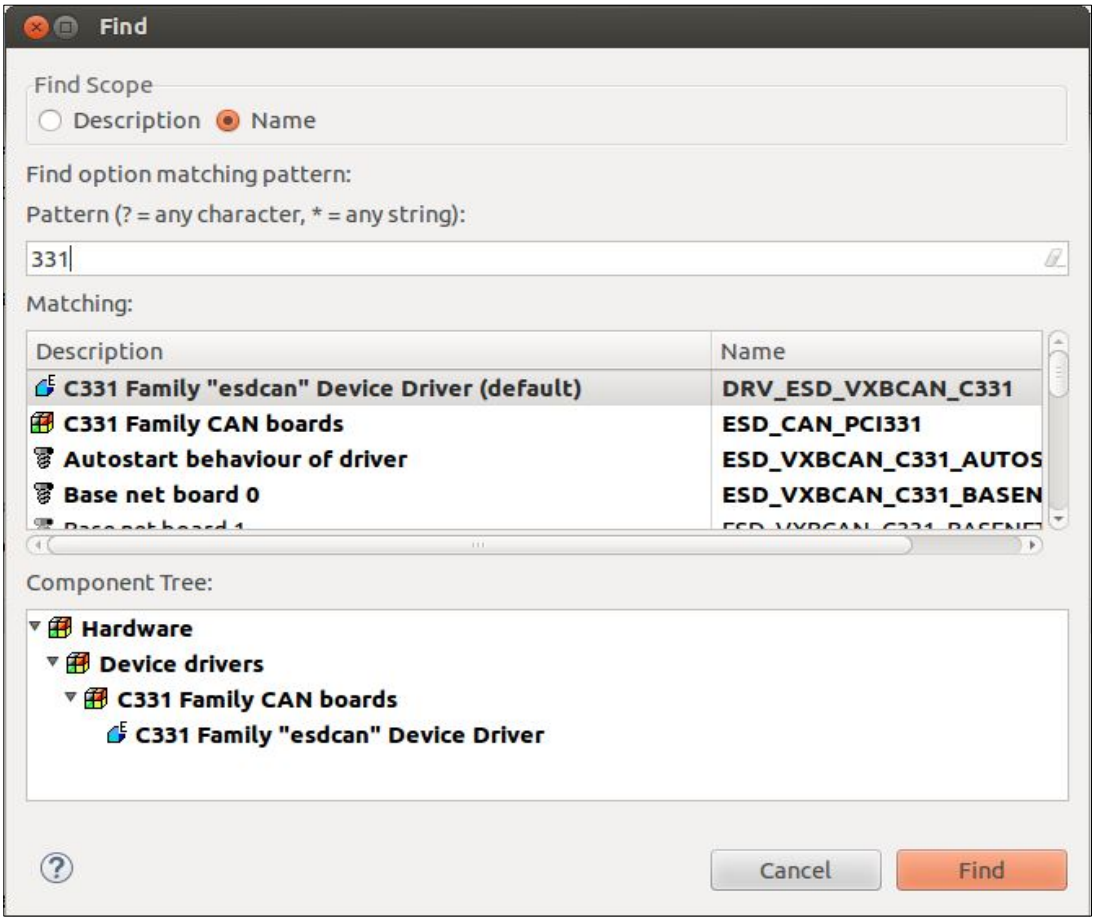
The integration of the VxBus CAN device driver and related software into your VxWorks image and the configuration of the driver parameter is done in the Wind River Workbench. After the driver installation described in the previous chapter you will have new nodes showing up when building a Source Build. By default all nodes are enabled which a reasonable starting point. Eventually the *cantest* utility can be removed, unless required.



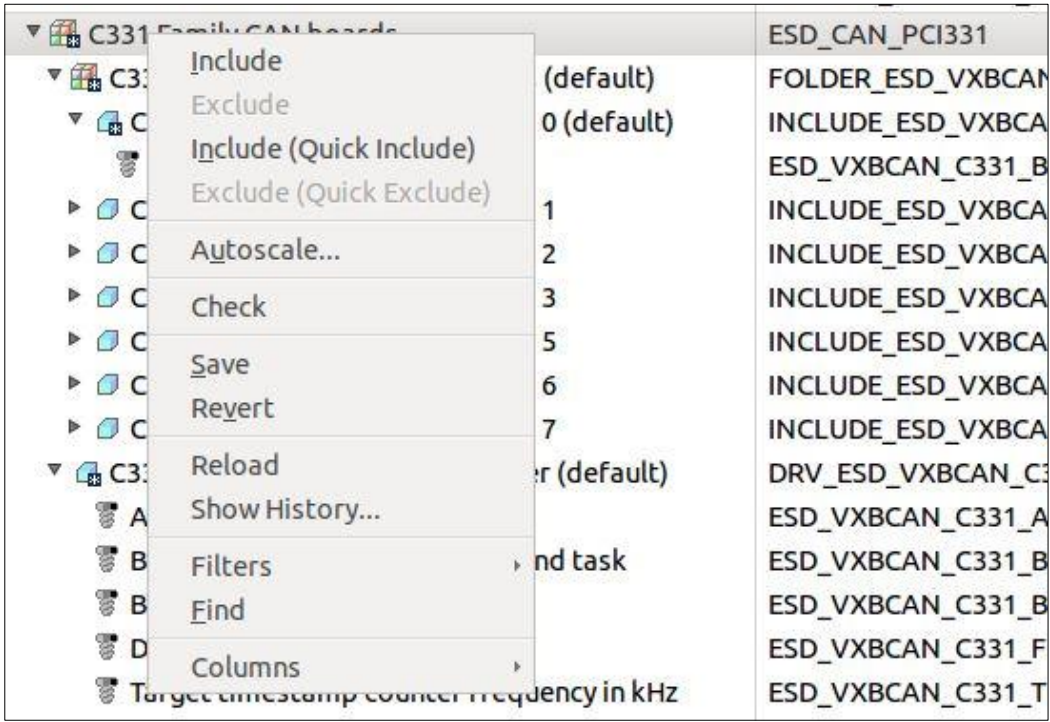
Option	Name	Value
▼ cantest_layer_bin_ATOM_gnu_smp		
▼ Enable esd can test application	ESD_CAN_CANTEST	y
▼ Choose ESD_CAN_CANTEST version	CH_ESD_CAN_CANTEST	esd can test application 3.0.7.0_B_PENTIUM4gnu_smp
▼ esd can test application 3.0.7.0_B_PENTIUM4gnu_smp	ESD_CAN_CANTEST_3_0_7_0_B_PENTIUM4gnu_smp	y
▼ ntcant_layer_bin_ATOM_gnu_smp		
▼ Enable esd ntcant library	ESD_CAN_NTCAN	y
▼ Choose ESD_CAN_NTCAN version	CH_ESD_CAN_NTCAN	esd ntcant library 3.7.4.0_B_PENTIUM4gnu_smp
▼ esd ntcant library 3.7.4.0_B_PENTIUM4gnu_smp	ESD_CAN_NTCAN_3_7_4_0_B_PENTIUM4gnu_smp	y
▼ pci331_layer_bin_ATOM_gnu_PAE		
▼ Enable esd CAN C331 driver	ESD_CAN_PCI331	y
▼ Choose ESD_CAN_PCI331 version	CH_ESD_CAN_PCI331	esd CAN C331 driver 4.0.0.0_B_PENTIUM4gnu_smp_pae
▼ esd CAN C331 driver 4.0.0.0_B_PENTIUM4gnu_smp_pae	ESD_CAN_PCI331_4_0_0_0_B_PENTIUM4gnu_smp_pae	y

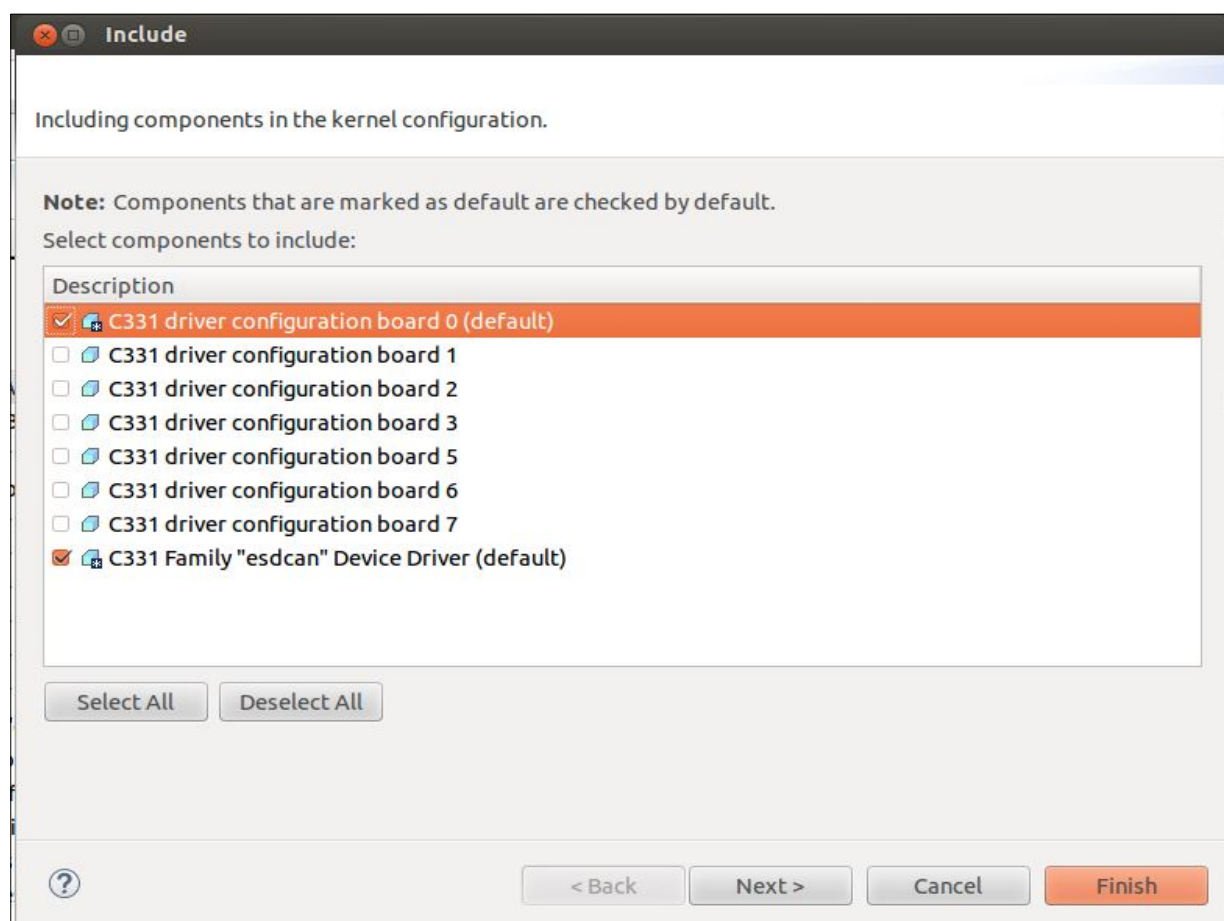
There are a number of combinations the supported by the driver with the appropriate combination (e.g. NEHALEM, SMP, LP64) selected automatically. Don't be confused by the name "...bin_ATOM_..." as the layer base name which is generated during the driver build stage. Only the chosen version (...PENTIUM4gnu_smp in the sample above) is of any meaning at this point!

Based on that source build, a VxWorks Image can be build. When opening the *Kernel Configuration*, you can search for the can driver, library or tools:



By default, the driver and software is **not** included in the kernel image. Choose "include" on the driver, library and the test application to be able to test the installation.





Once all components have been included the configuration should look like this:

Device drivers	FOLDER_DRIVERS		
ADC	FOLDER_DRIVERS_ADC		
Busses	FOLDER_DRIVERS_BUS		
C331 Family CAN boards	ESD_CAN_PCI331		
C331 Family Board Configurations (default)	FOLDER_ESD_VXBCAN_C331_BOARD_0		
C331 driver configuration board 0 (default)	INCLUDE_ESD_VXBCAN_C331_BRD_0		
Base net board 0	ESD_VXBCAN_C331_BASENET_0	uint	0
C331 driver configuration board 1	INCLUDE_ESD_VXBCAN_C331_BRD_1		
C331 driver configuration board 2	INCLUDE_ESD_VXBCAN_C331_BRD_2		
C331 driver configuration board 3	INCLUDE_ESD_VXBCAN_C331_BRD_3		
C331 driver configuration board 5	INCLUDE_ESD_VXBCAN_C331_BRD_5		
C331 driver configuration board 6	INCLUDE_ESD_VXBCAN_C331_BRD_6		
C331 driver configuration board 7	INCLUDE_ESD_VXBCAN_C331_BRD_7		
C331 Family "esdcan" Device Driver (default)	DRV_ESD_VXBCAN_C331		
Autostart behaviour of driver	ESD_VXBCAN_C331_AUTOSTART	BOOL	FALSE
Backend affinity of driver backend task	ESD_VXBCAN_C331_BKND_AFFINITY	uint	0
Backend priority of driver	ESD_VXBCAN_C331_BKND_PRIO	uint	50
Driver flags	ESD_VXBCAN_C331_FLAGS	uint	1
Target timestamp counter frequency in kHz	ESD_VXBCAN_C331_TIMESTAMP_FREQ	uint	0

You can modify the parameter as described for VxWorks 6.x (see chapter 4.1.3.3).

4.1.4 Driver Start

This chapter describes the CAN device driver start for the various versions of VxWorks after the CAN driver is integrated into the VxWorks image and configured.

4.1.4.1 VxWorks 5.x

If you need a specific configuration we recommend to use `caninit.c` as a template, change the source code according to your needs, compile and load it after the driver binary.

If you use a modified `caninit.c` to start your driver you should call **`xxxStart()`** otherwise you have to call the **`xxx_install()`** routine which is exported by the driver with an array of initialized `XXX_CAN_INFO` structures. In both cases 'xxx' is the CAN Family device driver specific prefix as described in chapter 1.4. In case of a CAN device driver which comes as part of a VxWorks BSP

for an **esd** embedded board, the driver is usually started just with **`canStart()`**.

If the driver start-up banner is enabled you will see some informations to the hardware/software environment similar to the example below:

```
C200: "CAN_PCI266" with 2 Nets identified
C200: Hardware-Version = 1.0.00
C200: Firmware-Version = 0.0.00
C200: Driver-Version   = 2.5.08
C200: Net 0 successfully created on card 0
C200: Net 1 successfully created on card 0
```

In the example above it is the start of a CAN-PCI/266 CAN Interface which belongs to the C200 Family. Refer to chapter 1.4 for the prefix overview of other device drivers.

4.1.4.2 VxWorks 6.x (Non-VxBus)

For a first test you should configure the *Autostart behaviour of driver* option in the Wind River Workbench to `FALSE` so you can start the driver manually on the target shell with **`xxxStart()`** where 'xxx' is the CAN Family device driver specific prefix as described in chapter 1.4. In case of a CAN device driver which comes as part of a VxWorks BSP for an **esd** embedded board, the driver is usually started just with **`canStart()`**.

If the driver start-up banner is enabled via the *Driver Flags* option in the Wind River Workbench you will see some informations to the hardware/software environment similar to the example below:

```
C331: Using IRQ/Vector* (0x0b/0x158) shared for card 0
C331: "CAN_PCI331" with 2 Nets identified
C331: Hardware-Version = 1.1.00
C331: Firmware-Version = 0.c.1e
C331: Driver-Version   = 2.7.00
C331: Timestamp frequency is 1476535544 Hz (Software)
C331: Net 0 successfully created on card 0
C331: Net 1 successfully created on card 0
C331: CAN 2.0A firmware active
C331: PCI delayed read: Disabled (CNTRL: 0x18780fd7)
```

In the example above it is the start of a CAN-PCI/331 CAN Interface which belongs to the C331 Family. Refer to chapter 1.4 for the prefix overview of other device drivers.

If there are no problems you can configure the *Autostart behaviour of driver* option in the Wind River Workbench to `TRUE` so the driver is started automatically every time the VxWorks image is booted.

4.1.4.3 VxWorks 6.x/7.x (VxBus)

For a first test you should configure the *Autostart behaviour of driver* option in the Wind River Workbench to `FALSE` so you can start the driver manually on the target shell with **xxxStart()** where 'xxx' is the CAN Family device driver specific prefix as described in chapter 1.4. In case of a CAN device driver which comes as part of a VxWorks BSP for an **esd** embedded board, the driver is usually started just with **canStart()**.

If the driver start-up banner is enabled via the *Driver Flags* option in the Wind River Workbench you will see some informations to the hardware/software environment similar to the example below:

```
C405: "CAN_PCI405" with 4 Nets identified
C405: Hardware-Version = 1.3.01
C405: Firmware-Version = 3.8.16
C405: Driver-Version   = 3.9.00 (SMP / VxBus Rev. 4)
C405: Net 0 successfully created on card 0
C405: Net 1 successfully created on card 0
C405: Net 2 successfully created on card 0
C405: Net 3 successfully created on card 0
```

In the example above it is the start of a CAN-PCI/405 CAN Interface which belongs to the C405 Family. Refer to chapter 1.4 for the prefix overview of other device drivers.

If there are no problems, you can configure the *Autostart behaviour of driver* option in the Wind River Workbench to `TRUE` so the driver is started automatically every time the VxWorks image is booted.

4.1.5 Miscellaneous

This chapter covers several topics which should preclude problems building the VxWorks image and using the NTCAN architecture on VxWorks.

4.1.5.1 Unresolved Symbols Building the VxWorks Image

Using the NTCAN library the linker might fail with unresolved symbols like `__udivdi3` or `__umodi3`. The reason for this is that to support 64-bit arithmetic (for the NTCAN timestamps) on a 32-bit CPU the GCC compiler requires some helper functions which are part of the GNU GCC run-time libraries which is not included in any case. To prevent error messages about unresolved symbols you must include the GNU GCC run-time libraries in your VxWorks image by defining `INCLUDE_GNU_INTRINSICS`. Please refer to the *VxWorks Kernel Programmer's Guide* for more details.

4.1.5.2 Number of Available NTCAN Handles

A CAN driver supports up to 1024 individual CAN handles but the maximum number is also limited by the maximum number of open handles of the VxWorks system. The related VxWorks configuration parameter is `NUM_FILES` with a default value of 50. Changing the value of `NUM_FILES` can have some other undesirable effects on the system if certain limits are not observed. In VxWorks 5.4.x setting `NUM_FILES` to a value greater than 256 could cause problems for code that use the ***select()*** function. The `FD_SETSIZE` was limited to 256 in this version. Starting with VxWorks 5.5 this limitation was removed but you might still adapt the value of `FD_SETSIZE` if opening a NTCAN handle failed with a resource error.

4.1.5.3 Test Program 'canTest'

The driver package comes with the test program ***canTest*** which can be either optionally configured into your VxWorks image via the Workbench (VxWorks 6.x) or can be loaded with the VxWorks target shell (VxWorks 5.x). With the help of this program, you can do basic functionality checks of the CAN interface. The program and its parameters are described in more detail in */1/*.



