

ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Janztec / Janz Computer VMOD-ICAN3* at our website: ***Click HERE***

janz Automationssysteme AG

SOLUTIONS FOR AUTOMATION

PROPRIETARY NOTICE

All rights reserved by Janz Automationssysteme AG.

No parts of this technical manual may be modified, copied or reproduced in any form or by any means for commercial use without the prior written permission of Janz Automationssystme AG, Germany.

All instructions, information and specification contained in this manual are for reference only and remain subject to change without announcement.

Janz CAN Tutorial

A step-by-step guide for software developers
describing how to write own applications for
Janz intelligent CAN devices (ICAN)

(Software Tutorial)

Version 4.0

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Title:	This document describes a step-by-step how application can be written for Janz Intellegent CAN (ICAN) devices
File:	g:\devel\driver\dpm\doc\tutorial\janz can tutorial.doc
Pattern:	\\ntserv5\janzgrp\as\entwicklung\formulare\entwicklung (ew)\f_manual.dot
Created:	Christoph Mühlenhoff, 13.06.2005

Last Update: Christoph Mühlenhoff, 22.03.2006

© Janz Automationssystme AG 2006

Im Dörener Feld 8

D-33 100 Paderborn, Germany

Tel.: +49-5251-1550-0

FAX: +49-5251-1550-90

Email: support@janz.de

Internet: http://www.janz.de/as

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Contents

1 Introduction 9

1.1 Purpose of the document 9

1.2 Examples 9

1.3 Document structure 9

1.4 By the way … 10

2 Hardware 11

2.1 Supported hardware UPDATED 11

2.2 Hardware installation 13

2.2.1 PCI boards 14

2.2.2 CompactPCI boards 14

2.2.3 PMC boards 14

2.2.4 ISA carrier boards 14

2.2.5 PC-104 boards 14

2.2.6 CAN PCMCIA card (CANCard) 14

2.3 CAN basics 14

2.3.1 Some commercial aspects 14

2.3.2 Some technical aspects 14

2.3.3 Network topology 15

2.3.4 Messages and their priorities 15

2.3.5 Protocol layer 16

2.4 System architecture 17

2.4.1 Software package delivery NEW! 18

2.4.2 Device Driver Update Package 18

2.4.3 Software Development Kit 18

2.4.4 Software Development Kit with Device Driver Source Code 18

2.5 Firmware 19

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3 Creating a skeleton application UPDATED **20**

3.1 Basics ... 20

3.1.1 Communication mechanisms ... 20

3.1.2 'Firmware request messages' and 'CAN message frames' ... 20

3.1.3 Application structure ... 21

3.1.4 Library support ... 21

3.2 Device initialization UPDATED ... 22

3.2.1 Program structure UPDATED ... 22

3.2.2 A fundamental base application ... 23

3.2.2.1 Device access strings ... 24

3.2.2.2 Including header files and libraries ... 25

3.2.3 Base application ... 25

3.2.4 The all-in-one initialization function ... 26

3.2.5 Getting version information without message handler UPDATED ... 27

3.2.6 Initialize a device with message hander ... 30

3.2.7 Getting version information with message handler ... 31

3.2.8 Configuring DPM queues ... 33

3.3 BusOn request ... 33

3.3.1 'BusOn' request with ICANOS Firmware ... 34

3.3.2 'BusOn' request with CAL/CANopen firmware ... 35

3.4 Device reset ... 35

3.5 Using the internal CAN bus termination ... 37

4 Standard host interface **39**

4.1 Sending CAN frames ... 39

4.2 How to write message handler code NEW! ... 42

4.3 Receiving CAN frames UPDATED ... 44

4.4 Firmware error condition handling UPDATED ... 45

4.5 Message filtering ... 46

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

4.5.1 Software message filtering 47
4.5.2 Hardware message filtering 49
4.5.2.1 Hardware message filtering for standard CAN frames 49
4.5.2.2 Filtering standard CAN frames in single filter mode 51
4.5.2.3 Filtering standard CAN frames in dual filter mode 52
4.5.2.4 Hardware filtering for extended CAN frames 53
4.5.2.5 Filtering extended frames in single filter mode 53
4.5.2.6 Filtering extended frames in dual filter mode 54
4.6 Cyclic sending of CAN frames 55
4.7 CAN bus error handling NEW! 58
4.8 Bulk buffer 59
4.9 SNIFF buffer 62
4.9.1 SNIFF buffer initialization 62
4.9.2 SNIFF buffer configuration 63
4.9.3 SNIFF buffer message handling 64
5 Fast Host Interface 67
5.1 Initializing the Fast Host Interface 67
5.2 Sending CAN frames 69
5.3 Receiving CAN frames 70
5.4 Special features for using a second CAN channel UPDATED 72
5.4.1 Handling the second CAN channel with the 'Fast Host Interface' 73
5.4.2 Swapping between both CAN controllers 78
6 CANopen 80
6.1 Device configuration file (DCF) 80
6.2 Communication objects 81
6.3 Object dictionary 81
6.4 Initializing a slave node 81
6.5 Process data objects 85

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

6.6 Sending data of a asynchronous PDO 89
6.7 Receiving data of a asynchronous, event-based PDO 90
6.8 Receiving data from a asynchronous, requested PDO 92
6.9 Initialization of a master node 93
6.10 Emergency CAN frames 95
6.10.1 Sending emergency CAN frames 95
6.10.2 Receiving emergency CAN frames 95
6.11 Synchronous PDOs / CAN frames for synchronization (SYNC) 97
6.12 Service data objects (SDO) 99
6.13 Extended network management 102
6.13.1 Switching single nodes into OPERATINAL mode 104
6.13.2 Node guarding protocol features 105
6.13.2.1 Node guarding for an NMT slave 105
6.13.2.2 Node guarding for an NMT master 106
6.13.3 Heartbeat protocol features 107
6.13.3.1 Heartbeat producer features configured in the DCF 108
6.13.3.2 Heartbeat producer features configured in the application 109
6.13.3.3 Heartbeat consumer features configured in a DCF 110
6.13.3.4 Heartbeat consumer features configured in an application 112
6.13.4 Bootup protocol features 113
6.14 CANopen error handling 115

7 Appendices 119

7.1 MODULbus 119
7.2 Philips SJA1000 filter modes 119
7.2.1 Single Filter mode 119
7.2.1.1 Standard frame: 119
7.2.1.2 Extended frame: 120
7.2.2 Dual filter mode 121

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7.2.2.1 Standard frame 122
7.2.2.2 Extended frame 122
7.3 Firmware interfaces 123
7.4 DPM queue sizes 124
7.5 System requirements 125
7.5.1 Firmware 125
7.5.2 Device driver 125
7.5.3 Janz library support 125
7.6 References 126
7.7 Manual History 126

List of Figures

Figure 1: CAN network topology 15
Figure 2: Structure of a CAN message frame 16
Figure 3: CAN and the ISO/OSI communication layer model 16
Figure 4: System architecture 17
Figure 5: Program structure 23
Figure 6: The Janz VMOD-ICAN3 MODULbus module 119
Figure 7: Single filter configuration, receiving standard CAN messages 120
Figure 8: Single filter configuration, receiving extended CAN messages 120
Figure 9: Dual filter configuration, receiving standard CAN messages 121
Figure 10: Dual filter configuration, receiving extended CAN messages 123

List of Tables

Table 1: Supported hardware 12
Table 2: Supported hardware for Janz ICANOS daemon 13
Table 3: Provided libraries 18
Table 4: Provided firmware versions 19

 © **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Table 5: Terms used for message types 21

Table 6: Library names 22

Table 7: Device names 24

Table 8: Macros for distinguishing settings for different operating systems 25

Table 9: Firmware check strings 31

Table 10: Firmware error conditions 46

Table 11: Message filtering configuration functions 47

Table 12: Data structure of a bulk buffer indication message 61

Table 13: CAN adapter devices for which an optional second CAN channel is available 73

Table 14: CAN adapter devices for which an optional second CAN channel is available 81

Table 15: Identifier ranges for different communication objects 86

Table 16: Functions for writing PDO data 87

Table 17: Functions for reading PDO data 89

Table 18: Heartbeat consumer time settings in file hb_cons.dcf 110

Table 19: CANopen error cases 118

Table 20: Manual history 127

About this Manual

This is the <<type of manual>> manual for the <<product name>> <<product description>>. It gives all necessary information to users and programmers of the <<product name>>. We have tried to keep it compact, so there are no special sections for users and programmers. I.e. everything that can be said about the serial port is said in one place (connector layout, signal levels and programming hints). Users that are not programmers might therefore only read the first sections (Introduction and Installation), and come back to the detailed sections when they need special information.

The manual starts of with an **Introduction** to the <<product name>>. This is some prose that describes the features and architecture of the <<product name>>. You should already know this if you have bought the board! It then discusses some topics about **Installation** of the product.

The rest of the manual then covers **technical details** about the <<product name>>.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Conventions

If numbers are specified in this manual, they will be either decimal or hexadecimal. We use C-notation to identify hexadecimal numbers (the 0x prefix).

If we refer to low active signal names, they will suffixed by a “#” character.

Some parts of the text are really important. These are visually marked with the following signs:



Indicates information that when not fully understood or followed might cause permanent damage to the system. You should not start using the product before you have read this information.



Indicates information that we think you should have read to save your time by avoiding problems. Important suggestions that should be followed will also be marked with this sign.



Refers to the corresponding sample program code. This tutorial normally comes with a lot of sample source code together in one archive. This symbol indicates a location in the document to which a sample program belongs to. The text beside this symbol gives the reader the path to the sample source code file.



New term

Signals term definitions in this document. Sometimes we like to define some new terms to make it easier to use them in the following text. This symbol signals a new term definition in the text. The new term is printed below the symbol to make easier to find at a later time.

Sometimes source code snippets are embedded in the text of the tutorial. We will do this in the following manner:

```
int main(int argc, char* argv[])
{
        // device information
        dpm_dev_t devInfo;
        // device descriptor
        int fd;
        // buffer for error messages
        char errBuff[ERROR_MSG_BUFFER_SIZE];

        // print header information
        printf ("*************************************\n");
        printf ("* Janz Automationsssysteme AG          *\n");
```

Code lines which are important for the currently discussed topic are printed in colour. To be able to find these code lines in the corresponding sample source code file, always some lines above and some lines below are also printed but in grey colour.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Acronyms and Abbreviations

CAN	Controller Area Network.
ICAN	Janz intelligent CAN device, a CAN device with an own microcontroller onboard.
MODULbus	Industry standard mezzanine bus system.
VME	Versa Module Europe.
SDO	Service data object
PDO	Process data object
EMCY	Emergency data object
SYNC	Data object for network synchronization
OD	Object dictionary

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

1 Introduction

1.1 Purpose of the document

The purpose of this document is giving the reader an easy possibility to learn something about programming of an application, which uses the facilities of the intelligent Janz CAN modules. The tutorial starts with a white paper and shows a programmer step-by-step how to program a CAN bus application using functions of the Janz CAN libraries which are delivered with the product.

Janz CAN module device drivers and libraries are available for different operating systems and system architectures. In addition several different development tools can be used for each operating system. So this tutorial cannot explain how to work with a specific development environment. For working with this tutorial it is absolutely necessary, that the programmer has already a good knowledge about the used operating system and development environment.

The used programming language in this tutorial is C (not C++!). We use C to be able to support as much programming languages as possible. Programmers preferring C++ can easily include the libraries in their C++ development environment, but we do not explain how to this here. It is important, that the user has a good knowledge about this programming language in any case, because we will only explain programming language specific stuff, if it is necessary for understanding procedures in the example programs. So, this tutorial is not suitable for learning something about C or C++.

Furthermore the reader of this tutorial should have detailed knowledge about the CAN bus itself. This tutorial is not suitable to learn something about CAN. In fact, the CAN specifications contain the base knowledge to understand this tutorial!

This tutorial is only a programming guide. Library functions are not described in great detail. You'll find more detailed descriptions in the Janz CAN software reference manual which can be downloaded as PDF document from our download area as well as the most actual version of this tutorial.

1.2 Examples

You will find example programs for each chapter in this tutorial in the 'code' sub-directory in the tutorial's installation path. If this tutorial was delivered without source code (for example to reduce size), take a look at our home page. The tutorial is also available for download there with example programs.

An own sub-directory was created for each chapter and each section in this document where examples are used. Furthermore an additional sub-directory can be found for each supported operating system which provides you helpful accessories for compiling the examples such like a project file for Microsoft's Visual Studio and a 'makefile' for Linux. The sample programs are structured very easily, so that it should be also no problem to adapt the code for other operating systems and development environments.

1.3 Document structure

The document is divided into several chapters. Some chapters rely on the stuff read in the one before. You will find more information about how to read this document in the sections below. Subsections will lead the reader to the final result of a chapter and are always relying on the previous subsections of the same chapter. Some chapters contain operating system dependent stuff. In such a case a header page is provided in the currently selected page containing some common stuff and the reader will find

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

a list in this page in which the wanted operating system can be selected. After that the operating system dependent information for this step will be displayed.

- **Chapter 1** is this introduction.
- **Chapter 2** contains information about the installation of the Janz CAN hardware and about software installations.
- **Chapter 3** will lead the reader in programming a skeleton application for using the capabilities of the intelligent Janz CAN modules. Here also a lot of the system dependent stuff is described. The chapter should be read in any case.
- **Chapter 4** describes the standard firmware software API.
- **Chapter 5** describes the 'Fast Host Interface', which is available in each kind of Janz CAN firmware. The 'Fast Host Interface' represents a fast and easy way to use the CAN layer 2 facilities of the intelligent Janz CAN modules and is available in each firmware version (ICANOS as well as CANopen). Programmers who do not want to use this interface can omit this chapter.
- **Chapter 6** describes the CANopen firmware software API.
- **Chapter 7** contains some appendices. Here the version information and some FAQs (Frequently asked questions) can be found. This chapter can vary from minor version to minor version! Adding a chapter to this tutorial normally starts with an FAQ or a small additional chapter in the appendix. If we got response from our customers, that they like a chapter or that they want to have more detailed information about a theme discussed in a chapter, this chapter begins to grow up until we decide to add it to the main part of this tutorial.

The standard host interface is suitable for simple CAN Layer-2 data transport tasks for which performance and time accuracy is not so important. The fast host interface provides a high performance CAN Layer-2 interface. The CANopen interface contains the CANopen protocol stack.

1.4 By the way …

Of course, we know that this tutorial is not free of mistakes. If you found a mistake, please contact us! We will register this mistake and correct it as soon as possible. Normally the mistake should be corrected with the next tutorial version!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

2 Hardware

This chapter should give you some common information about Janz CAN products. First we will list and describe the hardware for which this document was written for. Most of Janz CAN products can be used with this tutorial because they are all designed quite similar. After that some aspects of hardware installation will be discussed.

Also a chapter is contained which will give you a really short introduction into CAN. We can give here only a short overview about CAN, because this topic is to complex for being totally described in this document. Therefore the standard literature is more suitable and should be read from a newcomer anyway!

Finally in this chapter we explain how a system is structured having a Janz intelligent CAN product installed and a user application running which uses a CAN connection. **Chapter 2.5** provides you descriptions of device driver installations for the most of the supported operating systems. **Chapter 2.6** contains important information about the available firmware for Janz CAN hardware.

2.1 Supported hardware UPDATED

Janz intelligent CAN products are very similar looking from the hardware site. In most cases only the hardware interface are different. Components like processors and CAN controllers are either identical or have an identical software interface implemented in firmware. They mostly are derived from the Janz VMOD-ICAN3 MODULbus module which was the historically first device of this type.

The VMOD-ICAN3 module can be plugged on different carrier boards. So, this MODULbus module can be accessed via different internal bus architectures. But these differences are equalized through device drivers providing one and the same software API to a program writer.

We implemented our library functions for different operating systems in the same manner. This gives us the possibility to provide quite the same software API over all supported operating systems. Therefore this tutorial is not only suitable for several Janz CAN hardware products; it also can be used for different operating systems.

This tutorial is suitable only for 'intelligent Janz CAN products' which means that the products need an own processor having a firmware running on it. The VMOD-FCAN MODULbus module, the CAN-USB device and the CAN-PCIL adapter card for example do not have an own processor and can therefore not be used with this tutorial.

Picture	Product name	Description
	VMOD-ICAN3	intelligent CAN MODULbus module, can be plugged on different carrier boards
	CAN-PCI	PCI carrier board with pre-installed VMOD-ICAN3 module on MODULbus slot 0

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

	CAN-PC	ISA carrier board with pre-installed VMOD-ICAN3 module on MODULbus slot 0
	CAN-104	ISA carrier board in PC-104 format with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with a second CAN channel.
	CAN-PCI2	PCI carrier with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with a second CAN channel.
	CAN-CP6	PCI carrier board in CompactPCI format, 6 units high, with an pre-installed VMOD-ICAN3 module on MODULbus slot 0
	CAN-CP3	PCI carrier board in CompactPCI format, 3 units high, with an pre-installed VMOD-ICAN3 module on MODULbus slot 0
	CAN-PMC	PCI carrier board in PMC format with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with a second CAN channel.
	CAN-PMC2	PCI carrier board in PMC format with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with a second CAN channel.
	CAN-PCIH	PCI carrier board in PMC format with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with four CAN channels. DMA master transfer can be used if the device driver is prepared for that.
	CANcard	Intelligent CAN adapter device with two CAN channels for PCMCIA slot.
	CAN starter kit	Starter kit including CAN hardware for different hardware architectures, a CAN cable, an I/O CAN node and a CD-ROM containing device drivers and software tools

Table 1: Supported hardware

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Because we do not want to enumerate all these products in our tutorial chapters each time when we are talking about the hardware, we will speak from 'intelligent CAN devices' or from ‘ICAN devices’. Please keep in mind, that therewith all the supported products are meant.

Janz Automationssysteme AG provides a software product named ‘ICANOS Daemon’. This product contains a background process which emulates the firmware behaviour for devices which have no own microcontroller on board. With help of this software also non-intelligent CAN device can be made behaving like an intelligent device. Of course, performance is not the same but if you don’t need time accuracy and high data through-put the ICANOS daemon could be a cost-effective alternative to the intelligent CAN devices.

The Janz ICANOS daemon is also available for Janz emPCs! In that case a remote TCP/IP connection will be used for exchanging data with the device. The DPM libraries are able to handle data transport in a compatible manner. An application developed with the DPM software development kit can work with an ICANOS daemon as well as with an intelligent CAN device. Nevertheless not all interfaces available for intelligent Janz CAN devices were implemented in the Janz ICANOS daemon (Bulk buffer and Sniff buffer not available). Also firmware download is not possible. Instead the ICANOS daemon process needs to be exchanged manually on the corresponding host system.

Picture	Product name	Description
	CAN-USB	USB CAN Adapter
	emPC	Janz embedded PC system, CAN interface integrated
	CAN-PCIL	Non-intelligent PCI CAN adapter.