Note:

As VxWorks allows less parameters in the command line as required by *canTest*, the parameters must be provided separated by spaces as a single string in quote signs.

4.1.5.4 Unexpected Behaviour of Software Timestamps

With some VxWorks versions and BSPs the driver may show unexpected behaviour with software timestamps. In this case the measured timestamp frequency as shown in the driver's startup banner may be negative or far beyond the expected value. Or the time differences calculated between to received CAN frames have surprising values.

The reason for that behaviour can be for instance on the **x86** architecture that the high-resolution time stamp counter (TSC) is reset every timer tick. This is done in some BSPs because WindView needs that behaviour. For further information refer to this website:

<http://borkhuis.home.xs4all.nl/vxworks/pc-bsp.html#pc-bsp.5>

4.1.5.5 Correct Interpretation of Error Codes Returned by the Driver

The interpretation of error codes returned by the driver is guided by a multi level approach. The first resource where you should search for a specific error code is the header file `ntcan.h` that is delivered in the driver package. This file contains definitions of macros for the NTCAN error codes for example:

```
#define NTCAN_SUCCESS 0
#define NTCAN_RX_TIMEOUT 0x00001001
```

Please observe that the upper word of all defined errors is zero. In `ntcan.h` you can search for a certain error code value and map it to the macro name. The meaning of these macro names are described in **chapter 'Return Codes' in /1/**.

If you did not find the macro name for the specific error code you want to look up then consider the error numbers the NTCAN API borrows from the system via the `errno.h` file, for example:

```
#define NTCAN_INVALID_PARAMETER EINVAL
#define NTCAN_INVALID_HANDLE EBADF
```

Therefore, on the second level you have to lookup that error code in the VxWorks `errno.h` that is installed in the target tree of your VxWorks installation at `target/h/errno.h`. This way a return code of `0x00000016` maps to `EINVAL` and therefore in turn to `NTCAN_INVALID_PARAMETER`.

Another kind of error code could occur with the upper word of the unsigned error code not being zero. To decode the VxWorks error code of `0x000d0003` you should refer additionally to the file `vwModNum.h` also being in the `target/h` directory.

```
From <vwModNum.h>:
#define M_iosLib (13 << 16)
```

```
From <errno.h>:
#define ESRCH 3 /* No such process */
```

This means that the VxWorks `iosLib` wants to tell us that there is "no such process". Our driver code seems not to be reached since already the `iosLib` returns an error code. From this we may deduce that something is wrong with the handle itself.

The most common error code returned by the `<drvname>_install()` routine is 19 i. e. `ENODEV`. That means that the install routine did not find all boards that were requested to initialize by the `<drvsig>_CAN_INFO` structure array.

The described method applies if you need to translate NTCAN error codes by manually. Since revision 3.3.0 the NTCAN library provides the function `canFormatError()` that should be used by an application to translate error codes into error strings.

4.1.5.6 Support of the CAN Extended Frame Format

All CAN Interfaces support the Extended Frame Format (29 Bit CAN-IDs) according to the CAN 2.0B specification [2]. The active CAN Interfaces which belong to the C331-, C360-, C331I- and ICAN4-Family (see chapter 1.4) are usually configured in a 29-Bit passive mode which means that CAN messages in the *Extended Frame Format* are tolerated but can not be received or transmitted by an application.

Starting with firmware revision 0.C.09 of these active CAN Interfaces the firmware can be switched between this 11 bit active/29 bit passive factory mode and the 11/29 bit active mode.

The firmware operation mode is indicated in the start-up banner and can be queried at runtime on the target shell with

```
xxx_switch(0, net)
```

where '**xxx**' is the driver specific prefix defined in chapter 1.4 and *net* is a logical net number assigned to one of the CAN Interface's ports.

You can toggle between the two modes of operation while the driver is running with the same command changing the first parameter.

```
xxx_switch(1, net)
```

**Note:**

After the operation mode is toggled you have to reboot your target to make the new configuration active.

4.1.6 Troubleshooting Hints

The following paragraphs will detail some common problems that may get in the way of a successful driver start. Also here you should find some advice to diagnose the problem and information about the internal working of the driver as far as installation is concerned.

Unfortunately the hardware abstraction under VxWorks is in a state of flow where the way the things are done is very dependent on the VxWorks version, the CPU-Architecture and even the BSP of your board. This is the reason why it is advisable to provide the exact platform you will be working with (VxWorks version, CPU-Architecture, BSP type and version, SMP requirements) when you order a CAN driver.

The first thing you must be sure of is to use the right driver binary for your architecture. If this is guaranteed the reason for the driver not working should only be a configuration problem.

If you have a driver that is installed in the VxWorks target tree and configured via the Workbench it should show up and be selectable in the components tab of the kernel configuration pane. If that is not the case you may use an CPU architecture for your kernel that the driver doesn't support. Look for `libesdCAN.a` in the target tree of your VxWorks installation. There must be one for your target architecture or the Workbench won't show the driver as selectable. Review chapter 4.1.2.2 for installation information.

The CAN driver needs to be able to access the board's registers and / or memory spaces. Also the driver tries to connect an interrupt service routine to handle the interrupts generated by the board. The chapter 4.1.6.4 deals with problems related to board access and the chapter 4.1.6.5 deals with problems with the interrupt handling.

In the case of the driver not starting successfully (`<drvname>Start()` calling `<drvname>_install()` failed) or not working properly there may be either issues with the access to the board or with the interrupt connection. Depending on the kind of CAN board the behaviour is a little bit different.

For passive boards (no local CPU) like the C200 family boards the driver start may seem to be successful. But the IRQ is not exercised while starting the driver. Therefore an error or crash may appear later when for example `canWrite()` is called the first time. This is the reason you should test a successful execution of `canWrite()` or `canRead()` for passive boards immediately with `canTest` from the target shell.

As a rule of thumb you can deduce that it is an address translation / board access issue if the driver start fails immediately. If the problems occur later (i. e. single CAN transfer successful) or crash it is more probably an IRQ connection issue.

For active boards (with local CPU or FPGA) like the C331 family boards the IRQ is exercised during installation. At first the driver needs access to the registers or dual-ported RAM areas of the boards then it connects the IRQ handler. The driver start should fail if anything goes wrong.

4.1.6.1 Where to Implement Needed Configuration Changes

A 2.x version driver consists of two parts. One part is the prebuilt `<drvname>.sys` binary-object (the driver core), the other part is the `caninit.c` source file that should be compiled as Downloadable Kernel Module (DKM). This is a wrapper around the driver core that provides configuration information to the driver and a simple `<drvname>Start()` function. We deliver for convenience a precompiled default version of `caninit.c` named `<drvname>.ini`.

Both should be loaded into your VxWorks target with `ld()`. An example load script is delivered as `ldcan`. Then you can use `<drvname>Start()` to get the driver running. If the default configuration fails you should end up here in the troubleshooting chapter.

Now you can try to add any changes in the configuration mentioned in the following paragraphs to the `caninit.c` file and compile this file **yourself** in your build environment. Then try the start of the driver again with the prebuilt `<drvname>.sys` and your self-built `caninit.o`.

If you have a 2.x version driver that is configured using the Workbench you can change all parameters by setting macros from the Workbench GUI. The VxWorks kernel build will integrate the prebuilt driver core `<drvname>.o` from the `libesdCAN.a` into the kernel image. Also a source file `caninit_<drvname>.c` (called a configlette) from the target tree is compiled using the macros defined via the GUI to change the configuration. It works the same way as the `caninit.c`. As a last resort you may edit the `caninit_<drvname>.c` file directly (perhaps adding a BSP specific IRQ vector translation routine via the `<drvname>GetIrqVector` function pointer?).

The 3.x version drivers are always configured via the VxWorks Workbench. But there are two build variants of the driver for a supported hardware. One variant is configured using a configlette like it is described for a 2.x version driver in the previous chapter. This kind of driver is included using macros named like `"*_ESD_CAN_*`".

The other variant relies completely on the VxBus Hardware Abstraction Layer and is included using macros named like `"*_ESD_VXBCAN_*`". Therefore, the VxBus implementation of your board must work correctly to make the driver work, i. e. if the driver doesn't work you need to fix the VxBus implementation of your BSP to provide the correct information to the driver. VxBus related issues are discussed in chapter 4.1.6.6.

4.1.6.2 Public Interface of the Version 2.x Driver Core

The following table shows the functions and variables that comprise the public API of the version 2.x driver core for the driver families C331I, C331, C200 and C200I as examples. The API for other board families can be deduced from that table. The next chapters will refer to some of these functions and variables and explain their usage.

<i>c331i_install()</i>	<i>c331_install()</i>	<i>c200_install()</i>	<i>c200i_install()</i>
<i>c331i_uninstall()</i>	<i>c331_uninstall()</i>	<i>c200_uninstall()</i>	<i>c200i_uninstall()</i>
<i>c331i_switch()</i>	<i>c331_switch()</i>		
	<i>c331PciOffset</i>	<i>c200PciOffset</i>	
<i>c331iIrqOffset</i>	<i>c331IrqOffset</i>	<i>c200IrqOffset</i>	
<i>c331iGetIrqVector</i>	<i>c331GetIrqVector</i>	<i>c200GetIrqVector</i>	<i>c200iGetIrqVector</i>
<i>c331iAttachIrqHandler</i>	<i>c331AttachIrqHandler</i>	<i>c200AttachIrqHandler</i>	<i>c200iAttachIrqHandler</i>
<i>c331iDetachIrqHandler</i>	<i>c331DetachIrqHandler</i>	<i>c200DetachIrqHandler</i>	<i>c200iDetachIrqHandler</i>

Notes:

- The `<drvname>_switch()` function is only provided for active boards with onboard microcontroller.

- The `<drvname>PciOffset` variable is only present for boards on the PCI bus.
- The `<drvname>*IrqHandler` and `<drvname>GetIrqVector` variables are present since version 2.7.x.

4.1.6.3 Public Interface of the VME-CAN4 Driver Core

The public interface of the VME-CAN4 driver core differs from the previous table and is detailed in the following table.

<code>can4_install()</code>	install routine
<code>can4_uninstall()</code>	uninstall routine
<code>can4_switch()</code>	switch CAN-2.0A / CAN-2.0B (see 4.1.5.6)
<code>can4_flags</code>	driver flags (see 4.1.3.1)
<code>can4_tickFreq</code>	timestamp frequency (see 4.1.3.1)
<code>can4_irqNumOffset</code>	like <code><drvname>IrqOffset</code>
<code>can4_pflntVectorGet</code>	like <code><drvname>GetIrqVector</code>
<code>can4_irqConnectMode</code>	select the used interrupt connect routine
<code>can4_pflntConnect</code>	pointer to a user provided interrupt connect routine
<code>can4_irqDisconnectMode</code>	select the used interrupt disconnect routine
<code>can4_pflntDisconnect</code>	pointer to a user provided interrupt disconnect routine

The variables `can4_flags` and `can4_tickFreq` are implemented as global variables because they don't need to be specified per board but only for the whole driver. Also there was no space left in the per board parameter structure.

For further information refer to the comments in the `caninit.c` / `caninit_ican4.c` files.

4.1.6.4 Address Translation and Board Access Issues

All addresses that you specify in the `base` member of the `<drvsig>_CAN_INFO` structure are always bus addresses. I. e. depending on the bus type the board is attached to these addresses will be ISA-Bus addresses for ISA-Boards, PCI-Bus addresses for PCI-Boards and VME-Bus addresses for VME-boards.

4.1.6.4.1 ISA-Bus Address Translation

The driver binary `<drvname>.sys` uses the following functions to access the board on the ISA Bus:

- `sysOutWord()`
- `sysOutByte()`
- `sysInWord()`
- `sysInByte()`

Any special accesses or address translation must be handled by these functions provided by your BSP. I. e. calling `sysInWord()` with a parameter `0x1e8` should generate an ISA-Bus word (16 bit) read to address `0x1e8`. Please refer to the board's hardware manual for the required number of I/O addresses and make shure that no ISA address ranges overlap with other devices.

4.1.6.4.2 PCI Bus Access Issues

The driver must be able to access the board on the PCI Bus. To test the accessibility you can find the board manually from the target shell and try to access it with the memory dump command **d()**.



Attention:

To have the PCI show functions used below included in your kernel image you need to include different modules into your kernel image build depending on the VxWorks version and bus configuration:

- For VxWorks versions before VxWorks 6.x or kernel images without VxBus support you must include `INCLUDE_PCI_CFGSHOW`.
- For VxWorks versions from VxWorks 6.x and later if you have VxBus enabled (`INCLUDE_VXBUS`) you must include `INCLUDE_VXBUS_SHOW` and `INCLUDE_PCI_BUS_SHOW`. Be aware that having VxBus enabled breaks the old PCI show module enabled with `INCLUDE_PCI_CFGSHOW`.

Use **pciConfigTopoShow()** to display all PCI devices VxWorks is aware of. This will also show you if all PCI buses have been detected that are implemented on your host board.



Watch out for PCI bridges that may need to be explicitly enabled when configuring your VxWorks kernel.

An example is the PCI-PCI bridge located on the PMCSPAN extension board for Emerson (former Motorola) MVME boards (i. e. MVME5110) that has to be explicitly included in the PCI Autoconfiguration run for that BSP.

After you checked that all expected PCI buses are present you may use **pciDeviceShow()** with a bus number parameter to look for **esd** CAN interface boards. Refer to chapter 1.5.1 to find the PCI IDs you have to look for. **pciDeviceShow()** only shows the PCI Vendor ID and the PCI Device ID. To fully identify the board you need to display also the PCI Subsystem Vendor ID and the PCI Subsystem Device ID.

Here follows some example output of a successful search for a CAN-PCI/331 board. The data used for the identification is marked **red** there:

```
-> pciConfigTopoShow
[0,1,0] type=P2P BRIDGE to [1,0,0]
      base/limit:
          mem= 0xe8000000/0xffffffff
          preMem=0xffff0000/0x000fffff
          I/O= 0xd000/0xdfff
      status=0x2220 ( 66MHZ DEVSEL=1 MSTR_ABORT_RCV )
      command=0x0007 ( IO_ENABLE MEM_ENABLE MASTER_ENABLE )
[1,0,0] type=DISP_CNTLRL
      status=0x0210 ( CAP DEVSEL=1 )
      command=0x0007 ( IO_ENABLE MEM_ENABLE MASTER_ENABLE )
      bar0 in 32-bit mem space @ 0xe8000000
[0,18,0] type=NET_CNTLRL
      status=0x0210 ( CAP DEVSEL=1 )
      command=0x0007 ( IO_ENABLE MEM_ENABLE MASTER_ENABLE )
      bar0 in I/O space @ 0x0000e800
      bar1 in 32-bit mem space @ 0xf5202000
```

```
[0,19,0] type=UNKNOWN (0x80) BRIDGE
        status=0x0280 ( FBTB DEVSEL=1 )
        command=0x0003 ( IO_ENABLE MEM_ENABLE )
        bar0 in 32-bit mem space @ 0xf5200000
        bar1 in I/O space @ 0x0000ec00
        bar2 in 32-bit mem space @ 0xf5100000
        bar3 in 32-bit mem space @ 0xf5000000
        bar4 in 32-bit mem space @ 0xf5201000
value = 0 = 0x0
-> pciDeviceShow 0
Scanning functions of each PCI device on bus 0
bus      device    function  vendorID  deviceID  class/rev
    0         0         0    0x1106    0x0598  0x06000004
    0         1         0    0x1106    0x8598  0x06040000
    0        18         0    0x10b7    0x9200  0x02000078
    0        19         0    0x10b5    0x9050  0x06800001
value = 0 = 0x0
-> pciHeaderShow 0,19,0
vendor ID =                0x10b5
device ID =                0x9050
command register =         0x0003
status register =          0x0280
revision ID =              0x01
class code =               0x06
sub class code =           0x80
programming interface =    0x00
cache line =               0x08
latency time =             0x00
header type =              0x00
BIST =                     0x00
base address 0 =           0xf5200000
base address 1 =           0x0000ec01
base address 2 =           0xf5100000
base address 3 =           0xf5000000    <- BAR3 base needed in 121
base address 4 =           0xf5201000
base address 5 =           0x00000000
cardBus CIS pointer =      0x00000000
sub system vendor ID =     0x12fe
sub system ID =            0x0001
expansion ROM base address = 0x00000000
interrupt line =           0x09
interrupt pin =            0x01
min Grant =                0x00
max Latency =              0x00
value = 0 = 0x0
```

This dump shows the conditions after a PCI PnP (Plug-and-play) run has successfully configured device addresses.

After you have found the device in the PCI config space you could try to access the device using the **d()** command from the target shell. Before trying that the next chapter will explain if an address translation is needed and then show how to access the device.

4.1.6.4.3 PCI Bus Address Translation

In most cases for a **x86** PC-style motherboard the CPU address (*cpuAdrs*) to access the PCI board at its configured PCI address (*pciAdrs*) is the same. But on some (often PowerPC based) host boards the *cpuAdrs* differs from the *pciAdrs* and the following equation needs to be applied:

$$cpuAdrs = pciAdrs + pciOffset$$

Therefore the driver reads the *pciAdrs* from the BARx registers of the PCI device on the CAN interface board and calculates the *cpuAdrs* it will use to access the device by adding *pciOffset*.

Our driver binary `<drvname>.sys` exports a variable `<drvname>PciOffset` that is used by the driver in this calculation. Therefore you need to set up that variable to a value that fits to your BSP. This is normally done in the `caninit.c` file before the `<drvname>_install()` function is called.



For a non-VxBus driver configured from the VxWorks Workbench you should enable the `INCLUDE_ESD_CAN_BSP_PNP_QUIRK` which will allow you to set the offset `<drvname>PciOffset` using the macro `ESD_CAN_PCI_ADR_OFFSET` in the configlet `caninit_<drvname>.c` located in the target tree.

To find the right value for the `<drvname>PciOffset` you should refer to the documentation of your target BSP. To test the validity of that value you could test the access from the VxWorks target shell.