Table 2: Supported hardware for Janz ICANOS daemon

2.2 Hardware installation

If the modular version of a Janz CAN product is used, the CAN module has to be plugged into a MODULbus slot on the carrier board. If you use the components of a Janz CAN starter kit or a bundle product like the CAN-PCI, this was probably already done. Products like the CAN-PCI2 and the CANcard have the CAN MODULbus module already integrated in their hardware on MODULbus slot 0. Please keep this in mind during the chapters of this tutorial. It will become important some sections below!

Below available carrier board architectures are listed with a description of the hardware settings, which are important for the programming of a CAN application. These settings can only be seen and checked, while the CAN hardware is not installed in the host system. So we recommend that you check these settings before the board is built in. You will find more information about how the hardware can be configured in the corresponding hardware manual of the board.

At least the Janz CAN hardware needs to be plugged in a free slot in your host system. If the Janz CAN module is plugged on a place, where the CAN connector is not reachable from outside your host system, you can use a further connector on an own slot sheet.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

2.2.1 PCI boards

PCI boards have a switch on board, with which the board number has to be set. All used boards need to have different board numbers. Default is 0. If you are using a modular CAN interface (MODULbus module), the slot number is needed for the application's initialization procedure. The number is printed on the carrier board in white or yellow colour. Default is 'MODULbus 0'. Please use 0 as slot number if you have no modular CAN device.

2.2.2 CompactPCI boards

CompactPCI devices can be handled like PCI devices (please see above).

2.2.3 PMC boards

PMC devices can be handled like PCI devices (please see above).

2.2.4 ISA carrier boards

The board number of ISA devices depends on what number was assigned to the device during the device driver configuration. Assigning a device number for ISA devices is extremely operating system dependent and will be described in this document in the system corresponding part of chapter 2.5.



If an ISA device is used in a PC host system, please note, that an interrupt must be available for ISA access. Please check this in the BIOS settings of your host system. Mostly problems with the ISA device occur because of wrong interrupt settings in the BIOS!!

2.2.5 PC-104 boards

PC-104 devices can be handled like ISA devices (please see above).

2.2.6 CAN PCMCIA card (CANCard)

The CANCard has no possibility to adapt its board number. As board number for this device type always 0 has to be used. Accordingly only one CANcard can be used at all in each system, except you are using Microsoft Windows 2000 or Microsoft Windows XP as operating system! In this case the board enumeration is done automatically by the system, so that more than one device can be supported. If a MODULbus slot number is requested for this device type, always start with 0 for the first device and use continuous numbers (1, 2, 3 ...) for further devices.

2.3 CAN basics

2.3.1 Some commercial aspects

CAN is a long life technology with a rich catalogue of industrial components and projects in all sectors of industry and it is an open standard supported by many automation suppliers and end-users. Cheap controller chips, no license fees and a high number of pieces because of the volume of automotive applications are the reasons why the Controller Area Network can be called a low-cost field bus system.

2.3.2 Some technical aspects

CAN is a high speed technology which is able to transfer data with up to 1 Mbit/s transfer rate and a minimum of protocol overhead. The protocol is hardware implemented and guarantees a very fast bus

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

access mechanism with short latency times. CAN is suitable for realization of distributed real-time systems.

Due to the fact that the arbitration mechanism was implemented in hardware as well as error detection and an error validation is available. CAN provide high transfer reliability and has the highest reliability of all field busses. Faulty messages will be signalled and transferred again automatically. 'Bad' bus participants will be recognized and disconnected to guarantee that such member will not disturb the bus longer than necessary. Furthermore the protocol inherents multi-master abilities.

2.3.3 Network topology

A CAN network consists per definition of at least two nodes or more. All nodes need to be connected to one single cable in a line topology. The cable, which connects the nodes, needs a termination resistor at both ends. Many CAN devices are able to activate an internal termination if they are placed at an end of a CAN bus. This is also possible with the Janz CAN hardware. How to activate the bus termination by software is explained in one of the following chapters.

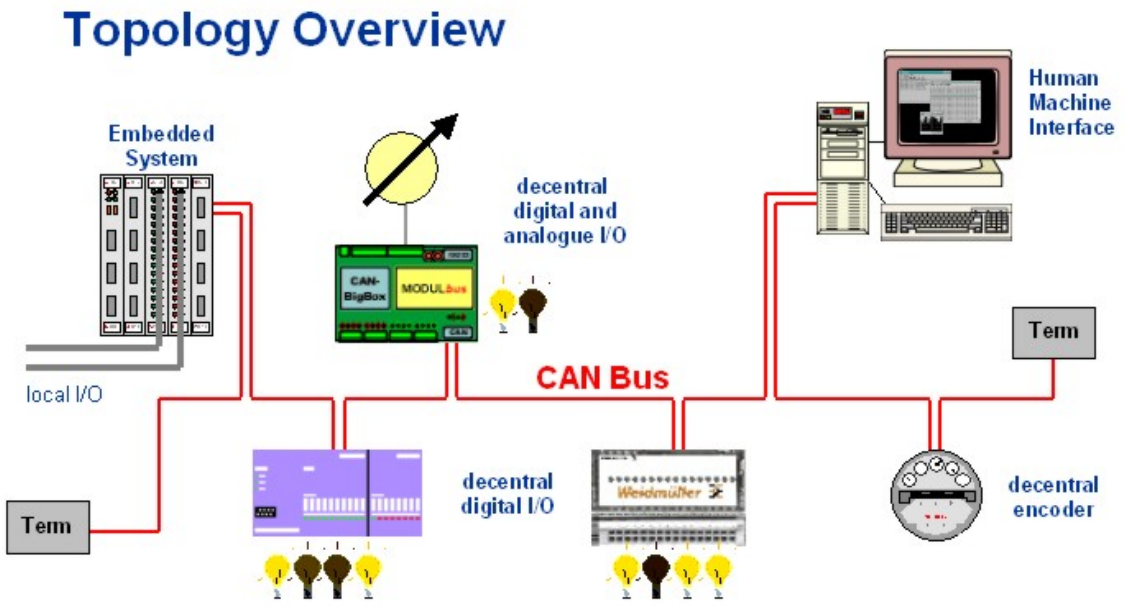


Figure 1: CAN network topology

2.3.4 Messages and their priorities

CAN is a message oriented transfer mechanism. Each message has an identifier, some administrative information and up to 8 data bytes. A message can only be identified by the message identifier; the message contains no information about from which node it was sent or which node shall receive it. All bus participants will receive a message and have to decide whether they want to work with this message or not. To avoid a node from receiving unwanted messages extensive message filter mechanisms are available.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

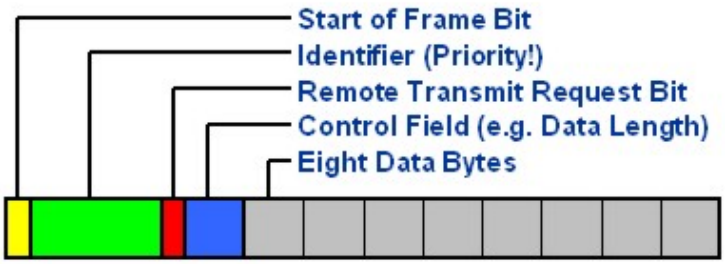


Figure 2: Structure of a CAN message frame

Each node is able to receive and to send messages at any time. If two nodes want to send a message at the same time, a special arbitration mechanism is used to determine which message is sent first. This arbitration mechanism uses the message identifier to grant a priority to a message. The message with the lowest identifier will get the higher priority and will be sent first. The node which could not send its message, because the arbitration was lost, will try sending the message again after sending of the higher prioritized message has finished.

2.3.5 Protocol layer

CAN implements the layer 1, 2 and 7 of the ISO/OSI communication layer model. Layer 1 and 2 are implemented in hardware (CAN controller). The CAN Layer 2 communication simply uses the capabilities of the CAN controller. CAN layer 2 communication provides a relative simple communication mechanism which simply works on basis of the raw CAN message frame. For Janz CAN products the ICANOS firmware provides layer 2 access to the CAN controller on Janz intelligent CAN boards. Writing programs for CAN layer 2 will be explained in chapter 4 ‘Fast Host interface’ and in chapter 5 ‘ICANOS’.

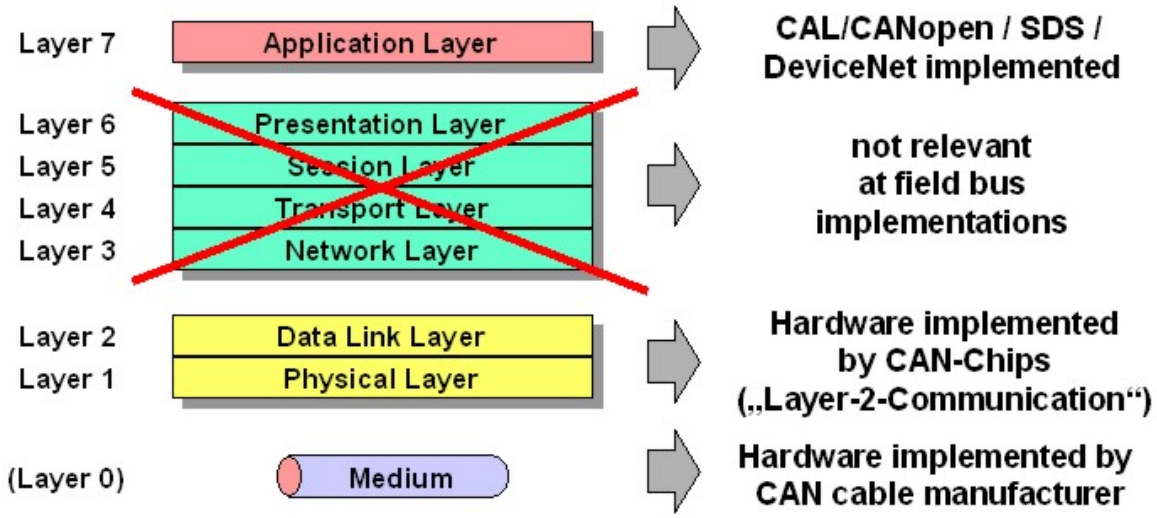


Figure 3: CAN and the ISO/OSI communication layer model

Based on layer 2 several higher layer protocols were developed in the past (CAL, CANopen, DeviceNet, etc.). These protocols implement several standardized services for network management, node guarding, usage of multiplexers, etc. The different higher layer protocols are not compatible. CAL and CANopen are open, manufacturer independent protocols. For Janz CAN products CANopen is

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

available as higher layer protocol using the corresponding firmware. The CANopen capabilities are described in more detail in chapter 6 ‘CANopen’.

2.4 System architecture

Looking from hardware side the host system consists of a computer, which has a Janz CAN board installed, which is a CAN MODULbus module on a system dependent carrier board or a compatible hardware in most cases. Looking from software side for both, carrier board and Janz CAN module, a device driver needs to be installed (also for integrated hardware products). For accessing the Janz CAN hardware the DPM device driver is used. This device driver needs a carrier board device driver to get access to carrier board hardware functions.

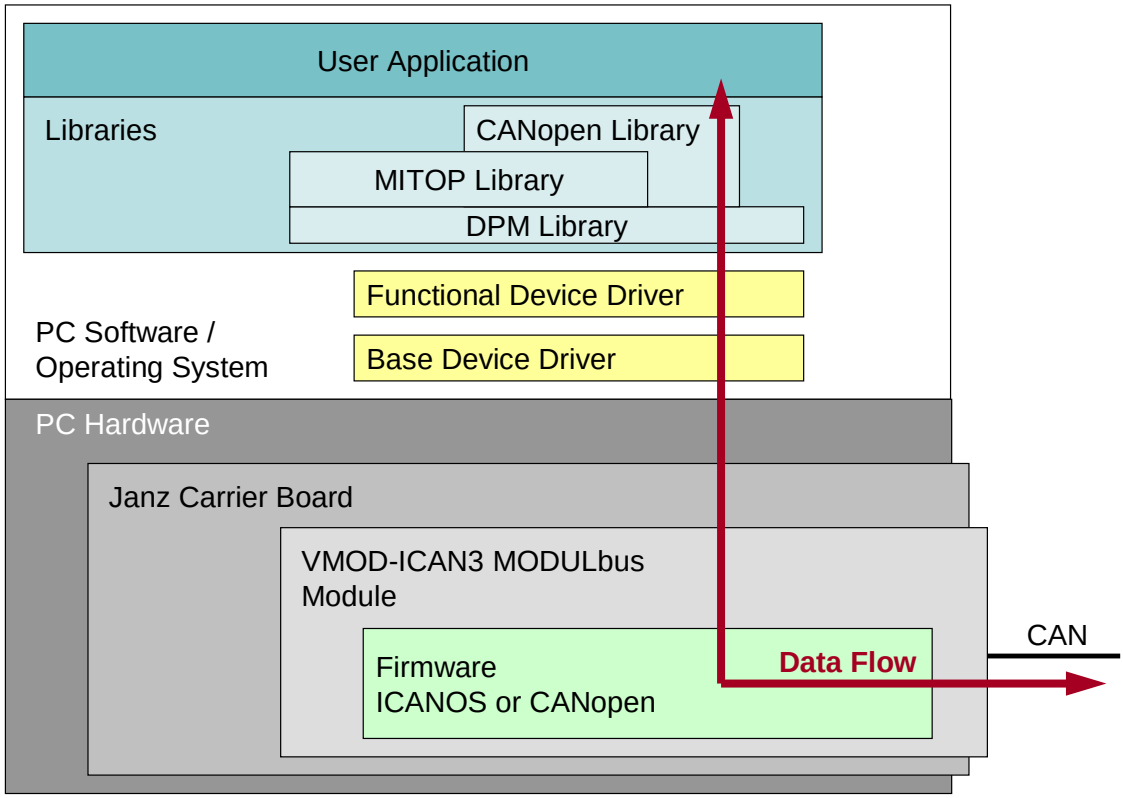


Figure 4: System architecture

Some of the Janz CAN products have the Janz CAN MODULbus module integrated in their hardware design, so that this module cannot be physically seen. Nevertheless the module's hardware is present. So, these products can be handled like a carrier board which has a Janz CAN module plugged into slot 0. For both, modular and integrated products, the same device drivers (carrier board device driver (!) AND (!) DPM device driver) are needed.

To make it easier for a programmer to use the DPM device driver functionality, some libraries are provided. The following table lists these libraries with a short description of the included functions. These libraries are provided for all supported operating systems but the library file names differ in some cases.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Library	Included functions
dpm, can_lib	Base functions for DPM device driver access
Mitop	Functions to fill data structures with complex data to be sent to Janz CAN firmware.
CANopen	Functions for using CANopen functionality

Table 3: Provided libraries

How to link these libraries with an application and how to use their functions is explained in Chapter 3: 'Creating a skeleton application'. The following figure shows the way data takes from the application over the libraries and the firmware down to the CAN bus.

2.4.1 Software package delivery NEW!

Hardware access to Janz CAN devices is made via device drivers and libraries. Both are provided in several software packages each created for different purposes.

2.4.2 Device Driver Update Package

Device Driver Update Packages contain only binary files of both, device drivers and libraries. Furthermore an installation routine or installation instructions are contained. This package type was created to enable customers to make bug fixes and updates for new hardware designs without changing their application software. Device Driver Update Packages are available for download on our homepage. They are not suitable for software developers, because they do not contain documentation and components (e.g. header files) needed for software development. Furthermore we will not provide customers support for Device Driver Update Packages. Device Driver Update Packages can be freely re-distributed to other people and customers.

2.4.3 Software Development Kit

Software Development Kits are needed if Janz hardware should be included in own software projects. They contain the components which a software developer needs to access Janz hardware from within self-written applications. Software Development Kits are not free of charge. A corresponding license needs to be ordered once per customer.

Software Development Kits are also available for download from our home page, but corresponding software packages are password protected. A license key is needed to be able to install them. A customer will get his license key when the Software Development Kit was ordered. After that he will have access to all future updates of the Software Development Package, because he can always download the actual version from the internet.

2.4.4 Software Development Kit with Device Driver Source Code

Software Development Kits can also be ordered together with the source code of the corresponding device driver. Customers, who work with operating systems which are not supported from Janz, can write their own device drivers with help of the source code. Best practice is to order the device driver source code for an operating system which is quite similar to one for which the custom driver should be developed.

Software Development Kits with Device Driver Source Code are not available for download. They need to be ordered separately. Software Development Kits with Device Driver Source Code must not be given to other people or customers!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

2.5 Firmware

On the Janz CAN module an own firmware is running which manages sending messages to and receiving data from the CAN bus as well as some configuration and initialisation settings. There are different types of firmware which can be installed on the module. The list below gives you an overview about what firmware types are available and for what purposes these firmware types have to be installed:

Firmware	Description
ICANOS	This firmware enables the programmer to interact with the CAN bus on layer-2 protocol basis. In parallel a very fast Layer-2 based access over a special interface (Fast Host Interface) is available.
CANopen	This firmware provides the CANopen capabilities to the application programmer. In parallel a very fast Layer-2 based access over a special interface (Fast Host Interface) is available.

Table 4: Provided firmware versions

Firmware installed on a Janz CAN module can be exchanged with a command line based application called ‘fwload’ which is contained in the DPM software development kit. Firmware is normally not contained in the software development kit. It can be either downloaded from our home page or requested from us. Please keep in mind, that CANopen firmware is not free of charge! License fees needs to be paid for each installation!

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3 Creating a skeleton application UPDATED

The 'Hello world!' program is one of the classic entry points for programming lessons. We also want to write a very easy program first, which only makes a connection to the Janz CAN module and prints out some text messages on your screen. In the following steps we will expand this small example program to get some information about device drivers and firmware versions from the hardware.

During this chapter we will build the skeleton application, which is used as basis for following steps. To have no needless work we will expand this application step-by-step and only describe the changes during the next steps. Therefore you should start with the first chapter and go through this continuously!

Former versions of this document described two different ways how devices can be initialized, the convenient and the conventional device initialization. The conventional method is declared as obsolete now! Functions needed for this method will be still contained in the libraries and supported from us. But customers who have new projects should use the device initialization described in this document which is the one formerly called convenient method. The corresponding library functions are available for nearly all operating systems in the same syntax.

Nevertheless we could not keep away all system dependencies from the program writer. Normally we do not describe system dependencies and how to handle them in this document exceptionally when system dependencies influence the API of our libraries. Therefore you'll find some pre-processor directives which make our examples usable with all supported operating systems. Pre-processor directives are not set in program code; they need to be set in your compiler environment.

3.1 Basics

3.1.1 Communication mechanisms

Communication between an application and a Janz intelligent CAN device (ICAN device) is structured like a communication between two independent communication partners. Both are able to send and to receive data at any time without paying attention to the other. The information is exchanged via messages between an application and the CAN device. So each communication partner has to provide an independent possibility for sending and receiving messages. To be sure, that no message is lost because of the other communication partner has to do some other work, different message queues were implemented in the CAN device's firmware for both directions. These message queues are all FIFO (First In First Out) queues.