CAN-PCI/331 Access Test Example

The following example is for the CAN-PCI/331 board. It is also valid for all boards of the C331 family (consider the different PCI IDs for the other boards of that family). You could verify the *c331PciOffset* by first extracting the BAR3 value from the output of `pciHeaderShow()` (how to find the device and get that value is described in 4.1.6.4.2), then calculate the resulting *cpuAddr* for that area to use the `d()` command to look into the board's memory. From section 4.1.6.4.2 we find the *pciAddr* for BAR3 being `0xf5000000`. With a *c331PciOffset* of zero in this case we get also `0xf5000000` for the *cpuAddr* here.

```
-> d 0xf5000000
NOTE: memory values are displayed in hexadecimal.
0xf5000000: 4000 fe07 0000 1800 1000 4f43 444c 2020 *.@.....COLD *
0xf5000010: 0000 0000 0000 4f4e f945 ff00 00fa 7c35 *......NOE.....5|*
0xf5000020: 8f7f 0400 0070 0053 fc66 7c35 4100 0000 *....p.S.f.5|.A..*
0xf5000030: 7c35 0400 2000 0970 fa41 6e00 ea43 4c00 *5|... p.A..nC..L*
0xf5000040: d832 d832 c851 faff 0f70 f941 4000 000f *2.2.Q...p.A..@..*
0xf5000050: fa43 7e00 d910 c851 fcff b94e 4000 000f *C...~..Q...N..@..*
0xf5000060: 7c35 0700 5c00 7c35 0700 6000 fc23 aaaa *5|...`5|...`#...*
0xf5000070: aaaa 0f00 f0ff fc23 5555 5555 0300 f0ff *......#.UUUU....*
0xf5000080: b90c aaaa aaaa 0f00 f0ff 0c67 7c35 0500 *......g.5|..*
0xf5000090: 5c00 7c35 0500 6000 7c35 0000 1c00 7c35 *.`5|...`5|....5|*
0xf50000a0: 4200 1e00 fa4e 5200 0000 0000 0000 0000 *.B..N..R.....*
0xf50000b0: 0000 0000 0080 f07f 0740 303c 0740 305c *......@.<0@.\0*
0xf50000c0: 0000 0000 0090 307c 0091 307c 0092 307c *......|0..|0..|0*
0xf50000d0: 7c25 07c0 b078 4800 2f00 c000 0100 754e *%|..x..H./....Nu*
0xf50000e0: 0d00 60af 0d00 8686 1000 754e 3763 352e *...`.....Nuc7.5*
0xf50000f0: 0000 0008 8827 4f4e 0060 e200 0060 b419 *....'.NO`....`...*
value = 0 = 0x0
```

The values marked in **red** in the dump before are some kind of magic numbers that you can find in the BAR3 area of a C331 board. This way you can verify the *c331PciOffset* needed to be zero in this case.

CAN-PCI/200 Access Test Example

The following example is for the CAN-PC/200 board. It is also valid for all boards of the C200 family (consider the different PCI IDs for the other boards of that family). You could verify the *c200PciOffset* by first extracting the BAR2 value from the output of *pciHeaderShow()*, then calculate the resulting *cpuAddr* for that area to use the *d()* command to look into the board's memory. From the output below we find the *pciAddr* for BAR2 being 0xf5201000. With a *c200PciOffset* of zero in this case we get also 0xf5201000 for the *cpuAddr* here.

```
-> pciHeaderShow 0,17,0
vendor ID = 0x10b5
device ID = 0x9050
command register = 0x0003
status register = 0x0280
revision ID = 0x02
class code = 0x0c
sub class code = 0x09
programming interface = 0x00
cache line = 0x08
latency time = 0x00
header type = 0x00
BIST = 0x00
base address 0 = 0xf5204000
base address 1 = 0x0000d801
base address 2 = 0xf5201000 <- BAR2 needed for access test
base address 3 = 0x00000000
base address 4 = 0x00000000
base address 5 = 0x00000000
cardBus CIS pointer = 0x00000000
sub system vendor ID = 0x12fe
sub system ID = 0x0004
expansion ROM base address = 0x00000000
interrupt line = 0x0b
interrupt pin = 0x01
min Grant = 0x00
max Latency = 0x00
value = 0 = 0x0
-> d 0xf5201000,0x20
NOTE: memory values are displayed in hexadecimal.
0xf5201000: ff21 e00c 0000 0000 0000 ffff ffff ffff *!.....*
0xf5201010: ffff ffff ffff fbff 0900 0900 fffe 00ff *.....*
0xf5201020: ff21 e00c 0000 0000 0000 ffff ffff ffff *!.....*
0xf5201030: ffff ffff ffff fbff 0900 0900 fffe 00ff *.....*
value = 0 = 0x0
```

The hex values 0x21 in the dump before show that you have found the CAN-PCI/200 (these values can only be seen after a power on reset). This way you can verify the *c200PciOffset* needed to be zero in this case.

4.1.6.4.4 Manual Configuration of PCI Addresses

If you have the case of a host board / VxWorks BSP without a PCI PnP run you need to setup the PCI stuff by hand. To do that you enter in the *base* member of your driver parameter structure a value different from `0xFFFFFFFF`. Then the driver will write PCI addresses into the PCI device bridges of our **esd** CAN boards. See the `readme` files from the driver packages how much memory space is needed for that specific board.

It is your responsibility to administer available memory ranges. Check all PCI-PCI bridges for being transparent at the right addresses. Verify that the PCI command register enables PCI device's memory spaces. After that you can check the accessibility via the `d()` command as described before.

4.1.6.4.5 VME Bus Access Issues (VME-CAN4)

In the *base* member of the driver parameter structure you specify the VME bus address you want to use. The driver uses the BSP specific `sysBusToLocalAdrs()` function to translate the configured VME address to the CPU address needed to access the VME-CAN4 board. Because the VME-CAN4 can be configured to appear either in the VME-A24 address space or in the VME-A32 address space you need to select the desired address space using the *vmeSpace* member of the configuration structure. The possible values defined in the provided `caninit.c` file for that variable refer to definitions from VxWorks' `vme.h`:

```
#define VME_A16      VME_AM_SUP_SHORT_IO      /* 0x2d */
#define VME_A24      VME_AM_STD_SUP_DATA      /* 0x3d */
#define VME_A32      VME_AM_EXT_SUP_DATA      /* 0x0d */
```

Be aware of the fact that after a reset the VME-CAN4 board only appears in the VME-A16 address space and waits for further configuration. When the driver starts it takes the desired VME addresses from the configuration structure and programs it into the VME-CAN4 hardware. To see how to do that refer to the VME-CAN4 hardware manual. Only after that step the VME-CAN4 responds in the configured address space and range.

VME-CAN4 Access Test Example

In this example we assume a configuration of `VME_A32 (0x0d)` for *vmeSpace* and `0x08000000` for *sramBase* in `CAN4_CAN_INFO` structure. After the driver started to verify that the board can be accessed by the CPU you may do the following on the target shell:

```
-> adrPointerBuf=0
New symbol "adrPointerBuf" added to kernel symbol table.
adrPointerBuf = 0x17611fd0: value = 0 = 0x0
-> sysBusToLocalAdrs(0xd,0x08000000,&adrPointerBuf)
value = 0 = 0x0
-> adrPointerBuf
adrPointerBuf = 0x17611fd0: value = 536870912 = 0x20000000
-> d 0x20000000,0x10
NOTE: memory values are displayed in hexadecimal.
0x20000000:  4341 4e5f 564d 455f 3034 5f56 5f30 2e43  *CAN_VME_04_V_0.C*
0x20000010:  0001 0000 0001 0000 1500 0000 0000 0000  *.....*
value = 0 = 0x0
```

This way you can manually verify that the driver can access the dual-ported RAM of the VME-CAN4. After translation of the `VME_A32` address `0x08000000` to a valid CPU address (here `0x20000000`) you should be able to dump the **red** marked string from the start of the dual-ported RAM area.

4.1.6.5 Interrupt Connection Issues

The driver needs a correctly connected interrupt handler to work. There are three main categories that interrupt issues could belong to:

- The functions to connect a C interrupt handler differ depending on CPU architecture or the BSP of the host board. There is an assortment of functions possible that may only be provided on a part of architectures and BSPs. Some of them are ***intConnect()*** and ***pcilntConnect()***. Some BSPs provide neither of both routines. Also for some interrupts you may need to use ***intConnect()*** and for others on the same board ***pcilntConnect()***.
- To connect a C interrupt handler the VxWorks routines need the interrupt vector address as parameter. How this interrupt vector address is determined from the interrupt number depends on CPU architecture and may also depend on the BSP and other environmental conditions. Even the interrupt number itself may depend on the BSP. To determine the interrupt number the BSP's documentation should be the first source.
- The driver uses the configured IRQ number to enable the interrupt utilizing ***sysIntEnable()*** or ***sysIntEnablePIC()*** depending on the CPU architecture.

These categories lead to an unusable driver. Please check at first with your BSP's documentation if the right functions are used for the interrupt numbers in question. Also check if the IRQ number is correct for your BSP and board (the driver prints the used IRQ number in the start-up banner). In a second step you may try to find out if any interrupt number translation is needed.

4.1.6.5.1 Interrupt Connection: Call of Wrong Function

Sometimes the failed interrupt connection has its cause by the driver using the wrong function to connect the interrupt. In some cases you will not be able to load the driver object because the BSP lacks some of the needed functions. This may for example happen if the driver is compiled to use the ***pcilntConnect()*** function but your BSP doesn't provide one. In this case check if there is another build variant of the driver available that uses the ***intConnect()*** call instead (see chapter 4.1.4.1 on how to identify build variants).

In any case you could inspect the **<drvname>.sys** file with the nm tool on the host to see which connect functions are utilized. The nm tool's full name changes depending on the target architecture (e. g. nmppc, nmpentium or the like).

There are some releases of the driver that support the selection whether ***intConnect()*** or ***pcilntConnect()*** is used to connect the interrupt. The behaviour is controlled via the `flags` member of the parameter structure (see also chapter 4.1.3.1 for information). To decide which function needs to be used you should refer to the documentation of your BSP.



For a non-VxBus driver configured from the VxWorks Workbench you can change the `flags` by adapting the `ESD_CAN_<drvsig>_FLAGS` macro to your needs.

To alleviate the problems arising from the use of the wrong interrupt connect or disconnect routines recent driver versions export function pointers that can be used by the user to provide callback functions to the driver. These callback function will be used by the driver core to connect or disconnect the interrupt handler.

User specific driver core IRQ connect function selection

Drivers that support this feature export the function pointers `<drvname>AttachIrqHandler` and `<drvname>DetachIrqHandler`. Addresses of user specific function should be assigned to these pointers. The driver core calls these functions via the function pointers to attach or detach the interrupt handler. If the user specified functions return the error code `ENOSYS` (from `errno.h`) then the driver core will execute as fallback its default connect and disconnect routines.

Please refer to the `caninit.c / caninit_<drvname>.c` file that belongs to your driver to see how these function pointers can be initialized. It can be done either manually in these files or via macros changeable via the Windriver Workbench.

VME-CAN4 Connect Function Selection

The CAN4 driver calls connect and disconnect functions using a function pointer that the user has to initialise. The `caninit.c / caninit_ican4.c` file provide the means to change this initialisation. You have to select a connection mechanism via the variable `can4_irqConnectMode` and provide an entry for a connect function via the function pointer `can4_pflntConnect`. The macros to change these values from the WindRiver workbench are `ESD_CAN_ICAN4_IRQ_CON_MODE` and `ESD_CAN_ICAN4_IRQ_CON_FUNC`. For the disconnect functions the same mechanism with other variables is implemented. To make this more clear please refer to the source files mentioned in this paragraph.

4.1.6.5.2 Interrupt Connection: Wrong IRQ Number to Vector Translation

The vxWorks functions to connect an interrupt take unfortunately an interrupt vector address as parameter and not an interrupt number which would be more sound. This way the task to translate an interrupt number to the correct interrupt vector address is imposed on the driver. Most unfortunately the way to do this translation also involves a macro `INUM_TO_IVEC()` that is at least architecture specific.

In the following paragraphs it is described how the driver does the IRQ number to IRQ vector translation by default and how it may be tuned to fit to your BSP.

The default formula for the number to vector translation involves the value of `sysVectorIRQ0` which may be either a global variable of the BSP or a macro used at build time. The typical values of the macro used at build time are `0x20` for the **x86** architecture and `0x00` for the **PPC** architecture.

```
irqVector = INUM_TO_IVEC( irqNumber + sysVectorIRQ0 )
```

The `irqNumber` mentioned in the formula is the IRQ number provided via the parameter structure except for PCI based boards where the interrupt number is taken by default from the PCI config space. To compensate for a wrong value of `sysVectorIRQ0` or any peculiar differences of your BSP most drivers provide an additional IRQ offset variable that can be used to tune the translation to your needs as the formula below shows.

```
irqVector = INUM_TO_IVEC( irqNumber + irqOffset + sysVectorIRQ0 )
```

Every driver that supports this variable exports this as `<drvname>IrqOffset`. It should be adjusted to your needs in the `caninit.c` file before `<drvname>_install()` is called.



For a non-VxBus driver configured from the VxWorks Workbench you should enable the `INCLUDE_ESD_CAN_BSP_PNP_QUIRK` which will allow you to set the offset `<drvname>IrqOffset` using the macro `ESD_CAN_PCI_IRQ_OFFSET` in the configlet `caninit_<drvname>.c` located in the target tree.

Using only this approach there are still some translation issues left that can't yet be solved:

- Still the `INUM_TO_IVEC()` macro is compiled into the driver binary. If this macros is BSP specific the driver won't work.
- With a single *irqOffset* value the IRQ numbers can only be shifted. Some BSPs that support virtual IRQ numbers or need a IRQ number reordering (use `INT_NUM_GET()`) can't be supported.

To provide a solution more recent versions of the driver export for that reason a function pointer variable `<drvname>GetIrqVector`. If that variable doesn't contain a NULL pointer then the driver calls that function to translate the *irqNumber* to the *irqVector*. You have there the opportunity to implement any interrupt number to interrupt vector translation formula that you need for your environment. A sample implementation of that function that avoids the `INUM_TO_IVEC()` and `INT_NUM_GET()` issues looks like this (for a C200I board):

```
/*-----*/
/* Callback to return assigned IRQ vector */
/*-----*/
LOCAL VOIDFUNCPTR * c200iIrqVector(int irq)
{
    VOIDFUNCPTR *irqVec;

    irqVec = INUM_TO_IVEC(INT_NUM_GET(irq));

    printf("--> BSP specific IRQ vector is 0x%x\n", irqVec);

    return(irqVec);
}
```

Now setup the pointer `c200iGetIrqVector` with the address of `c200iIrqVector()` and the driver will call it to do the IRQ number to IRQ vector translation.

VME-CAN4 Interrupt Number Restrictions and Offset

Please be aware that the IRQ number you provide to the driver via the `irq_num` member of the parameter structure needs to be a multiple of eight (8). This has its reason in a hardware limitation and is described in more detail in the hardware manual of the VME-CAN4.

If you need to provide an IRQ number offset you should set the variable `can4_irqNumOffset` accordingly. If you configure the driver from the VxWorks Workbench you should adapt the macro `ESD_CAN_ICAN4_IRQ_OFFSET` as needed which in turn will be used to setup `can4_irqNumOffset` in `caninit_ican4.c`.

4.1.6.6 VxBus Driver Prerequisites

The VxBus API provides the first time something that could be called a Hardware Abstraction Layer under VxWorks. Therefore the problems to find and claim the right resources for a driver should become less evident. But on the other hand it is now the duty of the BSP to setup and configure the VxBus libraries provided by WindRiver to establish the hardware abstraction. Here will follow some hints to check whether the BSP has done its job good enough to support our VxBus drivers for PCI interface boards.

The following things must be added to your kernel configuration to do these tests:

- INCLUDE_VXBUS adds VxBus support itself
- INCLUDE_VXBUS_SHOW adds VxBus show routines
- INCLUDE_PCI_BUS_SHOW adds PCI show routines based on the VxBus libraries

In the following paragraphs we will try to show you how to test if your BSP does not only have simplistic support for VxBus but at least includes the VxBus PCI hardware abstraction that our driver needs. The **vxBusShow()** routine will show you if the VxBus system is aware of the PCI device you plugged in. You may also use **vxbTopoShow()** to get a more condensed output that shows only the buses and devices found.

4.1.6.6.1 Example of Working VxBus Implementation

Here follows the output of a working implementation with the important parts marked in **red** and some unimportant entries removed. It shows the conditions without any **esd** CAN driver included into the VxWorks image but with an **esd** CAN board plugged in:

```
-> vxBusShow
Registered Bus Types:
  MII_Bus @ 0x004c52ac
  PCI_Bus @ 0x004c5840
  PLB_Bus @ 0x004c58c0

Registered Device Drivers:
  pentiumPci at 0x004c4ae0 on bus PLB_Bus, funcs @ 0x004c4b68
  i8253TimerDev at 0x004c55e0 on bus PLB_Bus, funcs @ 0x004c5660
  ns16550 at 0x004c5480 on bus PLB_Bus, funcs @ 0x004c5548
  m6845Vga at 0x004c4c20 on bus PLB_Bus, funcs @ 0x004c4c88
  i8042Kbd at 0x004c4b80 on bus PLB_Bus, funcs @ 0x004c4c00
  genericPhy at 0x004c5300 on bus MII_Bus, funcs @ 0x004c5340
  miiBus at 0x004c5260 on bus PCI_Bus, funcs @ 0x004c52c8
  elPci at 0x004c4f20 on bus PCI_Bus, funcs @ 0x004c4fb8
  plbCtrlr at 0x004c5880 on bus PLB_Bus, funcs @ 0x004c5978

Busses and Devices Present:
  PLB_Bus @ 0x004fb4d8 with bridge @ 0x004c58e0
    Device Instances:
      ns16550 unit 0 on PLB_Bus @ 0x004fc498 with busInfo 0x00000000
      pentiumPci unit 0 on PLB_Bus @ 0x004fc898 with busInfo 0x004fb718
      i8042Kbd unit 0 on PLB_Bus @ 0x004fe598 with busInfo 0x00000000
      m6845Vga unit 0 on PLB_Bus @ 0x004fe798 with busInfo 0x00000000
      i8253TimerDev unit 0 on PLB_Bus @ 0x004fe898 with busInfo 0x00000000
    Orphan Devices:
      i8042Mse unit 0 on PLB_Bus @ 0x004fe698 with busInfo 0x00000000
  PCI_Bus @ 0x004fb718 with bridge @ 0x004fc898
    Device Instances:
      elPci unit 0 on PCI_Bus @ 0x004fd398 with busInfo 0x00000000
      miiBus unit 0 on PCI_Bus @ 0x004fec98 with busInfo 0x004fc218
    Orphan Devices:
      (null) unit 0 on PCI_Bus @ 0x004fd198 with busInfo 0x00000000
      (null) unit 0 on PCI_Bus @ 0x004fd298 with busInfo 0x00000000
```

```
(null) unit 0 on PCI_Bus @ 0x004fe498 with busInfo 0x00000000
MII_Bus @ 0x004fc218 with bridge @ 0x004fec98
Device Instances:
    genericPhy unit 0 on MII_Bus @ 0x004fed98 with busInfo 0x00000000
Orphan Devices:
```

```
value = 1 = 0x1
```

This output tells us in the “Busses and Devices Present” section that the `pentiumPci` device instance is at `0x004fc898` (this is the PCI host controller). It provides access to the `PCI_Bus` instance at `0x004fb718` which has a backlink to the host controller instance at `0x004fc898`.