3.1.2 'Firmware request messages' and 'CAN message frames'



firmware request, firmware response

Messages are sent in both directions in parallel between application and CAN device. Please note that there are also messages sent over the CAN bus! But these messages are completely different to the messages which are exchanged between an application and the CAN device. To be able to distinguish both message types in this document we have defined some terms:

- If we are talking about CAN message frames, we will use the terms 'frames' or 'CAN frames' in this document. With this the raw data sent over the CAN bus is meant.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

- If we are talking about data which is sent from the application to the CAN device, we will use the terms 'firmware requests' or 'request messages'.

- If we talk about messages sent from the CAN device to the application we will use the terms 'firmware response' or 'firmware indications.

We hope these definitions will make it easier to understand about what we are talking. For a better overview the table below lists again the different terms and their meaning.

Terms	Meaning
'Frames' or 'CAN frames'	Message data transferred over the CAN bus
'Firmware request' or 'request message'	Message data transferred from the application to the CAN device
'Firmware response' or 'response message'	Message data transferred from the CAN device to the application

Table 5: Terms used for message types

3.1.3 Application structure

A CAN application working with Janz intelligent CAN products can be divided into two main parts, the main application and a message handler thread. Both parts are running asynchronously. Each part is responsible for one communication direction we have talked about above.

- The main application is the part which contains the program logic. Generally this part is also responsible for sending firmware request messages (direction: application -> CAN device).

- The message handler thread is controlled by an event mechanism and is responsible for receiving firmware response messages (direction: CAN device -> application).

This structure becomes not implicitly visible, because a call-back mechanism was defined to hide the thread handling from the program writer. Thread handling is strongly system dependent and is maintained internally. But behind the library function still works the message handler thread!

3.1.4 Library support

For getting access to the device and for supporting the higher layer protocol 'CANopen' some libraries are provided by Janz and have to be linked to the application. Both, the main application and the message handler thread, are able to call functions in the libraries.

The libraries are delivered in source code as well as in binary format in the software development kit. For the supported operating systems the auxiliary equipment for compiling the libraries can be found in the corresponding directories. The table below contains an overview over available libraries and header files together with a short description about which functionality is provided. Unfortunately the file names differ a little bit from operating system to operating system. So please, take a look into the 'lib' sub-directory of your software development kit installation for seeing which name is used in case of your operating system. In this document we normally use the common library name printed in the first line of the ‘functionality’ column in the table below.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Library names	Corresponding header file	Functionality
jic3dpm.dll, ic3dpm.dll, can_lib.o or libcan.a	can_lib.h	**DPM** (device driver) library The DPM library provides functions to access the device driver. It is always operating system dependent. That means functions contained in this library need not to have the same API over all operating systems.
jic3mitp.dll, ic3mitop.dll, mitop.o or libmitop.a	mitop.h	**MITOP** library (**M**essage **I**nterface **TOP**) This library makes it easier to exchange data with the firmware on the Janz CAN module. It provides functions creating firmware request messages for firmware communication. The API of this library depends not on the used operating system. The same source code is used to compile this library for all supported operating systems.
jic3cop, ic3cop, canopen.o or libcop.a	ic3copif.h or copif.h	**CANopen** (host) library This library provides CANopen functionality to the host program. The API of this library depends not on the used operating system. The same source code is used to compile this library for all supported operating systems.

Table 6: Library names

3.2 Device initialization UPDATED

We tried hardly to implement a homogenous interface over all operating systems for device initialization. Only few functions are needed for the complete mechanism.

Initializing a Janz CAN device really is not a hard work! We will start with some theoretical stuff, so that you are able to understand the steps automatically executed by the all-in-one initialization function. After that we will jump directly into the initialization procedure, because the initialization is quite simple to program. At least we will discuss some initialization options which can be set in the used data structure and some helper functions which will make live a little bit easier.

3.2.1 Program structure UPDATED

Like already described in chapter 3.1.3 ‘Application structure’ your program will be a multithreaded application using the main thread for writing your program logic and using a message handler thread for receiving firmware responses from the device. The initialization method hides this topology a little

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

bit by using a call-back mechanism. Each time a firmware response is received, a call-back function will be called for handling from within the library. But keep in mind that this call is done asynchronously from another thread! That means that you possibly need serialization mechanisms to avoid problems in data access.

Device initialization is done with one single call of function ‘can_device_setup’. Instead of making several function calls with different parameters the initialization parameters are bundled in one data structure which is named 'dpm_dev_t'. The programmer fills in all the initialization parameters into this data structure and let 'can_device_setup' do the work with it.

If message handling is wanted (in most cases), the program writer only needs to give a pointer to a call-back function to the setup function with help of data structure 'dpm_dev_t'. If this is done, the function will set up the thread. After that the thread will call the call-back function each time a firmware response could not be handled. Some firmware responses are handled automatically without notifying the application (only with CANopen firmware).

Keeping these things in mind we will discover, that the main program now exists of a main function and a call-back function if the message handler was set up. The figure below shows you the program structure in a graphical manner.

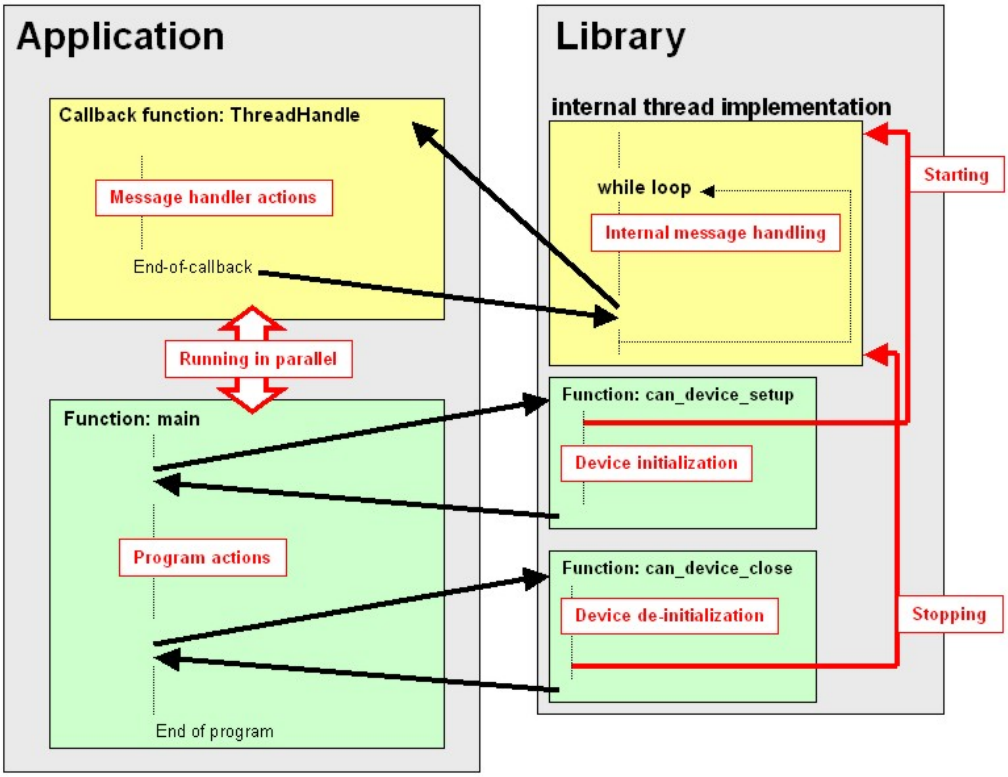


Figure 5: Program structure

3.2.2 A fundamental base application

This chapter does not really explain things regarding the Janz CAN module. We will create a common base application. But we will also explain some things which are not directly needed for the work with Janz CAN devices but which are useful for it! The explained program structure will be used in our entire example programs. So we guess this is one of the most important chapters!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.2.2.1 Device access strings



device name, device access string

For opening a connection to a Janz CAN device we will need to specify which device in our system we want to connect to. This is made with a string containing the device information. We call this string 'device access string' or ‘device name’. Depending on the used operating system, this string looks different. In the table below examples are shown for device access strings for supported operating systems.

In sample programs coming with this tutorial we will enter the device access string as program parameter, so we haven’t got operating system dependent stuff in our example code. Library function can_device_setup will parse the string and create the device connection.

Device name example	Operating system	Description
dpm_00	Microsoft Windows 2000, Microsoft Windows XP	The device access string consists of the constant string ‘dpm_’ followed by the carrier board number (set with rotary switch on the board) and the number of the MODULbus slot on which the CAN module resides
canusb0	Microsoft Windows 2000, Microsoft Windows XP	A special device access string is used for accessing the CAN-USB adapter via ICANOS daemon for example. It consists of a string prefix ‘canusb’ and a following device number.
mpcan_00	Microsoft Windows 2000, Microsoft Windows XP	The original device name of a CAN-PCIL adapter can be used for accessing it via an ICANOS daemon. It consist of a leading string ‘mpcan_’ followed by the board number and the number of the CAN channel to use.
/dev/dpm_00	Linux	The device access string is a directory path. The device's mount point exists in directory /dev. The device name also starts with the prefix 'dpm_' followed by the carrier board number (set with rotary switch on board) and the number of the MODULbus slot on which the CAN module resides.