The `PCI_Bus` itself has a list of “Device Instances” for which some device driver claims responsibility (i. e. `e1Pci`). Also the `PCI_Bus` has a list of “Orphan Devices”. This list has to grow after you plugged in the **esd** CAN board. You may inspect the device instance for being an **esd** CAN board (check PCI IDs) by using ***pciDevShow()***. For the current example selecting the orphaned device at `0x004fd298` you will get:

```
-> pciDevShow 0x004fd298
pDev @ 0x004fd298    [0,17,0]
    devID    = 0x9050
    vendID    = 0x10b5
value = 17 = 0x11
```

At least from these PCI IDs it may be an **esd** CAN board. To be sure you can interrogate the `PCI_Bus` instance at `0x004fc898` with ***vxbPciHeaderShow()*** like this:

```
-> vxbPciHeaderShow 0x004fc898,0,17,0
vendor ID = 0x10b5
device ID = 0x9050
command register = 0x0003
status register = 0x0280
revision ID = 0x02
class code = 0x0c
sub class code = 0x09
programming interface = 0x00
cache line = 0x08
latency time = 0x00
header type = 0x00
BIST = 0x00
base address 0 = 0xf5204000
base address 1 = 0x0000d801
base address 2 = 0xf5201000
base address 3 = 0x00000000
base address 4 = 0x00000000
base address 5 = 0x00000000
cardBus CIS pointer = 0x00000000
sub system vendor ID = 0x12fe
sub system ID = 0x0004
expansion ROM base address = 0x00000000
interrupt line = 0x0b
interrupt pin = 0x01
min Grant = 0x00
max Latency = 0x00
value = 0 = 0x0
```

Judging from this output you know now that this device is a CAN-PCI/200 and that the `VxBus` implementation is aware of the hardware so far. Even the interrupt assignment could now be displayed with ***plbIntrShow()*** for the same orphaned device at `0x004fd298`:

```
-> plbIntrShow 0x004fd298,1
(null)_0:
    numVectors      = 1 (defined in hwconf.c for device)
    intVecList      @ 0x00000158
    intLvlList      @ 0x0000000b
    intCtlrList     @ 0x00000000
    pIntCtlrTable   @ 0x00000000
value = 0 = 0x0
```

To display all this information, we used native VxBus routines to be sure that we really talk to the VxBus implementation. If you have seen all this information on your board the VxBus driver should work. If you only need to see some information about the PCI configuration or want to test if the board itself is accessible you may use the functions described in chapter 4.1.6.4.2.

4.1.6.6.2 Example of not Working VxBus Implementation

In the following section is some output from an insufficient VxBus implementation shown that won't be able to support our VxBus CAN driver. This is the condition with the CAN board plugged in but without the esd VxBus CAN driver compiled into the VxWorks kernel image.

```
-> vxBusShow
Registered Bus Types:
    MII_Bus @ 0x00311a04
    PCI_Bus @ 0x0031162c
    PLB_Bus @ 0x00311648

Registered Device Drivers:
    mottsec at 0x003118e8 on bus PLB_Bus, funcs @ 0x003117d0
    mdio at 0x00311be0 on bus PLB_Bus, funcs @ 0x00311bac
    mv88E1x11Phy at 0x00311b00 on bus MII_Bus, funcs @ 0x00311ad0
    bcm54xxPhy at 0x00311a70 on bus MII_Bus, funcs @ 0x00311a40
    genericPhy at 0x00311b6c on bus MII_Bus, funcs @ 0x00311b60
    miiBus at 0x003119bc on bus PCI_Bus, funcs @ 0x00311960
    miiBus at 0x0031197c on bus PLB_Bus, funcs @ 0x00311960
    plbCtlr at 0x00311670 on bus PLB_Bus, funcs @ 0x00311664

Busses and Devices Present:
    PLB_Bus @ 0x0035ae78 with bridge @ 0x003116b0
        Device Instances:
            mottsec unit 0 on PLB_Bus @ 0x0035be38 with busInfo 0x00000000
            mottsec unit 1 on PLB_Bus @ 0x0035bf38 with busInfo 0x00000000
            miiBus unit 0 on PLB_Bus @ 0x0035c238 with busInfo 0x0035b1b8
            miiBus unit 1 on PLB_Bus @ 0x0035c538 with busInfo 0x0035b1f8
        Orphan Devices:
        MII_Bus @ 0x0035b1b8 with bridge @ 0x0035c238
            Device Instances:
                genericPhy unit 0 on MII_Bus @ 0x0035c338 with busInfo 0x00000000
            Orphan Devices:
            MII_Bus @ 0x0035b1f8 with bridge @ 0x0035c538
                Device Instances:
                    genericPhy unit 1 on MII_Bus @ 0x0035c638 with busInfo 0x00000000
                Orphan Devices:

value = 1 = 0x1
```

At least the `PCI_Bus` shows up in the “Registered Bus Types” section. But in the “Buses and Devices Present” section there is no instance of the `PCI_Bus` present. Therefore the VxBus layer doesn't know anything about the PCI devices present and cannot provide any information to our VxBus PCI driver. Instead we miss a driver for a PCI host bridge in the “Registered Device Drivers” section and based on that driver an instance on the `PLB_Bus` that could serve as root bridge device for a `PCI_Bus` instance.

4.2 QNX®

This chapter covers the necessary steps to install, configure and start the device drivers for **esd** CAN boards supporting the real time operating system QNX®.

CAN device drivers are available for QNX® 6.x, QNX® 7.x and QNX® 8.x for different CPU target architectures.



Note:

The device driver installation for supported CAN boards on the legacy QNX 4.x version is covered in the older version 4.9 of this manual, which is downloadable from our website:

[NTCAN Part 2: Installation Guide \(Legacy OS support\), Rev. 4.9.](#)



Note:

CAN device driver for **esd** embedded boards with support for QNX® are part of the BSP and are not deployed as a separate CAN driver package.

A VxWorks CAN driver package supports one or more families of **esd** CAN Interfaces for a certain target architecture. The package contains a file `relnotes.htm` in HTML format which contains the revision history of the drivers and late-breaking information which did not make into one of the manuals. Please also read this file before installing the driver.



Note:

The CAN device driver for **esd** embedded boards with support for QNX are part of the BSP and are not deployed as a separate CAN driver package.

4.2.1 CAN Board Support Overview

Device drivers for **esd** CAN interfaces are available for different QNX® versions and CPU architectures. A single device driver often supports more than one CAN interface type (refer to chapter 1.4 for the CAN interfaces which belong to the same interface family).

QNX	6.x	7.0		7.1	8.x
Architecture	x86	x86	x86_64	x86_64	x86_64
CAN Family					
C200I	✓	✓	✓	✓	-
C331I	✓	✓	✓	✓	-
C200	✓	✓	✓	✓	-
C331	✓	✓	✓	✓	-
C360	✓	✓	✓	✓	-
C400	✓	✓	✓	✓	-
C402	✓	✓	✓	✓	✓
C405	✓	✓	✓	✓	-
USB2	✓	✓	✓	✓	✓



Note:

If a combination of CAN interface, QNX® version and/or CPU architecture is currently unsupported, please contact the **esd** support for help.

4.2.2 Driver Integration

The support for an **esd** CAN boards consists of a device driver (`devcan-board-proto`) and a shared library (`libntcan.so`).

The name of the device driver binary follows the naming convention described in Table 2 in the column *Driver Name - Naming Convention II* by replacing `board-proto` with the matching combination for the respective CAN board. This hardware specific device driver (that usually covers a complete CAN device family) is implemented as a QNX® [Resource Manager](#) which is a server program that accepts messages from other programs and communicates with the CAN hardware.

This approach supports different CAN boards (managed by different resource managers on one QNX® target) with the identical API and allows different QNX® applications to use the same CAN board in parallel as well as one QNX® application to use different CAN boards.

4.2.3 Driver Installation

The CAN driver software for QNX® is shipped as compressed tar archive which follows the naming convention

`board-proto-ntovvv-arch-vx.y.z.tar.gz`

with

<code>board-<i>proto</i></code>	CAN driver family name (e.g. pci331-i20 for the C331 family driver)
<code>ntovvv</code>	QNX target version (e.g. nto710 for QNX® 7.1).
<code>arch</code>	QNX target architecture (e.g. x84_64).
<code>x.y.z</code>	Driver version (e.g. 4.2.0)

The archives contain the following files:

<i>File</i>	<i>Description</i>
Readme	Current notes and installation information
CHANGELOG	Change list / driver history
devcan-xxx-yyy	
upd-isa331	Programs for firmware update (CAN board dependent)
upd-pci331	
upd-pci360	
pciclass-pci200	Programs for setting the PCI-Class
pciclass-pci266	
pciclass-pci331	
libntcan.so.x	Dynamic NTCAN-API library (x... library version)
ntcan.h	Header of NTCAN-API
cantest.c	Source code of example program 'cantest' (see /1/)
cantest	Binary file of example program 'cantest'

The installation of the device driver on QNX® requires administrator rights on the target.

4.2.4 Driver Start

Start the resource manager (CAN device driver) by executing its binary in an interactive console or from a (start) script.

Example: For a CAN-PCI/331 board the start command is

```
devcan-pci331-i20 [arguments]
```

In a console window you can append the option -v for a startup message:

Example: For a CAN-PCI/331 board the start command is

```
devcan-pci331-i20 -v
```

should show an output like the one below:

```
devcan-pci331-i20: version 3.9.3 build 16:52:53 Mar 19 2013
(0)CAN_PCI331: Clock=0x00000000 Baud=0x7fffffff Mode=0x00000000 Nodes=can0
(0)CAN_PCI331: Hardware-version 1.0.0
```

Further arguments are described in the following chapter.

4.2.5 Driver Configuration

The QNX® device drivers (resource managers) are configured with parameters to the *ldrta.exe* command.

4.2.5.1 Command Line Parameter

The QNX® resource managers are configured via options listed in the table below which are passed as arguments to the start command described in the previous chapter:

Option	Argument	Description
-b	<i>baudrate</i>	Initial baud rate (Default value: Disconnected from CAN bus)
-C	<i>frequency</i>	Override CAN controller clock in [Hz].
-c	<i>options</i>	CAN board HW interface specific options <u>PCI/PCle:</u> CAN board selection and configuration (Default: All supported CAN boards are initialized) • <code>pci = index</code> ("PCI Index" as indicated by the QNX Shell command "<code>pci -v</code>"). • <code>sdid = id</code> ("Subsystem ID" as indicated by the QNX Shell command "<code>pci -v</code>"). • <code>irq = interrupt</code> (Override the P'n'P IRQ assignment). • <code>nomsi</code> (Enforce PCIe INTx instead of MSI interrupts). <u>ISA:</u> Hardware resource settings: • <code>io = address</code> (I/O-address) • <code>irq = interrupt</code> (An unused interrupt of the system must be determined and set with this parameter.)
-e	<i>errorinfo</i>	Extended error info (0/1=Off/On) (Default = 1)
-h	-	Display a list of all possible command line options in the QNX® console window and terminate afterwards.
-m	<i>mode</i>	Mode flags (default value = 0x00000000) 0x00000040: Enforce Listen-Only mode on all CAN ports 0x00000080: Disconnect from bus after last handle is closed 0x00800000: Prefer 64-bit DMA memory over 32-bit DMA memory 0x20000000: Do not use firmware fast mode (Only C331/C360 !) <u>Note:</u> All other bits are reserved and must be set to 0.
-n	<i>net number</i>	This option configures the logical base net number which is assigned to the first physical port of the CAN interface. The logical net numbers are incremented by one for each additional physical port. The default value for <i>net number</i> is 0.
-p	<i>prio</i>	This option configures the thread priority of asynchronous CAN events. The default value for <i>prio</i> is 19.

Option	Argument	Description
-P	<i>prio</i>	This option configures the priority of the IRQ handling thread. Default for QNX 6.x/7.x: An IRQ Handler is connected. Default for QNX 8.x: The IRQ handling thread runs with prio 254.
-R	<i>rxbuffer</i>	Rx-buffer of CAN node (Default: 128)
-t	<i>poolsize</i>	Maximal size of threadpool (1...64) (Default = 4* CAN nodes)
-v	-	Increase verbosity. The level of detail increases with each additional 'v'. The option '-vv' has the identical effect as the option '-V3' described below.
-V	<i>mask</i>	Display additional (debug) messages during startup in the QNX® console window. The mask parameter may be given as a decimal value or as a hexadecimal value prepended by a '0x'

Table 14: Command Line Parameter of QNX® Driver

PCI/PCIe board example:

The command

```
devcan-pci331-i20 -n0 -c pci=0 -n2 -c pci=1
```

starts the ressource manager for two CAN-PCI/331 boards in one system assigning the logical base net number 0 to the 1st board and logical base net number 2 to the 2nd.

ISA board example:

The command

```
devcan-isa200-sja1000 -n0 -c io=0x1e8,irq=5 -n1 -c io=0x1e0,irq=7
```

starts the ressource manager for two CAN-ISA/200 boards in one system assigning the logical base net number 0, IO address 0x1e8, IRQ 5 to the 1st board and logical base net number 1, IO address 0x1e8, IRQ 7 to the 2nd.

**Note:**

With the command

```
use devcan-pci331-i20
```

all available parameters (here CAN-PCI/331) are printed which is identical to start the resource manager just with the option "-h"

4.3 IntervalZero RTX64® and MaxRT wRTOS®

This chapter covers the necessary steps to install, configure, start and stop the device drivers for **esd** CAN boards supporting the real time operating system *RTX64®* and *MaxRT wRTOS®* of *Interval Zero, Inc.*

RTX64 and *MaxRT wRTOS* (just referred to as *wRTOS*) are no stand alone real-time operating systems, but a kernel mode add-on that extends Microsoft Windows (64 bit) with high-speed and deterministic real-time capabilities.



Attention!

As the IntervalZero support and maintenance for all RTX (32-bit) versions is expired, technical support and maintenance by **esd** is also terminated but the latest version of the device driver files are kept available. For RTX installation support please refer to earlier versions of this manual.

Note:

All **esd** CAN device drivers require an interrupt. RTX does not support sharing IRQ lines with Windows® devices, so the interrupt line used by the driver must be available for exclusive use by RTX but can be shared between RTX devices.



Finding an exclusive IRQ often requires physically moving hardware in the system or disabling other Windows® devices.

For the PCIe bus the most trouble-free solution is using the **CAN-PCIe/402** because this CAN board supports MSI which means interrupts are never shared and the problems described above do not arise.

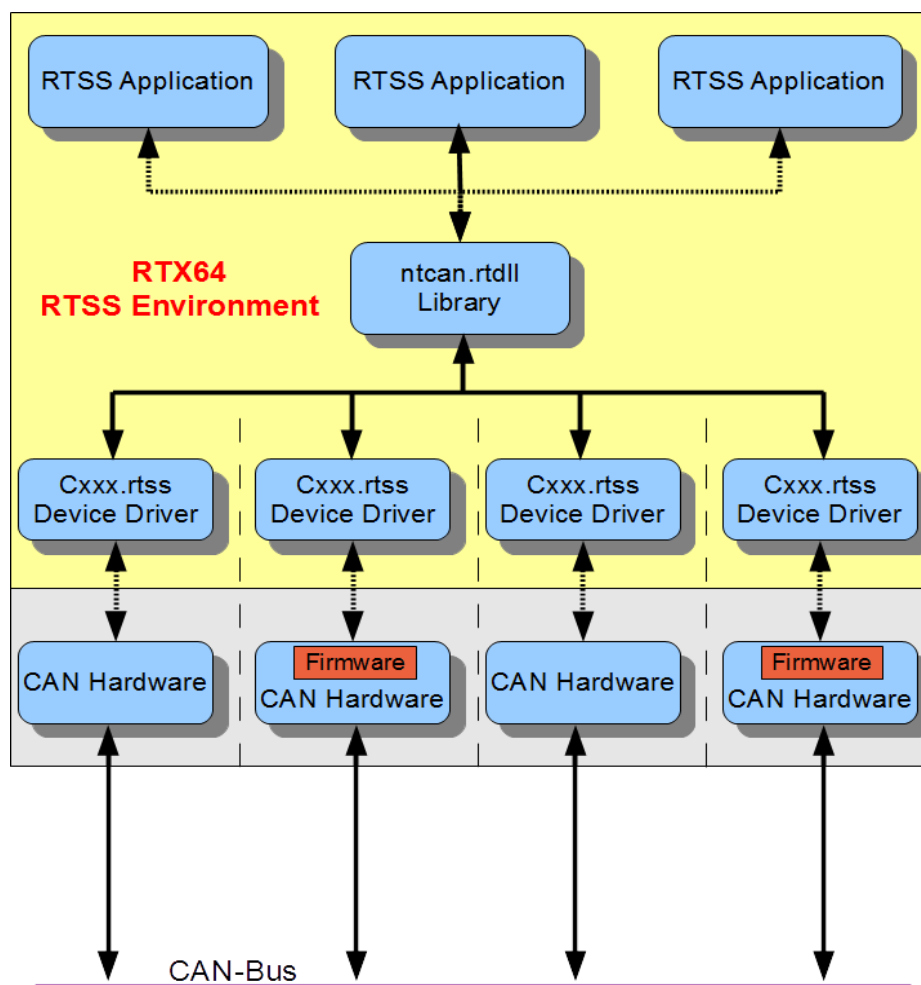
A device driver package for RTX64/wRTOS contains the following files where **<drvname>** is the device family specific driver name following driver naming convention I (see chapter 1.4).

<i>Filename</i>	<i>Description</i>
<code>bin/rtss/<drvname>.rtss</code>	The hardware specific CAN device driver.
<code>bin/rtss/ntcan.rtdll</code>	The NTCAN library.
<code>bin/rtss/cantest.rtss</code>	The <i>cantest</i> application
<code>doc/</code>	Folder with API documentation, release notes, etc.
<code>include/ntcan.h</code>	NTCAN header to compile the NTCAN based application.
<code>inf/</code>	Folder with digitally signed installation files.
<code>lib/vc/ntcan_rtss.lib</code>	Lib file for Visual Studio to link the NTCAN based application
<code>samples/cantest.c</code>	Source code of the <i>cantest</i> application

Table 15: Files of RTX64/wRTOS CAN driver package

4.3.1 Driver Integration

The support for an **esd** CAN interface consists of a device driver (Cxxx.rtss) and the NTCAN library (ntcan.rtdll) as shown in the picture below. A hardware specific device driver (which usually covers a complete CAN device family) implemented as an RTSS process must be loaded and the application must link explicitly or implicitly to the NTCAN library implemented as an RTDLL.



This approach allows using different CAN driver in the RTSS environment with the identical API and allows several RTSS processes using the same CAN hardware as well as one RTSS process using different CAN hardware.



Note: A CAN driver shares **one common** thread to exchange CAN messages with the hardware and **one common** thread to exchange data with the application via the NTCAN library for **all** CAN hardware handled by the driver.

The entries exported by the CAN device driver are only referenced via the NTCAN RTDLL library. A CAN based RTSS application will always reference to the NTCAN API exported by the NTCAN RTDLL described in /1/.



For implicitly linked RTDLLs or explicitly linking RTDLLs without specifying the full path name in LoadLibrary, the RTDLL files must be in the same directory where the RTSS executable file exists.