Table 7: Device names

For Windows 2000 and Windows XP also an IP address of an emPC followed by a port number can be used as device name. Both, the IP address and the port number, needs to be separated through a colon. Using '192.168.0.190:20119' would cause the open function to try establishing a connection to the emPC with IP address 192.168.0.190 on port 20119 for example.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



Not all interfaces described in this document are available to be used over a TCP/IP connection. The bulk buffer interface and the SNIFF buffer interface for example are not implemented. Furthermore the performance of a remote connection cannot be as well as it is, when your application uses a device built in your local host. Please keep this in mind!

3.2.2.2 Including header files and libraries

Because we do not want to explain including libraries each time a library is needed, we will explain this once for all libraries delivered in the device driver releases. The programmer has to decide which library is needed for which example program. In the Janz ICAN Software Reference Manual you will find more information about which function is stored in which library.

To include libraries in your development environment the following directories have to be added to search path for header files first. How and at which position this is made depends on the used development environment. <DRV_INSTALL_DIR> is the installation directory of the DPM software development kit, <OS> is a shortcut for the used operating system.

```
<DRV_INSTALL_DIR>\h
<DRV_INSTALL_DIR>\h\<OS>
<DRV_INSTALL_DIR>\ican\canopen\h
<DRV_INSTALL_DIR>\ican\mitop\h
<DRV_INSTALL_DIR>\ican\<OS>\h
```

After that the header files can be included in the source files of your project.



In addition a macro has to be defined <u>above</u> the corresponding include statements. This macro activates the function definitions for the used operating system. In the table below macros for supported operating systems are listed.

Macro	Operating system
WIN_NT40	Microsoft Windows NT 4.0 Microsoft Windows 2000 Microsoft Windows ME Microsoft Windows XP
LINUX	Linux

Table 8: Macros for distinguishing settings for different operating systems

3.2.3 Base application

Like already described, our first application does not contain much complicated stuff. We only have implemented a main function, which checks the number of the program arguments and prints them out in text format if they are correct. For us it does not make sense to print such a simple source code at this place.



For sample code please see directory :

code / chapter3 / baseapp / baseapp.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.2.4 The all-in-one initialization function



device information structure

First we need to create an instance of the data structure 'dpm_dev_t'. We do this at the beginning of the main routine. We call this data structure the 'device information structure' because it contains nearly all information needed for the device initialization.

```
int main(int argc, char* argv[])
{
	// device information
	dpm_dev_t devInfo;
	// device descriptor
	int fd;

	// buffer for error messages
	char errBuff[ERROR_MSG_BUFFER_SIZE];
	// buffer for device driver version information
	long int drvId;
	// pointer to data buffer with version information
	unsigned char *drvPtr;
	// Firware request data buffer
	Message msg;

	// print header information
	printf ("****************************************\n");
	printf ("* Janz Automationsssysteme AG          *\n");
```

After that the data structure needs to be initialized with default values. For this function 'can_device_init' has to be used. Furthermore the device name pointer must be to the string storing the first program argument argv[1]. The device information structure contains a lot of possibilities to activate functionalities insight 'can_device_setup'. But for this simple example entering the device name will be enough. Entering the device name needs always be done at least. We will come to further functionalities of this function during the following steps in this chapter.

We are ready now to set up a device connection with function 'can_device_setup'. This function gets a pointer to the device information structure as first function parameter. The other parameters can be used to get error information from this function. As second argument a pointer to a string buffer can be transferred. The third argument specifies the size of this string buffer. If both parameters are given to the function, additional error information will be stored in this buffer, if the function fails. Of course, we will provide an error message buffer to the function to get the information what happened when the function returns a negative return value. The buffer 'errBuff' is defined at the beginning of function 'main' (see above). We strongly recommend providing a text buffer for error information to this function, because it saves a lot of time for error searching when device initialization has failed.

```
	// print arguments
	printf ("PROGRAM ARGUMENTS: \n");
	printf ("\tDevice access string     : %s\n", argv[1]);
	printf ("\n\n");
	printf ("PROGRAM ACTIONS: \n");

	printf ("\tInitializing device information structure ... ");
	// initialize device data structure with default values
	can_device_init_struct (&devInfo);
	// fill in needed data into the device data structure
	devInfo.devname = argv[1];
	// set version information for requested interface
	devInfo.dpm_dev_t_rev = DPM_DEV_T_REV;
	printf ("done\n");

	// call the one-in-all initialization function
	printf ("\tInitializing device ... ");
	fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
	if (fd < 0)
	{
		// error, abort program execution
		printf ("failed\n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                printf ("\tERROR: %s\n", errBuff);
                printf ("\n\tProgram aborted\n\n");
        } else {
                printf ("done\n");
        }

        // Congratulations, the device was initialized successfully and
        // is greeting you:
        printf ("\tHello, here is the Janz CAN device %s\n", argv[1]);
```

If the device connection could be established successfully, we print out a message which contains greetings from the device. Of course, we have to close this connection again at the end of our program. For this function 'can_device_close' is used.

```
        printf ("\n\n");
        printf ("PROGRAM SHUTDOWN: \n");

        // de-initialize the device again
        printf ("\tClosing device ... ");
        can_device_destroy (&devInfo);
        printf ("done\n");

        printf ("\n\nDone.\n\n");
        return (0);
}
```

At least let us a little bit explain, what happens in function 'can_device_setup'. This function contains a lot of additional features which the program writer not longer needs to write himself! We have already used some of these features despite the fact that we only have entered a device name without activating anything else in the device information structure:

- The function will check the version number of the device information structure. This ensures that applications do not run into trouble if libraries were exchanged on a system which is working with another device information structure (for example after a device driver update). If ‘can_device_setup’ returns a corresponding error, the application needs to be re-compiled against the actual libraries to be able to run with the installed device driver update.

- The function automatically checks whether the device was already opened before.

- The function checks whether the DPM queue sizes were set. If not, the function sets the values to usable default values and continues. Furthermore in each case the requested queue sizes are checked, whether they exceed the limits of resources.

- The device access string is checked if it contains valid a string which matches the requirements of device names for Janz CAN devices.

- The standard interface will be automatically initialized. The standard interface this is not only needed for CAN message transfer but also for transferring administrative information with the device. Therefore it is needed in any case.



For sample code please see directory :

code / chapter3 / devinit / devinit.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

3.2.5 Getting version information without message handler UPDATED

For getting device driver version numbers function ‘can_device_idvers’ is contained in the libraries. This function returns the wanted information in a 4-byte value. The following piece of code shows the using of the 'can_device_idvers' function and how to get the version information out of the returned value.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
// Congratulations, the device was initialized successfully and
// is greeting you:
printf ("\tHello, here is the Janz CAN device %s\n", argv[1]);

// Getting version of device drivers
printf ("\tGetting device driver version information ... ");
drvId = can_device_idvers(fd);
if (drvId < 0 )
{
       printf("failed\n");
       printf ("Progam aborted!\n\n");
       return (-3);
}
printf ("done\n");

// setting pointers to positions in data buffer
drvPtr = (unsigned char *)&drvId;

printf("\n------------------------------------------------------------");
printf("\nDPM Driver Release:           V%d.%d",drvPtr[1], drvPtr[0]);
printf("\nCarrier Board Driver Release: V%d.%d", drvPtr[3], drvPtr[2]);
printf("\n------------------------------------------------------------\n\n");
```

But version information can also be got directly from the device. It will contain type and version of the installed firmware in text format. Normally this works asynchronously with help of a messages handler thread. For this example we will do this exceptionally in a synchronous manner with a small trick, which is normally not correct, because it uses a great delay to ensure, that the version information was provided by the firmware. We will see some chapters below how to do this better in an asynchronous manner! For the moment we use this example to explain how device communication works in general.

Like already mentioned device communication is based on messages, messages sent from the application and messages returned by the firmware. Don't mistake this with CAN message frames which are completely different! That's the reason, why we talk about firmware requests and firmware response instead of messages when talking about device communication (please see also chapter 3.1.2).

Firmware version information needs to be requested from the firmware and it will be asynchronously returned as a firmware response. A queuing mechanism for firmware responses enables us to work like we do in this example. Building the corresponding firmware request is as easy as it could be in this case, because we need a request messages without additional data only having a command id set.

```
printf("\nCarrier Board Driver Release: V%d.%d", drvPtr[3], drvPtr[2]);
printf("\n------------------------------------------------------------\n\n");

// Getting firmware information
printf ("\tSending request for firmware version reading ... ");
msg.cmd = M_IDVERS;
msg.len = 0x0;
if (can_send(fd, &msg) < 0)
{
       printf("failed\n");
       printf ("Progam aborted!\n\n");
       return (-4);
}
printf ("successful\n");

// waiting 500ms
printf ("\tWaiting ... ");
sleep(1);
printf ("done\n");

printf ("\tReceiving firmware version data ... ");
// receiving data from firmware
if (can_recv(fd, &msg) < 0)
{
       printf("failed\n");
       printf ("Progam aborted!\n\n");
       return (-4);
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        printf ("successful\n\n");

        printf("Firmware version:\n%s\n",msg.data);
        printf("\n-----------------------------------------------------------\n\n");

        printf ("\n\n");
        printf ("PROGRAM SHUTDOWN: \n");
```

Firmware requests are data buffers of the structure defined by data structure 'Message'. We declared such a data structure already at the beginning of our main routine (please see above). For requesting firmware version information only the 'cmd' member needs to be set to the value defined by macro M_IDVERS. The 'len' member which specifies the length of additional data needs to be set to 0.

Normally inserting data into this structure is not necessary because the MITOP library comes with a lot of functions helping us to do this. But the MITOP library hasn't got functions included for so easy firmware requests like the one we currently want to use. So, we need to do the work 'manually' in this case. We'll come to MITOP library functions a little bit later in this document.

Sending firmware requests to the device is done with help of function 'can_send'. This function really transmits the contents of the buffer given in the second argument; nevertheless its name might say something other. Selecting this name for the function was not so lucky, but changing it now would cause more problems then it helps! The device connection to use is identified with help of the device descriptor value in argument one.

If the request was successfully sent the firmware will return the response as soon as possible. We inserted a delay of one second to be absolutely sure, that the response was present in the queue. Of course, responses will be returned much faster. We'll see this when using the asynchronous message handler later on in this document. After that we use function 'can_recv' to fetch the firmware response from a firmware internal queue and print out text on the screen. Following you will find the text issues produced by the sample program. The lines marked in blue colour give important information about the kind and the version of the installed firmware. Please note that this information is important to know for support requests. We recommend requesting and this information in each of your applications and saving it permanently somewhere on the host system for each application run. On the one hand, it signals that firmware communication is working; on the other hand it can give us very helpful information in case of support.