4.3.2 Driver Installation

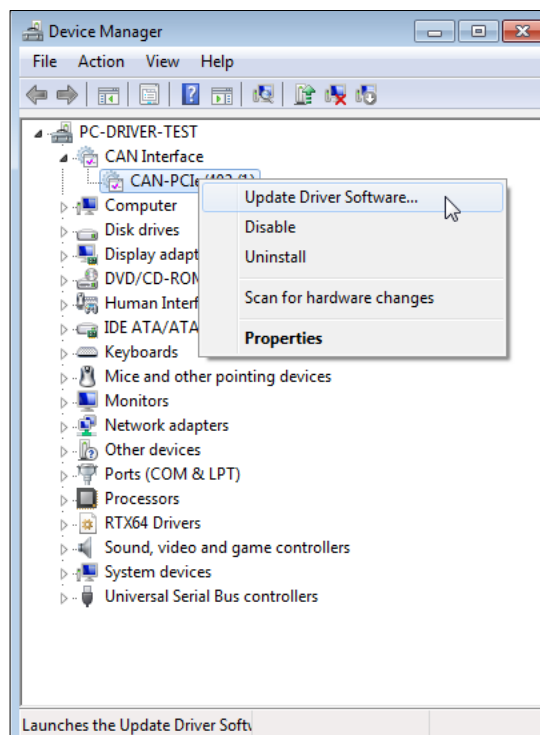
To support an **esd** CAN interface in the RTSS environment it is necessary to convert it into an RTX64/wRTOS managed device as described below (for further details refer to the *RTX64/wRTOS Runtime Documentation*).

4.3.2.1 RTX64

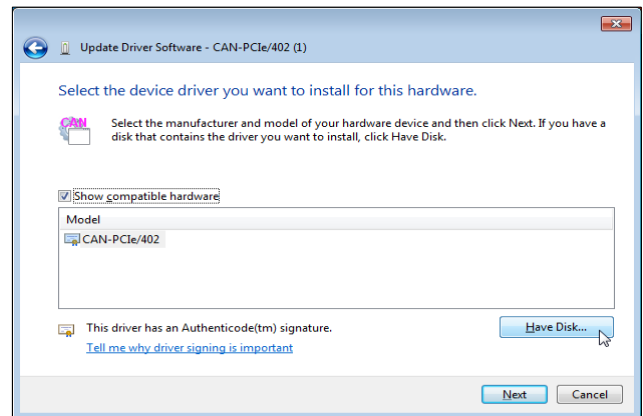
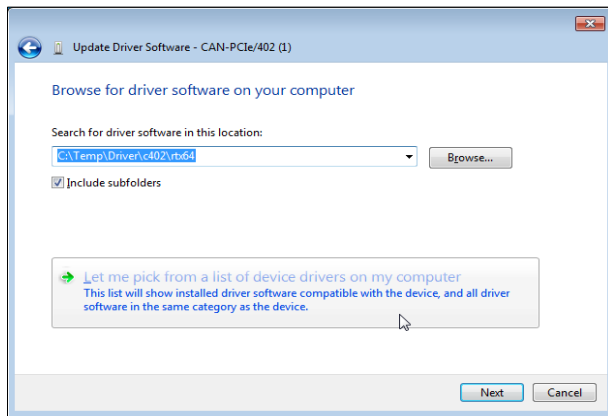
The 64-bit versions of Windows only accept digitally signed drivers. For this reason, the RTX64 device driver package comes with an INF file and a related CAT file which will allow to install the RTX64PNP driver for the CAN interface. The steps to this are like the installation process of a standard Windows device driver described in chapter 2 for the various (RTX64 supported) versions of Windows.

For a general detailed instructions how to convert a device to RTX64 please refer to the chapter Converting a PCI/PCIe Device to RTX64 in the RTX64 SDK Help.

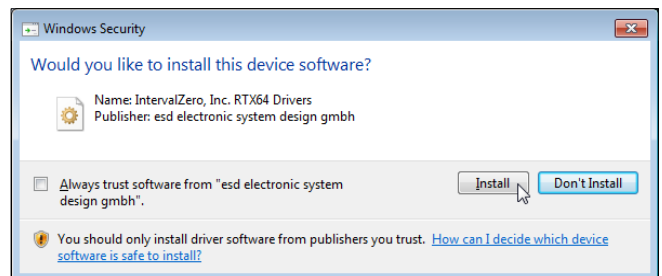
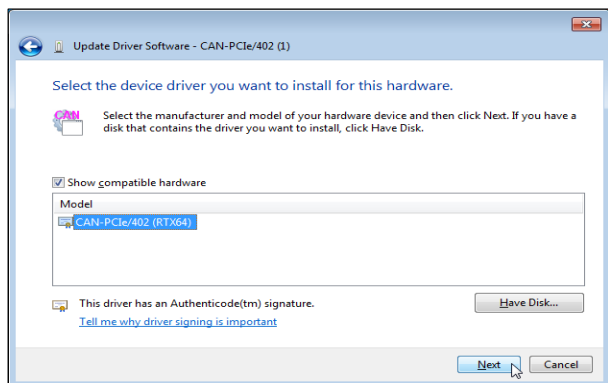
You must use the *Windows Device Manager* to assign the RTX64PNP driver to the CAN interface. In the *Device Manager* windows there will be a device under *Other Devices* with a yellow exclamation point next to the icon (if there is no device driver installed yet. The text next to the device will depend on the CAN module attached) or the device is listed under CAN Interfaces if a Windows driver is already installed (see the picture below).



Follow the installation instructions and click the **Have Disk...** button to browse for the signed copy of the `esdrtx64.inf` file.



Proceed with the installation and Windows will present you a security dialogue box. The RTX64 device driver which are loaded in the RTSS environment are not digitally signed but the INF file which assigns the CAN interface to the IntervalZero RTX64PNP driver has to. Please refer to chapter 2.9 for more details about *Digital Signatures*.

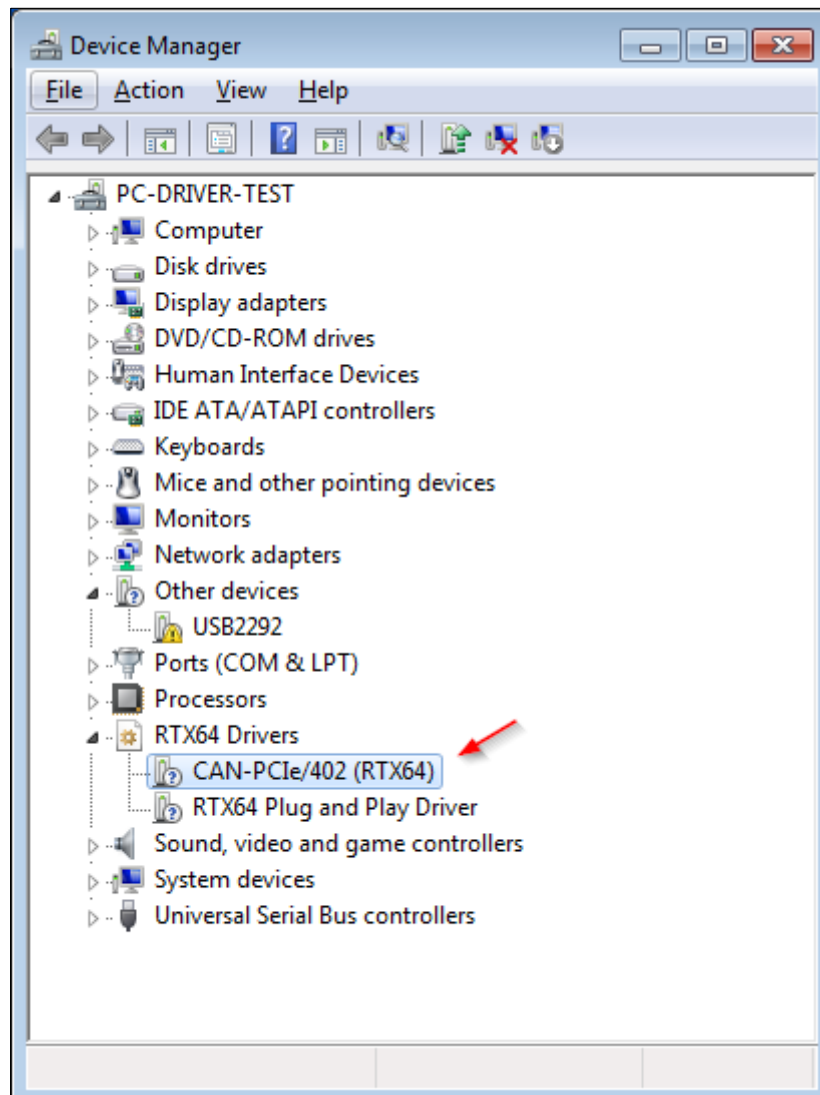


Note:

If you activate the check box "Always trust software from *esd electronic system design gmbh*" you will not have to confirm this dialogue in the future during the installation of another digitally signed driver for an **esd** device.

Press the **Install** button to continue.

If device conversion to RTX64 completes successfully, the CAN interface has to appear under *RTX64 Drivers* in the Device manager.



If a Windows driver is already installed for this device, it is possible that Windows prevents migration to RTX/RTX64 with an indication that "The Best Driver is already Installed" for this device. Refer to chapter 2.5.4 to overcome this situation.

4.3.2.2 MaxRT wRTOS

The 64-bit versions of Windows only accept digitally signed drivers. For this reason, the wRTOS device driver package comes with an INF file and a related CAT file which will allow to install a required *wRtosPNP* driver for the CAN interface. The steps to this are like the installation process of a standard Windows device driver described in chapter 2 for the various (wRTOS supported) versions of Windows.

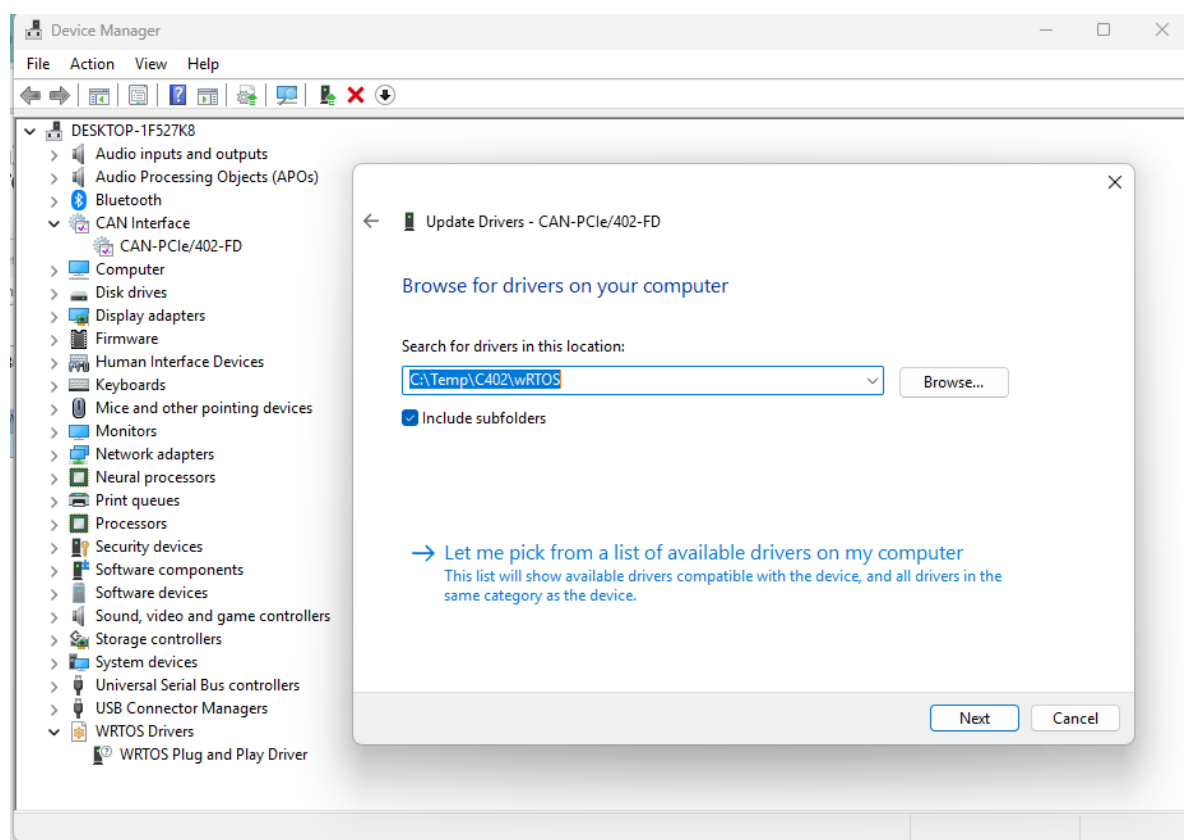
For a general detailed instructions how to convert a device to wRTOS please refer to the section *Converting a Device from Windows to wRTOS* in the *MaxRT wRTOS (Online) Help*.



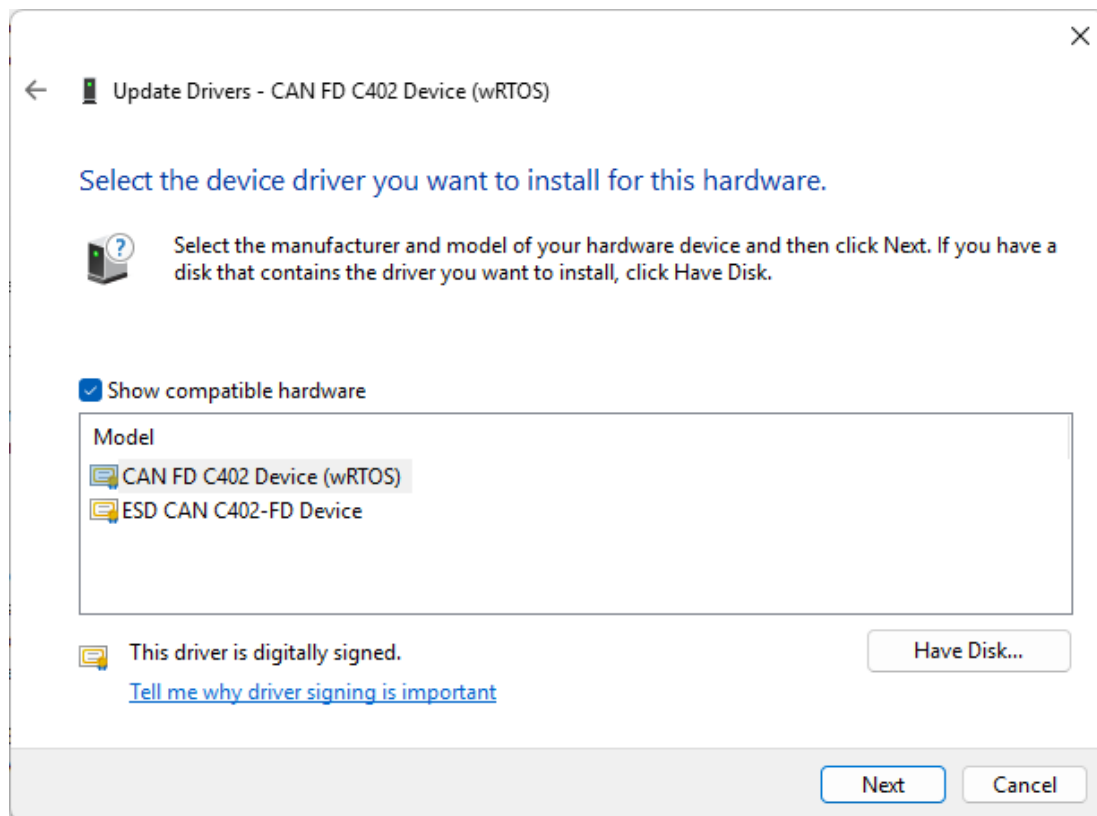
Attention!

Before you can start converting the CAN device from Windows to wRTOS you must copy the driver ***WRTOSpnp.sys*** into a folder which contains the files ***esdwrtos.inf*** and ***esdwrtos.cat*** that are part of the CAN driver package. You will find the driver ***WRTOSpnp.sys*** in your wRTOS runtime installation directory in the folder `MaxRT\wRTOS\inf\`

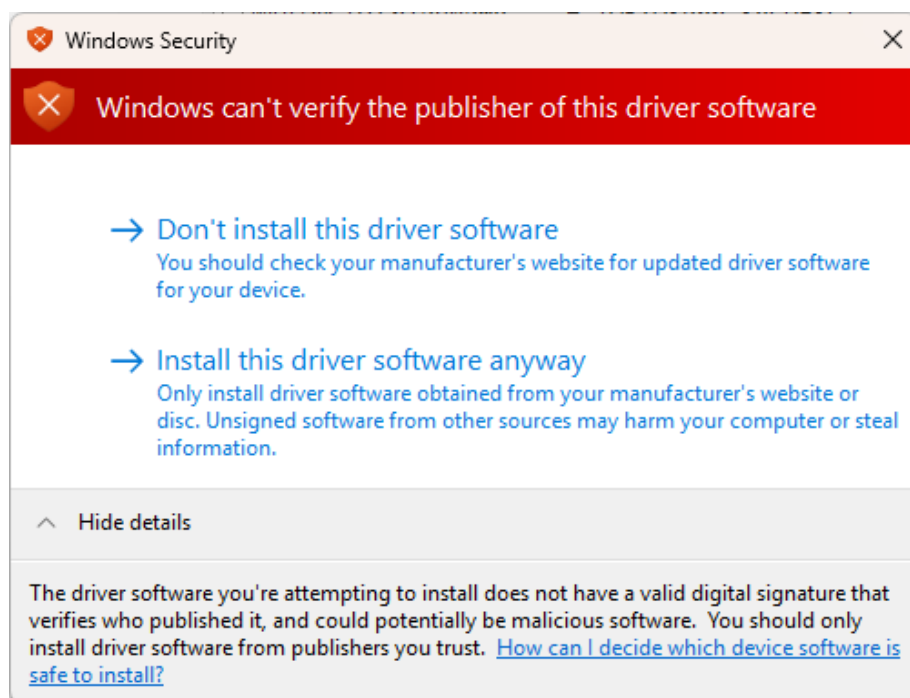
You must use the *Windows Device Manager* to assign the *wrtosPNP* driver to the CAN interface. In the *Device Manager* windows there will be a device under *Other Devices* with a yellow exclamation point next to the icon (if there is no device driver installed yet). The text next to the device will depend on the CAN module attached or the device is listed under CAN Interfaces if a Windows driver is already installed. Right-click on the device and select *Update Driver* from the context menu. A dialog will open where you can choose the directory with the INF/CAT file which comes with the MaxRT wRTOS driver package (see picture below):



Follow the installation instructions and select the driver which description ends with (wRTOS) if several device drivers are listed.



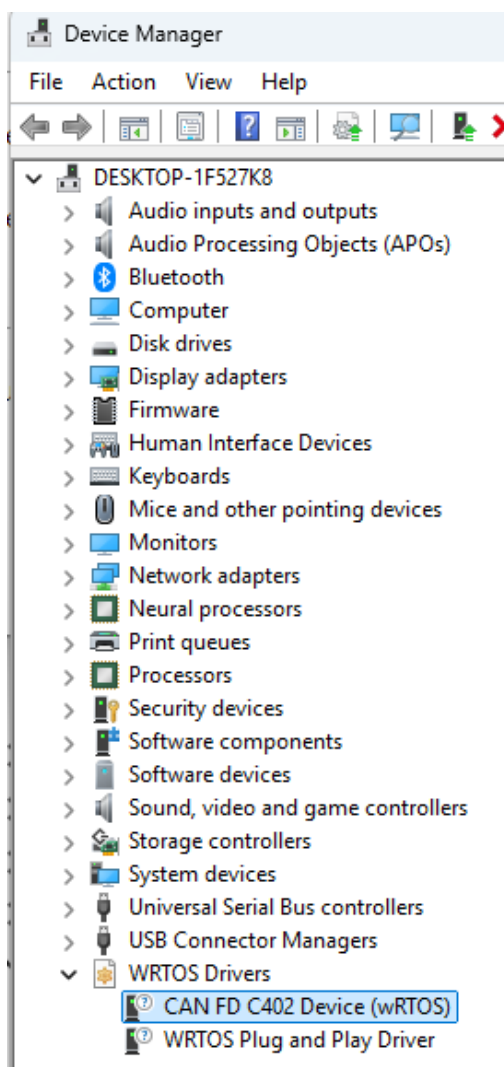
Unfortunately, Windows will present you the following alarming security warning



**Attention!**

This behavior is caused by the current Microsoft signing policy and cannot be solved by *esd electronics gmbh*. The reason is that the required device driver is released by *IntervalZero*, so it is not possible to create a digital signature with an *esd electronics gmbh* certificate. *IntervalZero* signs the INF files of 3rd party hardware with their certificate but the WRTOSpnp.sys driver version might differ from the one you copied into the installation folder. As obtaining a digital signature from Microsoft requires that the INF file contains a driver copy directive (which was not required in earlier days) there exist currently no workaround to prevent the dialog box.

To finalize the conversion process, choose “Install this driver software anyway”. If device conversion to wRTOS completes successfully, the CAN interface has to appear under *WRTOS Drivers* in the Device Manager.



If a Windows driver is already installed for this device, it is possible that Windows prevents migration to RTX/RTX64 with an indication that “The Best Driver is already Installed” for this device. Refer to chapter 2.5.4 to overcome this situation.

4.3.3 Driver Start

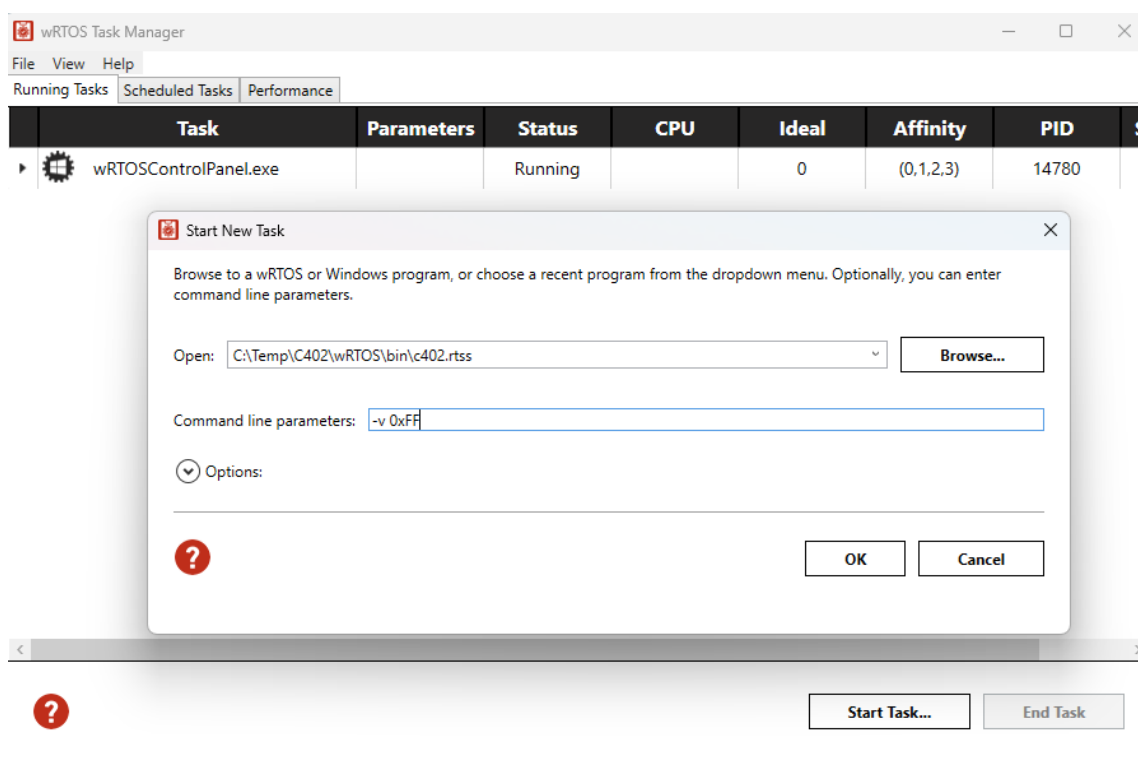
The driver is loaded into the RTSS environment and started with the console or the GUI version of the RTX64/wRTOS tool *RTSSrun*. Type in a Windows console the following command:

```
rtssrun driver.rtss [Arguments]
```

where `driver.rtss` must be replaced with the name of the driver (e.g. `c402.rtss`) and `Arguments` contain the device driver configuration parameter described in the next chapter.

If an error occurs at start, the numeric error code is displayed in the RTX64/wRTOS console and the RTSS- process is terminated. In this case the driver can be started with the option `-v` followed by a numerical hexadecimal value to increase the verbosity of the debug trace messages reporting initialization errors.

With the task manager this may also be performed like this:



4.3.4 Driver Configuration

The RTX64/wRTOS device drivers are configured via the *RTSSrun* command.

4.3.4.1 Device Driver Options

The RTX64/wRTOS device driver's behaviour can be adapted with the options listed in the table below:

Option	Argument	Description
-a	<i>core, core, ...</i>	- This option configures the affinity assignment of the IRQ/IST for the 1st, 2nd, ... CAN interface to the 1st, 2nd, ... CPU core number. - The default behaviour is to run the IRQ/IST on the core the device driver is running during initialization.
-A	<i>core, core</i>	- Assign the affinity of the thread which exchanges data with the CAN hardware to the CPU core number given as first argument. - Assign the affinity of the thread which exchanges data with the application to the CPU core number given as second argument. - If just the 1st argument is provided, both threads are assigned to this CPU core. - The default behaviour is to run the threads on the core the device driver is running during initialization.
-c	<i>N</i>	Start the device driver instance just for the n-th CAN interface (see chapter xxx for details)
-h	-	Display a list of all possible command line options and terminate afterwards.
-k	-	Force clean unload of a resident driver (after closing all applications).
-m	<i>mode</i>	Mode flags (default value = 0x00000000) 0x00000040: Enforce Listen-Only mode on all CAN ports 0x00000080: Disconnect from bus after last handle is closed 0x00800000: Prefer 64-bit DMA memory over 32-bit DMA memory Note: All other bits are reserved and must be set to 0.
-n	<i>net number</i>	This option configures the logical base net number which is assigned to the first physical port of the CAN interface. The logical net numbers are incremented by one for each additional physical port. The default value for <i>net number</i> is 0.
-p	<i>prio</i>	This option configures the priority of the IST thread that processes the CAN messages. The default value for <i>prio</i> is 127.
-V	-	After the start the driver displays driver version information in the RTX64 console and terminates afterwards.
-v	<i>mask</i>	Display additional (debug) messages during startup in the RTX64 console. The mask parameter may be given as a decimal value or as a hexadecimal value prepended by a '0x'
-w	<i>time in ms</i>	Time window in ms for the HW TX operation mode. The default value is 10 ms.

Table 16: Driver Options of RTX64/wRTOS Driver

Example:

The command

```
rtssrun c402.rtss -v0xFF -n2 -p125
```

- Starts the C402 family device driver with additional (debug) output
- Assigns the logical base-net number 2
- Configures an IST (interrupt service thread) priority of 125.

If a driver is started without any command line options the logical net number 0 is assigned to the first physical CAN port, all IST threads get a priority of 127 and the messages shown in the RTX64/wRTOS console are reduced to a minimum.

4.3.4.2 Priority Layout

The active CAN device driver starts several threads as shown in the picture below for a configuration with two physical CAN boards.

The screenshot shows the RTX64 Task Manager window. The 'Running Tasks' tab is active, displaying a table of tasks. The task 'c402.rtss' is selected, and its details are shown below the table. The 'Threads' section lists five threads with their TID, CPU usage, Affinity, and Status. Red arrows point to specific threads with labels: 'Start-Up and Termination Thread' (TID 52), 'Interrupt Service Threads (IST)' (TID 53, 57, 58, 59), 'Communication Thread Driver <-> CAN Hardware' (TID 58), and 'Communication Thread Driver <-> Application' (TID 59).

Task	Status	CPU	Ideal	Affinity	PID	Scheduled
Windows Driver Linked to RTK 1	Running		2	(0,2)	1	
RTX64ControlPanel.exe	Running			(0,1)	3616	
c402.rtss	Running	0.02%	2	(2,3)	1009	

File Location
c:\ESD\candrv\trunk\esdcanMake\rtx64\vs2019\RTX64_CAN\X64\Release\c402.rtss

Local Memory
MSpace used: 1311168 of 2105344 bytes
MSpace auto expand is enabled

Threads

TID	CPU	Affinity	Status
52	0.00%	2	Running
53	0.01%	2	Running
57	0.01%	2	Running
58	0.00%	2	Running
59	0.00%	2	Running

Start Task... End Task

Figure 3: RTX64/wRTOS Device Driver Priority Layout

RTX64/wRTOS scheduling allows RTSS threads to execute at any one of 128 real-time priorities, 0 to 127, where 127 is highest priority and 0 is the lowest. By default, all RTSS threads are created with a priority of 0. Some of the priorities might be adapted via driver options described in chapter 4.3.4.1 if the assigned default priorities do not fit into your priority layout.

Thread	Priority	Description
Main	0	The priority of the initial thread created by the RTSS scheduler. After starting the device driver, this thread is just responsible to handle the driver stop.
IST	124 to 127	One Interrupt Service Thread (IST) for each active physical CAN board. The default priority of these threads is 127 and can be overridden with the device driver option '-p' (see Table 16).
DPC	123	CAN driver thread (common for all CAN cards handled by this device driver) to exchange data with the CAN hardware.
DISP	121	CAN driver thread (common for all CAN cards handled by this device driver) to exchange data with the application.

Table 17: RTX64 Device Driver Thread Priorities

As a rule, you should always keep the following numerical priority relation:

IST > DPC > DISP > RTSS

with RTSS as priority of the threads which perform NTCAN API calls.

4.3.4.3 64-Bit DMA memory

The default behaviour of the device driver is to request DMA memory from the host memory area below 4 GB. If the CAN hardware supports 64-Bit DMA addresses and the host has a memory area above 4 GB, it is possible to enforce the driver to request DMA memory from this area.

Even with the availability of physical memory ≥ 4 GB for DMA the device driver will always try to request memory from the area < 4 GB first. Only if this fails, the driver will try to request memory from an area ≥ 4 GB. To invert this proceeding you can start the driver with a respective flag of the mode parameter (see chapter 4.3.4.1).



Note:

The use of DMA memory in an area ≥ 4 GB increases each PCIe packet by an additional four bytes for the address, which may lead to a small drop in performance.

4.3.4.4 Multi Core Support

If RTX64/wRTOS is assigned more than one physical core on a multi-core system, the IRQ and the IST of the driver are handled on the processor the process runs on during initialization. You can influence this with the *RTSSrun* utility to optimize your overall system performance by using the '/a' parameter to configure the affinity and/or using the '/p' parameter to define an ideal processor (see RTX runtime help).

Example:

The command starts the C402 family device driver as in the example above but forces IRQ/IST execution to the processor with the system-wide processor number 2.

```
rtssrun /a 2 c402.rtss -v 0xFF
```

If the device driver handles more than one CAN interface on a multi-core system you can assign the core to IRQ/IST handling with the device driver option '-a' and the core to handle the DPC and/or DISP thread with the device driver option "-A" (see **Table 16**).

Example:

The command starts the C402 family device driver as in the example above but forces DPC/DISP thread execution to the processor with the system-wide processor number 3.

```
rtssrun /a 2 c402.rtss -v 0xFF -A3
```

For a system with many CPU cores assigned to RTX64/wRTOS and many CAN cards handled by the same device driver the common DPC and DISP thread might become a bottleneck if handled by just one core. In this case you can overcome this limit and run several instances of the identical device driver in parallel using the device driver option '-c'. Each device driver instance handles just one CAN card using individual DPC/DISP threads assigned to an individual CPU core.

Example:

The commands below start an individual instance of the C402 family device driver for the first CAN card in the system on CPU core 2 and the second CAN card in the system on CPU core 3. As each driver instance starts assigning logical net numbers from 0 you also must provide a different net number base for the 2nd instance to prevent overlapping (logical) CAN net numbers.

```
rtssrun /a 2 c402.rtss -v 0xFF -c 1 -n 0
```

```
rtssrun /a 3 c402.rtss -v 0xFF -c 2 -n 4
```

4.3.5 Driver Unload

A driver must be gracefully stopped (unloaded) by running the driver binary with the driver parameter option '-k' with the following command line in a Windows console window

```
rtssrun driver.rtss -k
```

or if the driver was started for just one card with the following command line:

```
rtssrun driver.rtss -k -c number
```

with *number* as respective card instance.

**Attention!**

You must never terminate a driver calling the RTX64/wRTOS tool **rtsskill** or the task manager for its RTSS process as this might result into severe system instabilities. Due to this forced termination RTX64/wRTOS will free and maybe reuse the processes memory resources which are still be used as DMA memory by the CAN hardware which is not aware of this.

4.3.6 Miscellaneous

This chapter covers several topics using the NTCAN architecture on RTX64/wRTOS.

4.3.6.1 Application Development

To develop your own NTCAN based applications with Microsoft® Visual Studio the driver package contains the header `ntcan.h` and the library file `ntcan_rtss.lib`. To compile for RTX64/wRTOS, you must define `UNDER_RTSS` before the header is included.

You can also start developing your application in the Win32 non-real-time environment before you change into the real-time RTSS environment to use the more comfortable possibilities to debug your application and take advantage of the **esd** CAN tools which just support the Windows environment. In this case you must install the Windows device driver, and the CAN SDK as described in chapter 2. Use the header `ntcan.h` which comes with your RTX driver package (to prevent implementing features which are already available on Windows but not yet available on RTX64/wRTOS) but do not define `UNDER_RTSS` before you include the header.



Note:

The NTCAN API on Windows and RTX64/wRTOS is identical with the exception that overlapped operations are not supported in the RTSS environment.

4.3.6.2 Example Application

The driver package comes with the RTSS application `cantest_rtss` which can be loaded into the RTSS environment in the same way as described for the NTCAN library. With the help of this program, you can do basic functionality checks of the CAN interface. The program and its parameters are described in more detail in [11](#).

4.4 TenAsys® INtime®

This chapter covers the necessary steps to install, configure and start the device drivers for **esd** CAN boards supporting the real time operating system *INtime®* of *TenAsys®*

INtime® is a hard real-time, event-driven OS for the x86 architecture which can operate stand-alone (*INtime dustributed RTOS*) or side-by-side with Windows (*INtime for Windows*). The RT kernel supports Real-Time Applications (RTAs) and Real-Time Shared Libraries (RSLs).

A device driver package for INtime® contains the following files where **<drvname>** is the device family specific driver name following driver naming convention I (see chapter 1.4) and **<ver>** is an INtime major version.

Filename	Description
<code>bin/<ver>/<drvname>.rta</code>	The hardware specific CAN device driver.
<code>bin/<ver>/ntcan.rsl</code>	The NTCAN library
<code>bin/<ver>/cantest.rta</code>	The <i>cantest</i> application
<code>doc/</code>	API documentation, release notes, etc.
<code>include/ntcan.h</code>	NTCAN header to compile the NTCAN based application.
<code>lib/vc/ntcan.lib</code>	Lib file for Visual Studio to link the NTCAN based application
<code>samples/cantest.c</code>	Source code of the <i>cantest</i> application

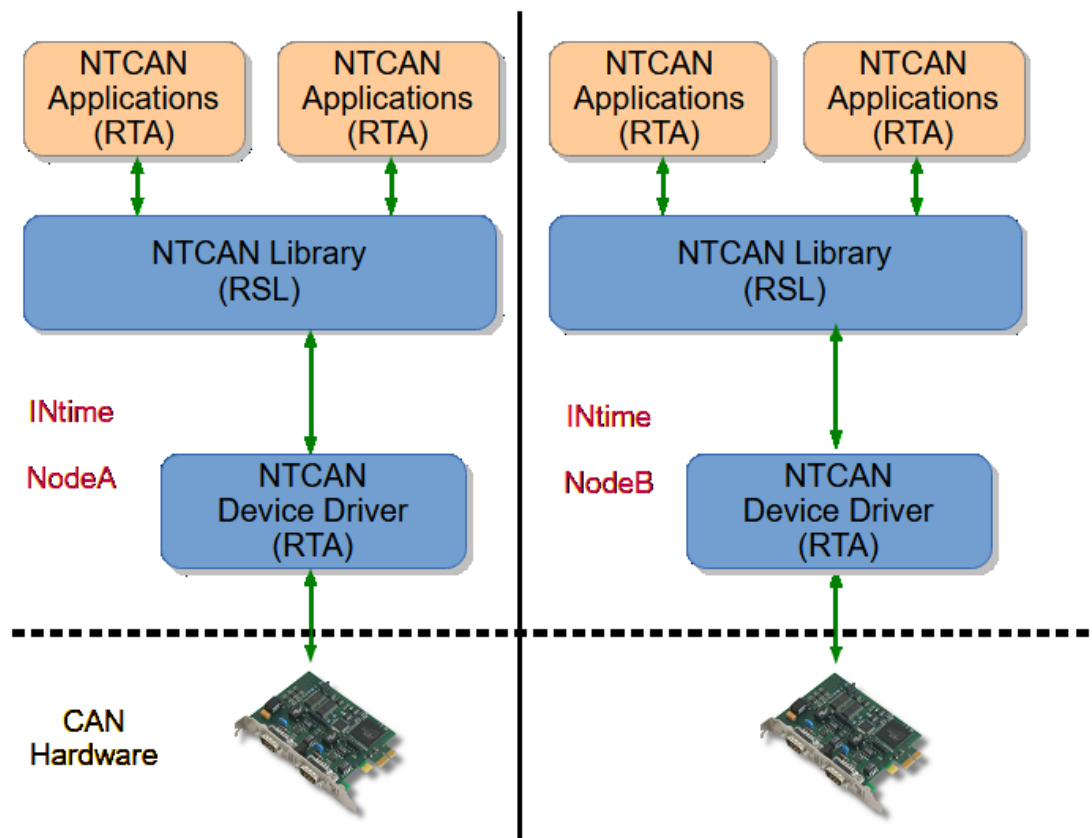
Table 18: Files of Intime® CAN driver package

4.4.1 Driver Integration

The support for an **esd** CAN interface consists of a device driver (cxxx.rta) and the NTCAN library (ntcan.rsl) as shown in the picture below. The hardware specific device driver (which usually covers a complete CAN device family) is implemented as an INtime® Real-Time Application (RTA) which creates a resident service after the successful start. The driver must be started before any other RTA which links explicitly or implicitly to the NTCAN library implemented as an INtime® Real-time Shared Library (RSL).