```
**************************************
* Janz Automationssysteme AG          *
* CAN Tutorial                         *
**************************************

GETTING VERSION INFORMATION WITHOUT MESSAGE HANDLER

PROGRAM ARGUMENTS:
        Device access string     : dpm_00


PROGRAM ACTIONS:
        Initializing device information structure ... done
        Initializing device ... done
        Hello, here is the Janz CAN device dpm0
        Getting device driver version information ... done
        Sending request for firmware version reading ... successful
        Waiting ... done
        Receiving firmware version data ... successful

------------------------------------------------------------
DPM Driver Release:           V3.2
Carrier Board Driver Release: V5.1
------------------------------------------------------------
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
Firmware version:
VMOD-ICAN3, ICANOS/2 CAN Interface.
Version: 1.26 Date: 01/05/16 17:02:17
(C) 1996 JANZ Computer AG, Paderborn. All rights reserved.
256 KByte ram

-----------------------------------------------------------

PROGRAM SHUTDOWN:
        Closing device ... done

Done.
```



For sample code please see directory :

code / chapter3 / idvers1 / idvers1.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

3.2.6 Initialize a device with message hander

We will now start describing some settings which can be made in the device information structure 'dpm_dev_t' for activating further included features. The device initialization function contains a possibility for automatically checking the firmware installed on the connected module. That helps customers who need to ensure, that a specific firmware release is installed on the device. Furthermore we provide you a description in this document (please see below) how to exchange a firmware programmatically when it was detected, that it has a wrong type or a wrong version number.

The device information structure contains a character pointer member 'fw_check_string' which can be set to a storage area containing a string, which must be contained in the version information string returned by an 'idvers' firmware response. You can set this string to each value you want. The content of this buffer is automatically compared with the version information string received from the installed firmware. Function ‘strstr’ is used for that. If the check string is found in the response, the firmware version is determined as valid. Otherwise device initialization will be aborted and an error message will be returned.

In our following example we want to have a CAL/CANopen firmware type installed. If this firmware was detected on the module, we will simply print out the release numbers. We will do more with it in one of the following chapters.

```
int main(int argc, char* argv[])
{
        // device information
        dpm_dev_t devInfo;
        // device descriptor
        int fd;
        // buffer for error messages
        char errBuff[ERROR_MSG_BUFFER_SIZE];
        // pointer to data buffer with version information
        unsigned short *fwVersionPtr;
        // minor version information
        unsigned short minor;
        // major version information
        unsigned short major;

                :
                :

        printf ("PROGRAM ACTIONS: \n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
printf ("\tInitializing device information structure ... ");
// initialize device data structure with default values
can_device_init_struct (&devInfo);
// fill in needed data into the device data structure
devInfo.devname = argv[1];
// set version information for requested interface
devInfo.dpm_dev_t_rev = DPM_DEV_T_REV;
// prepare device information structure for automatical firmware check
devInfo.fw_check_string = argv[2];

printf ("done\n");

// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// Congratulations, the device was initialized successfully and
// is greeting you:
printf ("\tHello, here is the Janz CAN device %s\n", argv[1]);

// setting pointer to start position in data buffer
fwVersionPtr = (unsigned short *)&devInfo.fw_version;

// simply treat integer 'fw_version' as array of two unsigned short values
minor = fwVersionPtr[0];
major = fwVersionPtr[1];

// Print firmware information
printf("\n------------------------------------------------------------");
printf("\nDetected firmware type      : %s", argv[2]);
printf("\nDetected firmware version   : %d.%d", major, minor);
printf("\n------------------------------------------------------------\n\n");

printf ("\n\n");
printf ("PROGRAM SHUTDOWN: \n");
```



The correct firmware version can only be found, if the string to which member 'fw_check_string' points to is contained in the M_IDVERS firmware response. Of course, you can use all strings you want. But we recommend using the strings from the table below, because these strings are surely contained in future releases of our firmware. The strings are case sensitive!

Check string	Firmware type
ICANOS	CAN Layer 2 firmware for all intelligent Janz CAN devices
CAL/CANopen	CANopen firmware for all intelligent Janz CAN devices

Table 9: Firmware check strings

3.2.7 Getting version information with message handler

Like you already probably know, communication with a Janz CAN device is bi-directional. Message handling works with a call-back mechanism. The program writer has to define a call-back function which is called by the library when a firmware response arrived and no default processing can be made. Configuring an application for using the call-back mechanism is as simple as it could be. You

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

only need to set the member pointer 'std_if_handler' of the device information structure to the function you want to use as call-back entry point. That's all. The library will call this function asynchronously each time a firmware response needs to be handled.

First we define the message handler call-back function. In our example this function only prints out a text message. We will fill this function with live in nearly all of our following chapters independently from the used firmware type.

```
void messageHandler (void *context, Message *msg)
{
        printf ("MESSAGE HANDLER: firmware response received\n\n");
}
```

After that we have to set member variable 'std_if_handler' for transferring the function pointer to the library. The library will setup the needed semaphores, transfer the semaphore handles to the device driver and creating and starting the needed thread.

```
        // fill in needed data into the device data structure
        devInfo.devname = argv[1];
        // set version information for requested interface
        devInfo.dpm_dev_t_rev = DPM_DEV_T_REV;

        // setting pointer to callback function
        devInfo.std_if_handler = messageHandler;
        devInfo.std_if_context = NULL;

        printf ("done\n");

        // call the one-in-all initialization function
        printf ("\tInitializing device ... ");
```

Additionally a context pointer can be set in member ‘std_if_context’ of the device information structure. This pointer value will be given as first parameter to the call-back function. This enables a software engineer to use the same call-back function for several devices.

If you want to use the 'Fast Host Interface' you have to use an own message handler for this. The initialization of the 'Fast Host Interface' message handler is similar to that described above. Simply use member 'fast_if_handler' and 'fast_if_context' for the message handler call-back function pointer and the context pointer. You can also work with both interfaces in parallel if you initialize both message handlers! We will describe the 'Fast Host Interface' in an own chapter later on in this tutorial.



For sample code please see directory :

code / chapter3 / idvers2 / idvers2.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



We now save the actual example program under another name 'template.c', because this will be the basis for all examples in the following chapters. So, each time when we start with a new programming lesson, we will use this program as a starting point. We have deleted some printf's from the code before to make the template not so talky like our former examples were. You can also use this template as starting point for your own applications.



For sample code please see directory :

code / chapter3 / template / template.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.2.8 Configuring DPM queues

In one of our former examples we got firmware version information without using a message handler simply by waiting a short time after sending the firmware request and receiving the firmware response by hand. This was possible only because the firmware queues responses. We call these queues ‘DPM queues’ because they reside in the DPM memory of the device in fact!

DPM queues can be configured regarding their sizes. Two queues are used for standard, one for sending firmware requests on the standard host interface and one for receiving firmware responses for the standard host interface. If nothing is done for DPM queue configuration, the library will set these both DPM queue sizes to default values. In most cases this is all what needs to be done!

Adapting DPM queue sizes is quite difficult! DPM queues are organized in DPM pages of 256 bytes in the firmware. There are a total number of 256 pages (64 Kbytes) available in the DPM. Several host interfaces (standard host interface, fast host interface, process picture for CANopen) are available for data transport and each interface needs some DPM pages at least for exchanging data with the host. The available memory size for DPM queues hardly depends on which interface is in use!

There are some rules, how the available size of DPM queues can be calculated. We provide you short information about available firmware interfaces and queue size calculation in the appendix of this document. You'll find more detailed information about the firmware structure in the corresponding chapter of our ICAN software reference manual.

To ensure, that wrong settings regarding DPM queue sizes won't reach the firmware, we implemented an additional function 'ican_check_dpm_resources'. This function compares the available DPM pages with the needed pages for requested queue sizes. It is automatically called each time when 'can_device_setup' is used to initialize the device connection. The function will return with an error code if wrong DPM queue sizes were detected. 'can_device_setup' also will abort with a corresponding error message in this case.

Using queues with different priority is possible only for the write direction which means, that firmware requests can be transferred from the host to the firmware with different priorities. Firmware requests can be received from the firmware only through one queue without priorities.

For default the members 'std_if_wbuffers_high' and 'std_if_wbuffers_low' are set to 0, which disables prioritized firmware request queues for the standard host interface. For default the medium queue is used. Prioritized queues can be activated by setting these members greater than 0. Please keep in mind, that possibly the settings regarding queue sizes of the standard queue needs to be changed to make memory available for the additional queues. Function 'ican_check_dpm_resources' will detect memory overflow and return with an error if the settings are wrong.

After a device initialization with prioritized queues the programmer can additionally use functions 'can_send_low' and 'can_send_high' to send firmware requests via the corresponding queues. Function 'can_send', of course, is still available and will still use the medium queue for transferring firmware requests

3.3 BusOn request

Before the Janz CAN device is able to send CAN frames to or receive CAN frames from the CAN bus, the device has to be switched physically on. Because switching the module on depends on which firmware is installed on the module, we have to describe some firmware-specific stuff at this point.

Switching the Janz CAN module on is done in two steps. First a bit rate has to be set, second the command for switching the module on has to be sent to the firmware. For both a corresponding firmware request is needed.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

The libraries delivered with the software development kit contain functions which make it easier for a programmer to fill in data into the 'Message' data structure. Depending on which request message was used the firmware may send back a corresponding response to the message handler or not. Depending on what firmware version you want to use, you have to continue with the corresponding chapter below.

3.3.1 'BusOn' request with ICANOS Firmware

Creating a firmware request for setting the bit rate on the CAN device which has the ICANOS firmware installed is done with the help of MITOP function 'IcWriteBtrBCAN'. It helps us filling correct data into a firmware request buffer. Bit rates a specified with macros defined in file 'bcan.h' located in directory 'mitop/h' in the installation path of your software development kit. Macros are provided for several standard bit rates used for CAN bus communication. In fact, the values used for SJA1000 CAN controller configuration are transmitted to the device. If a customer would like to work with an own bit rate, he can use a custom value instead of the macro.