This approach allows using different CAN driver in the INtime® environment with the identical API and allows several INtime® RTAs using the same CAN hardware as well as one INtime® RTA using different CAN hardware.

INtime for Windows or INtime Distributed RTOS



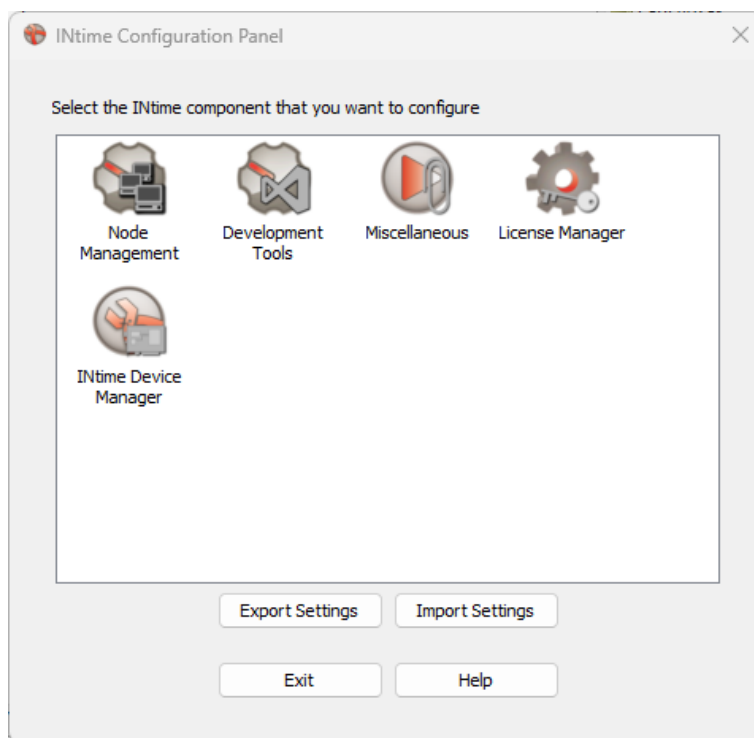
An RTA which uses the CAN hardware must always use the entries exported by the NTCAN RSL described in [/1/](#) and must communicate directly with the device driver.

4.4.2 Driver Installation

To support an **esd** CAN interface in the INtime® environment it is necessary to remove the control from convert it into a RTX managed device as described below (for further details refer to the *INtime SDK Documentation*).

4.4.2.1 INtime for Windows

Without additional measures Windows gets control on the CAN hardware and tries to install an associated device drivers as necessary. This is undesired when a device must be controlled from within the INtime environment.

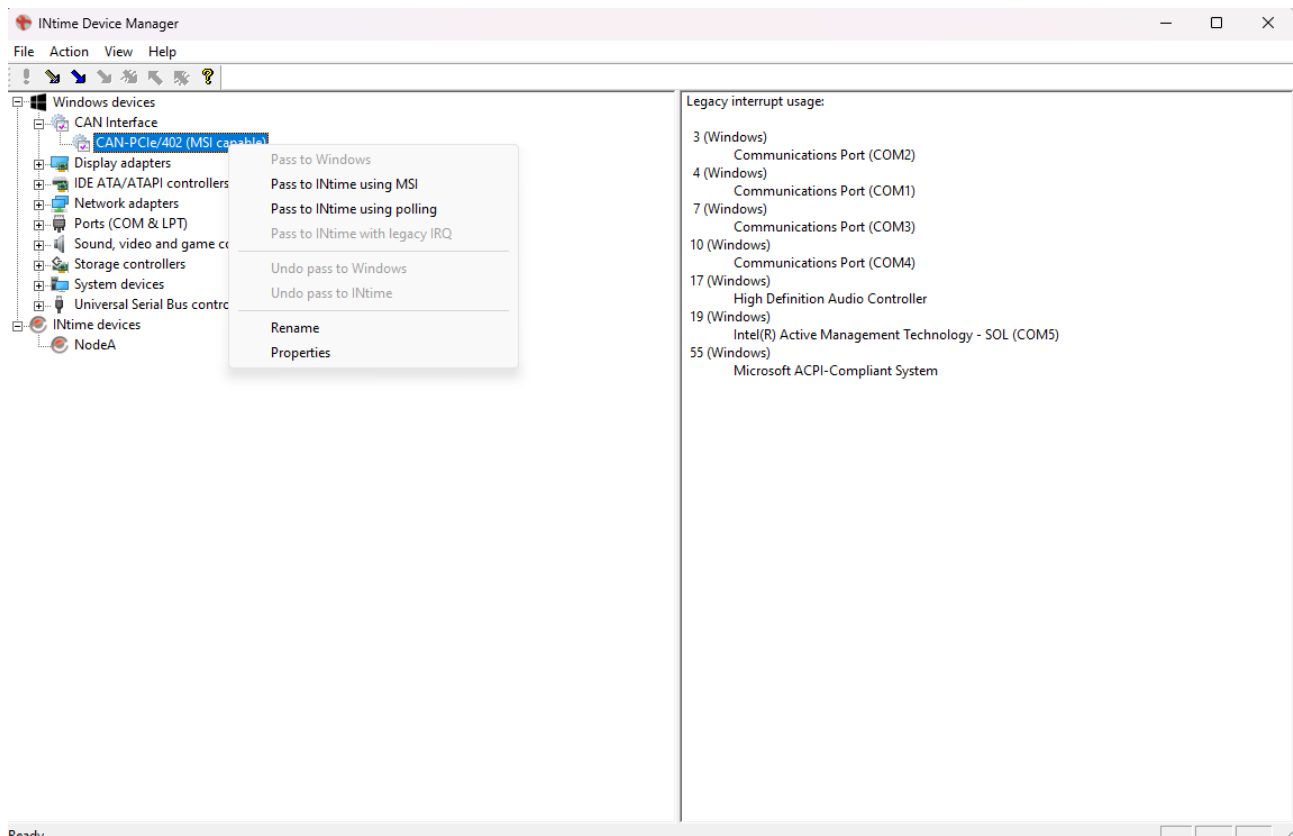


You must use the *INtime Device Manager* to pass control of plug-and-play devices between Windows and a local INtime node. This utility assigns a device to INtime use by installing a simple device driver on the requested device. The device driver does two things:

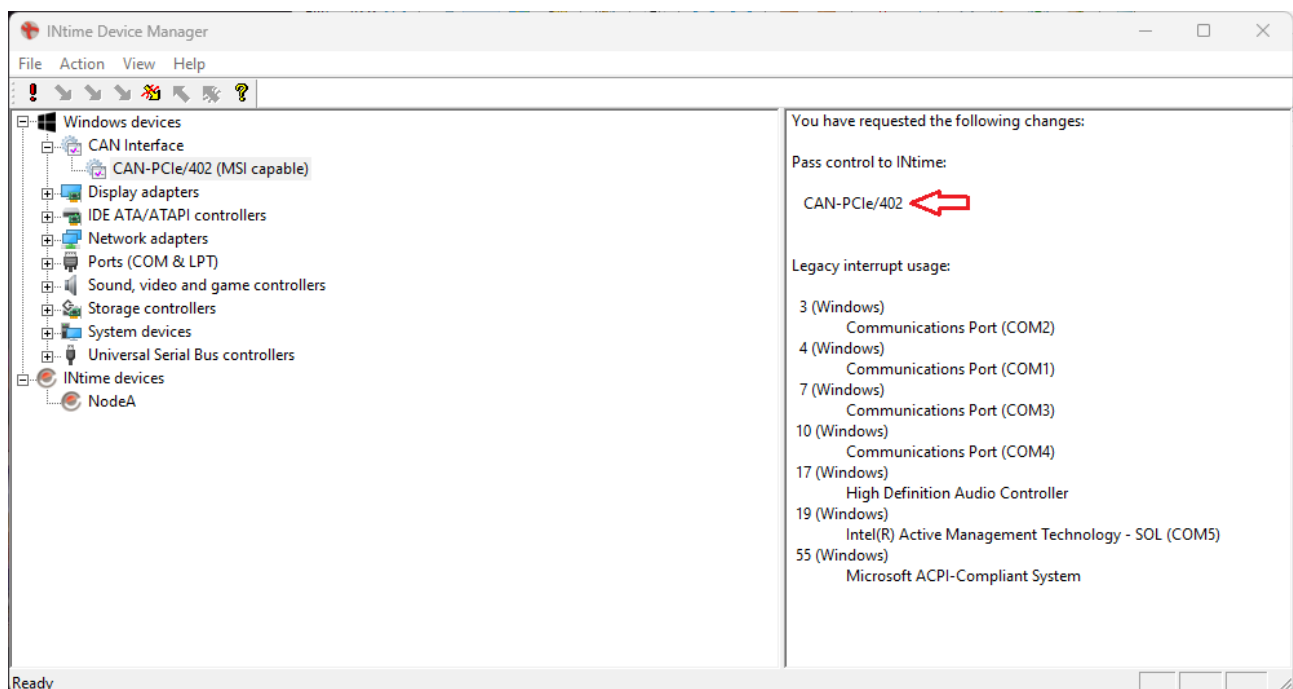
- It satisfies Windows plug-and-play management to stop it trying to install a normal Windows device driver, and
- It signals to the INtime node that it "owns" the device.

Real-Time Operating Systems

Open the *INtime Device Manager* and open the right-click context menu of the CAN interface which control you will pass from Windows to INtime.

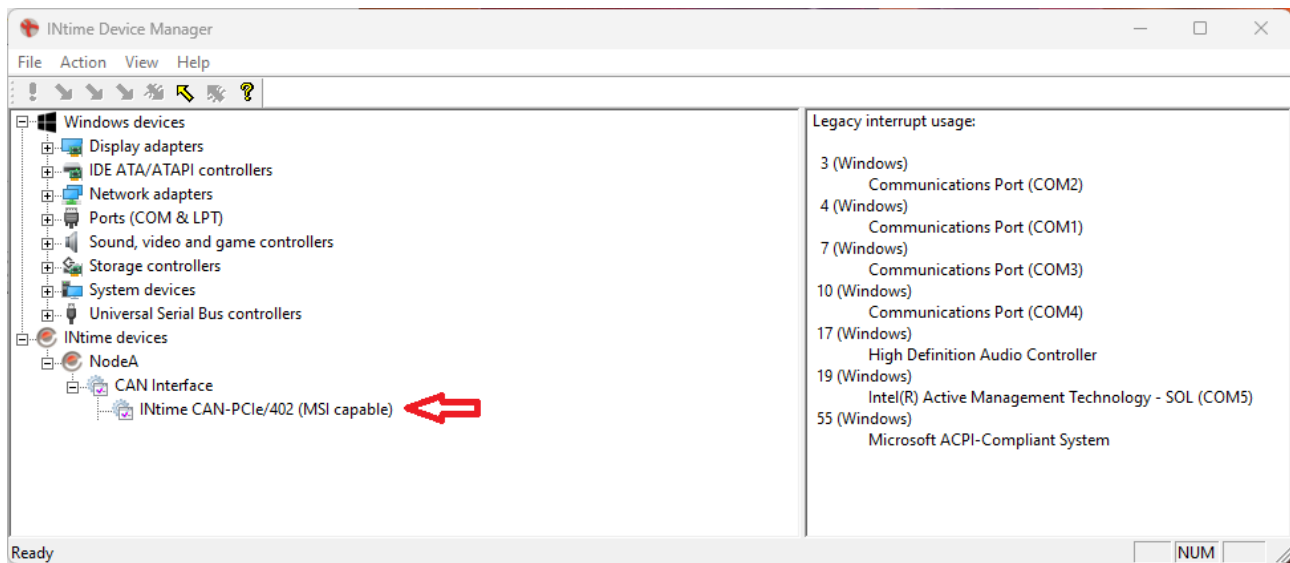


Select “Pass to INtime using MSI” to prepare the use of the CAN hardware in the INtime environment.

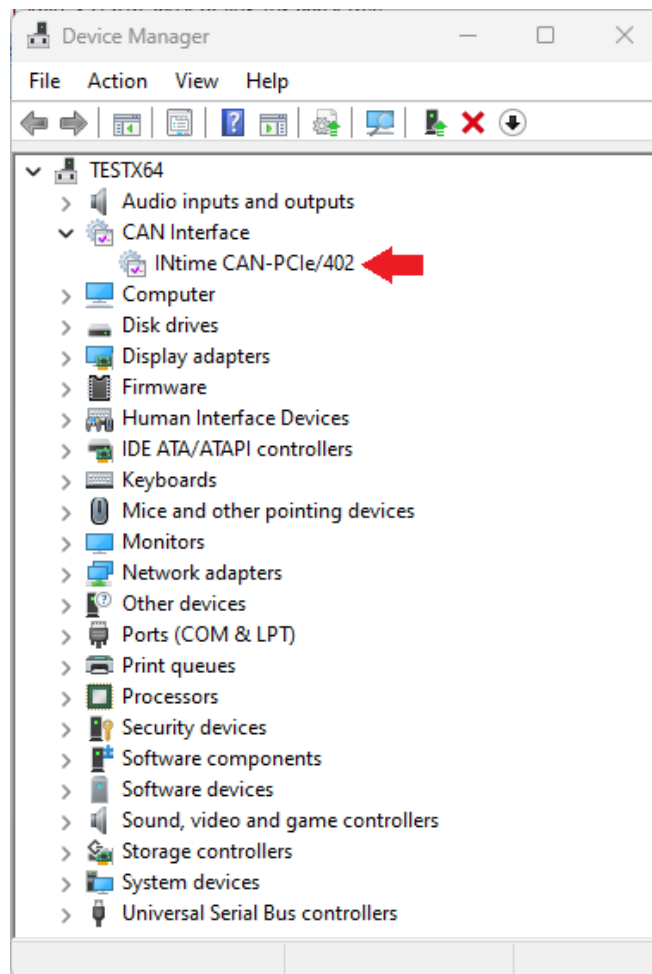


If you exit the *INtime Device Manager*, you will be asked to finalize the requested changes.

After the changes are applied the *INtime Device Manager* will indicate that the CAN device is assigned to an INtime node.



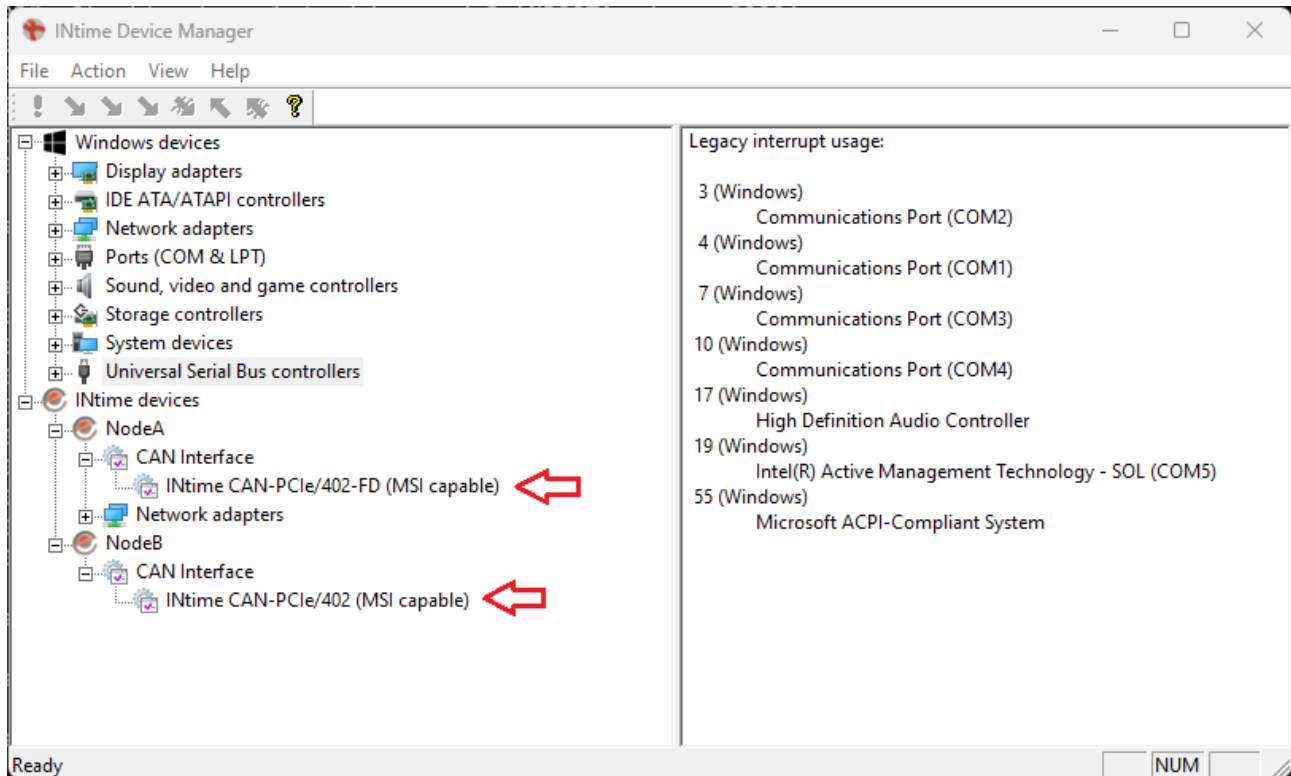
The *Windows Device Manager* will indicate that the device is assigned to INtime.





The INtime CAN device driver is currently limited to supporting one physical CAN board per INtime node. If you need more physical CAN ports available on a single CAN board you must create additional INtime nodes to assign them additional CAN boards.

The picture below shows a configuration with two CAN boards assigned to different INtime nodes. You must start the device driver as described in the next chapter for each node.



4.4.3 Driver Start

The INtime kernel is designed to support dynamical loading of executable code. This involves three components:

- **INtime Load Server:** A service within the INtime kernel which handles load requests. The load server manages the loading of both application (RTA) and dynamic libraries (RSL).
- **Windows Load Client:** A Windows application (*ldrta.exe*) which interacts with the load server to load a .RTA file and to create new process with it.
- **INtime Load Client:** An INtime kernel service which allows an INtime application to interact directly with the load server.

This architecture is identical wheter the kernel runs on an INtime node of INtime for Windows or INtime Distributed RTOS.

The driver is loaded with ldrta.exe executing the following command line in a Windows console window:

```
ldrta [-node nodename] [-a "Arguments"] driver.rta
```

where `driver.rta` must be replaced with the name of the driver RTA (e.g. `c402.rta`) and `Arguments` contain the device driver configuration parameter described in chapter 4.4.4.1. For configurations with support for more than one node you also must provide the INtime node name the respective CAN interface is assigned.

After a successful start the root process of the device driver is cataloged with the respective (family) name and is ready to be used by applications (via the NTCAN RSL). In case of any error during the startup a Windows console window is opened to show an error message. In the latter case the driver can be started with the option `'-v'` followed by a numerical hexadecimal value to increase the verbosity of the debug trace messages reporting initialization errors.

4.4.4 Driver Configuration

The INtime® device drivers are configured with parameters to the *ldrta.exe* command.

4.4.4.1 Command Line Parameter

The INtime® device drivers are configured via options listed in the table below which are passed as arguments to the driver start command described in the previous chapter:

Option	Argument	Description
-h	-	Display a list of all possible command line options in the Intime® console window and terminate afterwards.
-m	<i>mode</i>	Mode flags (Default value = 0x00000000) 0x00000040: Enforce Listen-Only mode on all CAN ports 0x00000080: Disconnect from bus after last handle is closed 0x00800000: Prefer 64-bit DMA memory over 32-bit DMA memory. Note: All other bits are reserved and must be set to 0.
-n	<i>net number</i>	This option configures the logical base net number which is assigned to the first physical port of the CAN interface. The logical net numbers are incremented by one for each additional physical port. The default value for <i>net number</i> is 0.
-p	<i>prio</i>	This option configures the priority of the DPC thread that processes the CAN messages. The default value for <i>prio</i> is 135.
-P	<i>prio</i>	This option configures the priority of the dispatch thread that handles requests from the applications. The default value for <i>prio</i> is 140.
-V	-	After the start the driver displays driver version information in the Intime® console window and terminates afterwards.
-v	<i>mask</i>	Display additional (debug) messages during startup in the INtime® console window. The mask parameter may be given as a decimal value or as a hexadecimal value prepended by a '0x'
-w	<i>time in ms</i>	Time window in ms for the HW TX operation mode. The default value is 10 ms.

Table 19: Command Line Parameter of INtime® Driver

Example:

The command

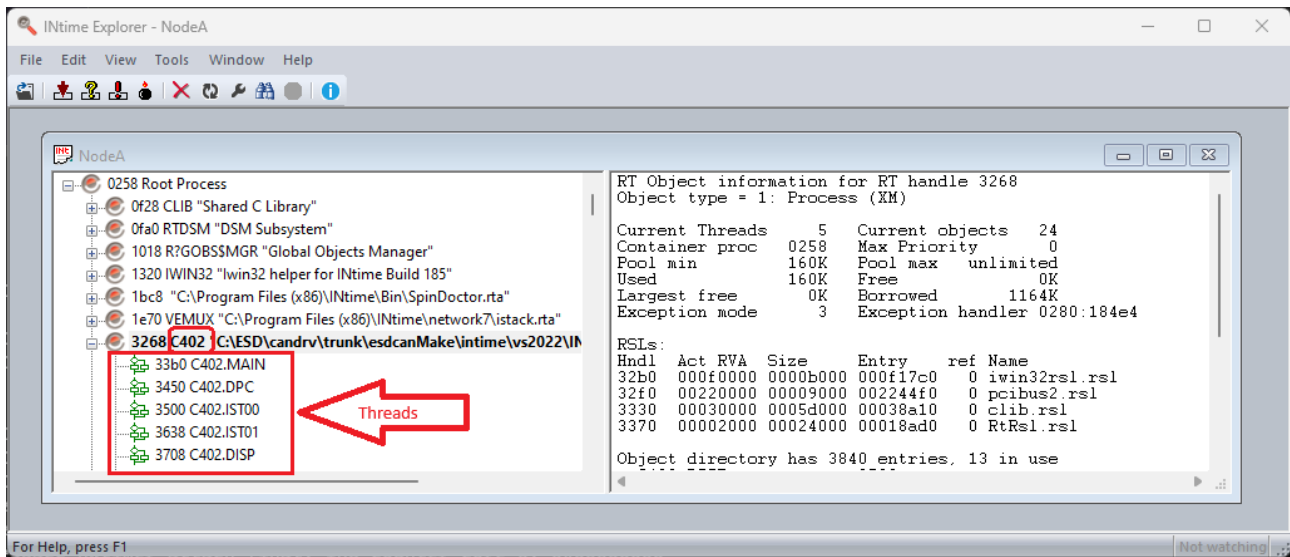
```
ldrta -a "-v0xFF -n2 -p130" c402.rta
```

- Starts the C402 device driver with additional (debug) output.
- Assigns the logical base-net number 2.
- Configures the DPC priority to 130.

If a driver is started without any command line options, the default values according to the table above are applied.

4.4.4.2 Priority Layout

The active CAN device driver starts several threads as shown in the picture below for a configuration with two physical CAN boards.



Some of the priorities might be adapted via the command line as described in the previous chapter if the assigned default priorities do not fit into your priority layout. The threads are catalogued with thread names starting with the device driver family name followed by a '.' as described in the table below:

Thread name	Priority	Description
DRV.MAIN	155	The priority of the initial thread created by the INtime load server. Refer to <i>Loading INtime Applications</i> in the <i>INtime SDK</i> for details on how this value can be adapted.
DRV.ISTxx	N/A	One Interrupt Service Thread (IST) for each active physical CAN board. The priority of these threads is assigned by the INtime kernel dynamically in the range from 0 – 130.
DRV.DPC	135	Common CAN driver event handler which priority can be adapted at load time.
DRV.DISP	140	Common CAN driver application message dispatch handler which priority can be adapted at load time.

As a rule, you should always keep the following numerical priority relation:

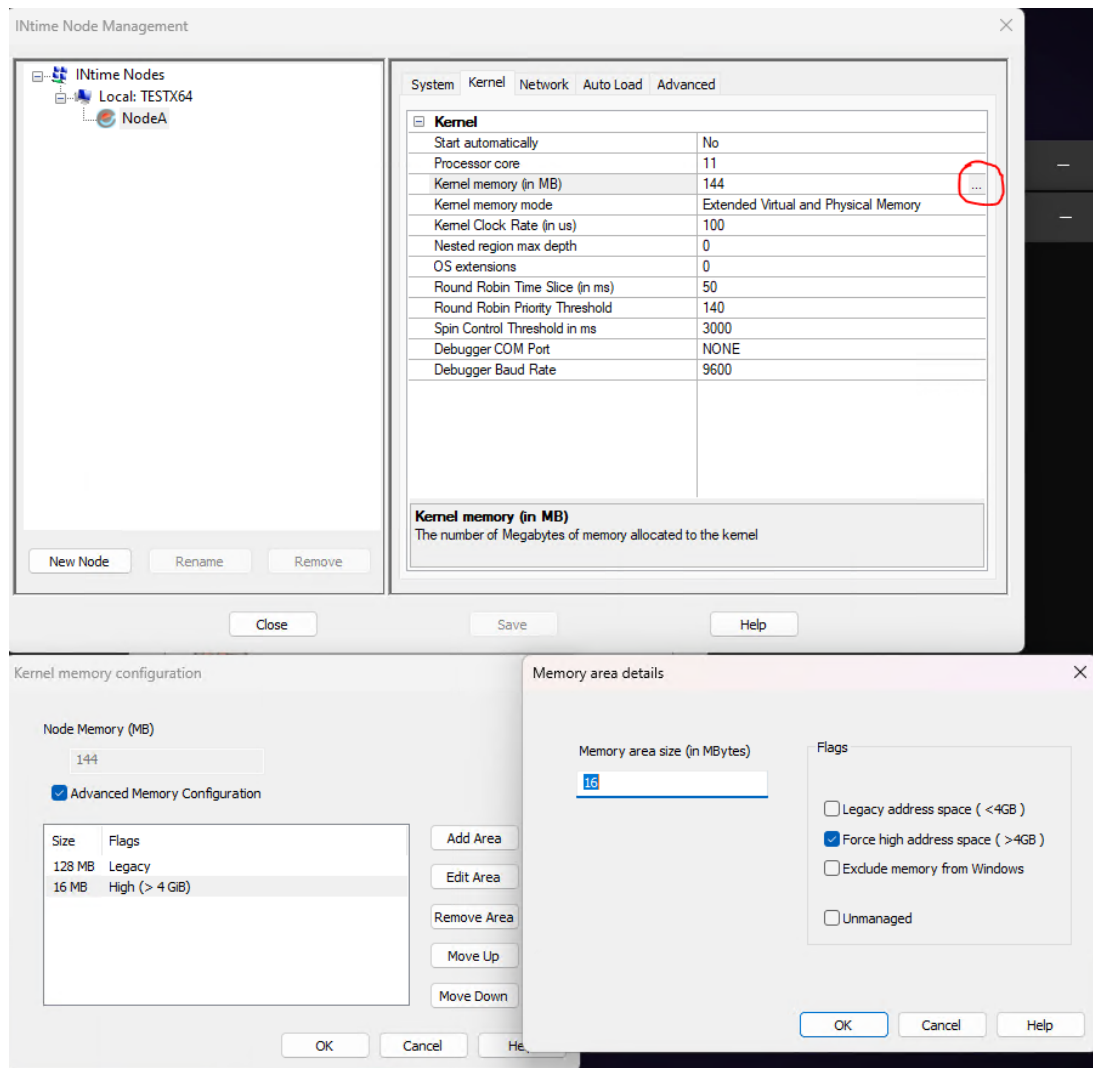
$$\text{DRV.ISTxx} < \text{DRV.DPC} < \text{DRV.DISP} < \text{RTA}$$

with RTA as priority of the threads which perform NTCAN API calls.

4.4.4.3 64-Bit DMA memory

The default behaviour of the device driver is to request DMA memory from the host memory area below 4 GB. If the CAN hardware supports 64-Bit DMA addresses and the INtime node is assigned a memory area above 4 GB it is possible to enforce the driver to request DMA memory from this area.

The assignment of memory above 4GB (if physically available on the host) must be configured using the *INtime Node Management* as shown in the picture below:



Even with the availability of physical memory ≥ 4 GB for DMA the device driver will always try to request memory from the area < 4 GB first. Only if this fails, the driver will try to request memory from an area ≥ 4 GB. To invert this proceeding you can start the driver with bit 23 set in the *mode* parameter.



Note:

The use of DMA memory in an area ≥ 4 GB increases each PCIe packet by an additional four bytes for the address, which may lead to a small drop in performance.

4.4.5 Driver Unload

A driver can be gracefully stopped (unloaded) with the INtime tool **killrta.exe** executing the following command line in a Windows console window:

```
killrta [-node nodename] [-name rtaname]
```

or alternatively

```
killrta [-node nodename] [-proc handle]
```

with `rtaname` as cataloged driver name or `handle` as process Id you might figure out with the *INtime Explorer*.

4.4.6 Miscellaneous

This chapter covers several topics using the NTCAN architecture on INtime®.

4.4.6.1 Application Development

To develop a NTCAN based applications with Microsoft® Visual Studio the driver package contains the header `ntcan.h` and the library file `ntcan.lib`. To compile for INtime® you must define `__INTIME__` before the header is included. The latter is already defined if you start your project with the Visual Studio INtime wizard to create an RTA.



Note:

The NTCAN API on Windows and INtime® is identical with the exception that overlapped operations are not supported in the INtime® environment.

4.4.6.2 Example Application

The driver package comes with the INtime® application `cantest.rta` which can be loaded into the INtime® environment with the *RT Application Loader* (`ldrta.exe`) or the *PipeRTA* (`piperta.exe`) tools. Output is displayed in the respective console windows. With the help of this program, you can do basic functionality checks of the CAN interface. The program and its parameters are described in more detail in *I/1* and the source code of this example application is part of the driver package.

Please refer to the chapter *Loading INtime Applications* of the *INtime SDK* for a detailed description of loading RTAs. Any NTCAN API based application requires to link to the NTCAN API RSL which must be accessible by the INtime Load Server e.g. by keeping RTA and RTS in the same folder.



Note:

Refer to the *INtime SDK Help* for the search paths which are used by the RT kernel to load RTAs with implicitly or explicitly linked RSLs.

5 Firmware Update Application

Some of the active **esd** CAN boards store their firmware in a NVRAM which is in most cases updatable by the end user. A firmware update to a different version than the one the CAN board is shipped with might be necessary to support new features or to troubleshoot problems. The task of performing the firmware update is handled by a console application which is described in chapter 5.1.

An additional task which is also covered by the same application is switching between active and passive support for CAN messages in the extended frame format (29-bit CAN-IDs) for a certain family of active CAN boards (see chapter 5.2).

The firmware update applications are usually available for Windows and in many cases also as native versions for operating systems with shell support (Linux, LynxOS, QNX, VxWorks, ...). If the firmware update tool is not available as native version for your target operating system you must do the update with a supported OS (e.g. Windows).

The table below gives an overview on the updatable **esd** CAN boards with a firmware stored in the NVRAM. The name of the tool always starts with '**upd**' followed by a CAN board and platform specific extension. An update application for active CAN boards which can be switched between active and passive support for 29-bit CAN-IDs are marked with an asterisk (*).

CAN Board	Update Tool Name
CAN-PCI/331, CPCI-CAN/331, PMC-CAN/331	updc331*, upd-pci331* or upd-pci331-i20*
CAN-PCI/360, CPCI-CAN/360	updc360*, upd-pci360 or upd-pci360-i20*
CAN-PCIe/402, CAN-PCI/402, CPCIserial-CAN/402, CAN-PCIMini/402, CPCI-CAN/402	updc402
CAN-PCIe/402_FD, CAN-PCI/402-FD, CPCIserial-CAN/402-FD, CAN-PCIMini/402-FD, PMC-CAN/402-FD, XMC-CAN/402-FD	updc402fd
CAN-USB/Mini	updusb331
CAN-USB/Micro	updcanmicro
CAN-USB/2	updusb2292
CAN-USB/2V2	updusb2v2.exe
CAN-USB/3-FD	updusb3fd.exe
CAN-USB/400, CAN-USB/400-IRIG-B	updusb400
CAN-USB/400-FD	updusb400fd
CAN-AIR/2	updcanair2
CAN-ISA/331, CAN-PC104/331	updc331i*, upd-isa331* or upd-isa331-i20*

Table 20: Overview of firmware update applications

**Note:**

The EtherCAN and EtherCAN/2 boards also have an updatable firmware but as a network attached device the update is also performed via the network and not with the firmware update tool described here. Please refer to /3/ for further details.

Other active CAN boards not mentioned in the table above have the latest firmware in the device driver binary.

5.1 Updating the Firmware

A firmware can be updated with a CAN board and operating system specific console application.

**Attention!**

Even if the updating process is made fail-safe, we do not recommend updating the firmware if the CAN board works without any problems or the update is explicitly recommend/required. For any damages caused by improper operation of updating the firmware **esd** assumes no responsibility.

**Prerequisites:**

For the firmware update the device driver has to be installed and started and you have to make sure that CAN board works properly in your system. You also have to make sure, and no other application is using the CAN board. Otherwise, the update tool will return with an error.

Follow the steps below to update the firmware:

- To start the firmware update you must open a console window and change into the directory with the firmware update tool for your CAN board.

**Note:**

On Windows the update tool requires a console with Administrator Privileges.

If you start the update tool in a standard console without these privileges modern Windows versions will show the UAC prompt and start the tool in a privileged console which will open and close immediately without any further action.

- Type the name of the update tool which covers you CAN hardware (see Table 20) followed by one of the logical net numbers which is assigned to the CAN board you want to update. Below is an example for a CAN-USB/2 which is assigned the logical net number 6 in the system (1).

```
E:\TEMP\driver\usb2292>updbus2292 6 1
updbus2292 - firmware updater (Version: 2.2.7 - built Sep  5 2008 17:07:44)
(C) esd electronic system design gmbh

Available image:
  card-id       = "USB2"
  firmware-version = 1.0.05 2
  firmware-length = 59392(0xE800)

Current image on board:
  card-id       = "USB2"
  firmware-version = 1.1.00 3

Current Host-Driver:
  driver-version  = 2.6.03

||||| WARNING!!! Do not turn off power or reset your machine before the firmware
||||| download is completed! Removing the power before the firmware
||||| update is completed could cause the firmware to be inadvertently
||||| deleted and will render the unit inoperable. If this occurs, the
||||| device will not start up and will require factory service to
||||| reload the firmware.
||||| Update only if you're having problems with the current firmware.
||||| "If it ain't broke, don't fix it."
|||||

The firmware on board is newer! 4
Do you want to continue(y/n)?
```

- The tool will display the version of the firmware the CAN board can be updated to (2) and the active firmware version (3). If the tool can not find a suitable CAN hardware for the given logical net number, it just shows the built-in firmware image.
- You will also see a warning message about the dangers of a firmware update and in case you do a downgrade an information that you change to an older firmware version (4).
- If you answer the question to continue with 'y' the firmware update process is started, and you will be informed about its progress. If you want to cancel now and keep the active firmware answer with 'n'.



Attention!

Once started do not interrupt the update process by aborting the program or resetting the system before the firmware update tool reports that it has updated the firmware successfully.

- After the new firmware is successfully updated an USB based device (with the exception of the CAN-USB/400) will automatically do a reset, the device is re-enumerated by the host system and can be used immediately with the new firmware. CAN boards for other buses require an explicit hardware reset before the new firmware becomes active which can either be performed by rebooting the host system or by loading/unloading the device driver (if this is supported by the host OS).

**Attention!**

For all devices of the **C402 family** (see Table 1) and the **CAN-USB/400** the method described above for the hardware reset is not sufficient to activate the new firmware. The hardware requires a real power cycle where it is disconnected from the power supply for a second.

5.2 Switch between CAN 2.0A and CAN 2.0B Mode

Some **esd** CAN boards (marked in Table 20 with an asterisk) support two different firmware operation modes. One mode (according to CAN 2.0A) can transmit and receive only 11-bit CAN-IDs (passive support of 29-bit CAN-IDs), the other mode (according to CAN 2.0B) can transmit and receive 29-bit as well as 11-bit CAN-IDs. The first version is the factory default, and this chapter describes how you can switch to the other version.

**Note:**

Only the CAN boards marked in Table 20 with an asterisk come with two different firmware versions. All other **esd** CAN boards always support the universal CAN 2.0B mode and you can ignore this chapter if you have one of these other boards.

To switch between the two firmware modes the update application offers in addition to the real firmware update capability described in the previous chapter the options '-ta' to switch into the (factory default) 11-bit mode and '-tb' to switch into the 29-/11-bit mode.

Follow the steps below to switch to another firmware version:

- Open a console window and change into the directory with the firmware update application for your CAN board.
- Type the name of the update application which covers your CAN hardware (see Table 20) followed by the option '-ta' to switch into the CAN 2.0A mode or '-tb' to switch into the CAN 2.0B mode followed by one of the logical net numbers which is assigned to the CAN board.
- The CAN board requires an explicit hardware reset before the new firmware mode becomes active which can either be performed by rebooting the host system or by loading/unloading the device driver (if this is supported by the host OS).

**Note:**

In comparison to the real firmware update described in the previous chapter just changing the firmware operation mode does not contain any risks to make the CAN board unusable.