```
// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// create message for baud rate settings
printf ("\tSetting baud rate to 1000 Bit/s ... ");
// set baud rate
IcWriteBtrBCAN(&msg, BTR_1MB);
// send 'baud rate' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot set baud rate\n");
        printf ("Progam aborted!\n\n");
        return (-5);
}
printf ("done\n");

// create message for 'BusOn' message
printf ("\tSwitching bus on ... ");
IcBusOnBCAN(&msg);
// send 'BusOn' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot change bus status\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}
printf ("done\n");

printf ("\n\n\tReady for sending and receiving data ...\n\n");
```

Creating the firmware request for switching the device on is quite similar. For this the MITOP function 'IcBusOnRequest' is available for inserting request's data into the buffer. After sending the 'BusOn' request to the firmware, the device is ready for receiving and sending CAN frames.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter3 / buson / icanos / buson.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

3.3.2 'BusOn' request with CAL/CANopen firmware

If the CANopen firmware is installed on the Janz CAN device only one command message is used for both, setting the bit rate and switching the module on. For filling the data into the 'Message' data structure function 'LsBusOnReq' is used. The bit rate macros are the same like also used for ICANOS.

```
// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// create data for 'BusOn' message
printf ("\tSetting bitrate and switching bus on ... ");
LsBusOnReq(&msg, BTR_1MB);
// send message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot change bus status\n");
        printf ("Progam aborted!\n\n");
        return (-4);
}
printf ("done\n");

printf ("\n\n\tReady for sending and receiving data !\n\n");
```

For creating a request to switch the device off function 'LsBusOffReq' can be used. This function has no additional parameter except a pointer to the data buffer.

You can check whether the device was successfully connected with the bus with help of the LEDs located on the device. After the 'LsBusOnReq' call the red LED should be off, only the green LED should be on! This will change again after calling sending the BusOff request.



For sample code please see directory :

code / chapter3 / buson / canopen / buson.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

3.4 Device reset

A device reset is triggered each time the 'can_device_setup' function is called. So the programmer normally does not need to reset a device manually. However, if you want to perform a manual device reset during the program run you can simply use the command 'can_reset' for this.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Please be aware that this function does not use a firmware request to do the reset! Resetting a device is a device driver functionality. Therefore no data needs to be sent to the device with help of function 'can_send' in this case.

```
// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// Issue message, that user has to press the RETURN key
printf ("\tPlease press [RETURN] key to start ...\n");
getchar();

// Waiting some seconds to give the user the chance to see
// the LEDs blinking ...
printf ("\tWaiting some seconds ");
for (i=0; i<3; i++)
{
        printf (".");
        sleep (1);
}
printf (" done\n");

// Initialite device reset
printf ("\tResetting device ... ");
can_reset(fd);
printf ("done\n");

// Waiting some seconds to give the user the chance to see
// the LEDs blinking ...
printf ("\tWaiting some seconds ");
for (i=0; i<3; i++)
{
        printf (".");
        sleep (1);
}
printf (" done\n");

// shutting down the program
printf ("\n\n");
printf ("PROGRAM SHUTDOWN: \n");
```

The program starts with some text issues and waits until the user has pressed the [RETURN] button. After that again some seconds delay were programmed to give the user the possibility to take a look to the LEDs of the CAN device. You should be able to see the LEDs blinking after the reset request was sent to the CAN device.

After the reset a delay of some seconds was build in again before the program shuts down.



After a manual reset the firmware and the device driver status information are inconsistent. The application needs to close its currently used device connection and re-establish it again to ensure that device driver and firmware have the same status information! We recommend to stop all threads (message handler) and to run through the complete initialization procedure.

In the example above the program stops after device reset. It's your turn to implement a reconnection mechanism!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter3 / reset / reset.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

3.5 Using the internal CAN bus termination

If the Janz CAN device is placed on one end of the CAN bus, the bus can be terminated with the termination resistor which is integrated in the device.

Switching the internal termination on and off is quite simple. The programmer needs to create the corresponding firmware requests for that and send them to the device. In addition MITOP function 'SwitchCanTermination' can be used to fill in data into the request buffer. 0 and 1 are valid values as second parameter of this function standing for on and off.

```
// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// create firmware request for switching the CAN bus termination
printf ("\tSwitch CAN termination on ... ");
// create firmware request for switching on the CAN bus termination
SwitchCanTermination (&msg, 1);
// send message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot switch on CAN bus termination\n");
        printf ("Progam aborted!\n\n");
        return (-5);
}
printf ("done\n");

printf ("\n\n");
printf ("PROGRAM SHUTDOWN: \n");

// create firmware request for switching the CAN bus termination
printf ("\tSwitch CAN termination off ... ");
SwitchCanTermination (&msg, 0);
// send message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot switch off CAN bus termination\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}
printf ("done\n");

// de-initialize the device again
printf ("\tClosing device ... ");
```

After the request was transferred to the CAN device, the firmware will switch on or off the internal termination resistor. Unfortunately this is nothing which can be seen physically. The only possibility to get information about whether this was successful or not, is sending some messages over the bus

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

without having an extra termination resistor at the end where the Janz CAN device is located. With the termination set, this should work without problems. Otherwise error frames will occur.

The command 'SwitchCanTermination' is available for all firmware versions and should be used before the 'BusOn' request!



For sample code please see directory :

code / chapter3 / termination / termination.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

4 Standard host interface

This chapter describes functions which can be used with the ICANOS firmware installed on the Janz CAN module. The ICANOS firmware does not support a higher layer protocol. It only provides functions for sending and receiving simple CAN frames, filtering of CAN frames and sending and receiving of some administrative information to and from the device.

We hope, you already read something about our first skeleton application in the chapters above. We will start with exactly this skeleton and show you how CAN frames can be send and received in your application using the standard host interface.

Receiving events from the bus is done in the message handler call-back function. So we will expand the existing call-back function code to be informed when events like bus error messages occur and to handle them.

4.1 Sending CAN frames

For sending CAN frames we will define us a function 'icanos_send'. This function will have the following parameter values:

- a file descriptor specifying the device connection
- the CAN identifier of the frame
- a boolean value containing the information whether the RTR bit has to be set
- a value which specifies the manner of sending
- an identifier defined from us
- a pointer to an data buffer
- the length of data stored in this buffer

Our function will fill the parameter values into a firmware request data structure with help of function 'IcSendReqBCAN'. After that we are using function 'can_send' again to transfer the request to the firmware of the device.

```
////////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
////////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    icanos_send
//
// PURPOSE:
//    send function for VMOD-ICAN3 with ICANOS firmware
//
// PARAMETERS:
//
//
// RETURN VALUE:
//    int - return code
//     0  - no error
//    -1  - wrong spec identifier
//    -2  - wrong message length
//    -3  - no data buffer specified
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
//    -4  - can_send failed
//    -5  - invalid value for file descriptor fd

int icanos_send (     int fd,
                      unsigned short id,
                      unsigned int rtr,
                      unsigned char spec,
                      unsigned short myReqId,
                      unsigned char len,
                      unsigned char *data)
{
        // Message buffer
        Message msg;
        // DEscriptor
        WORD_t lDesc;
        // RTR bit value
        unsigned char cRtr;

        // check file descriptor
        if (fd < 0)
        {
                // print error message
                printf ("ERROR: wrong file descriptor (fd=%i)\n", fd);
                // return error
                return -5;
        }

        // check spec identifier
        if (spec > 3)
        {
                // print error message
                printf ("ERROR: wrong spec identifier %i, should be < 4\n", spec);
                // return error
                return -1;
        }

        // check data buffer
        if (data == NULL)
        {
                // print error message
                printf ("ERROR: no data buffer specified\n");
                // return error
                return -3;
        }

        // set RTR value
        if (rtr)
        {
                cRtr = 1;
        } else {
                cRtr = 0;
        }

        // create descriptor value
        lDesc = (id << 5) | (cRtr << 4) | len;

        // fill message structure
        IcSendReqBCAN (&msg, spec, myReqId, lDesc, data);

        // send CAN frame
        if (can_send (fd, &msg) < 0)
        {
                // print error message
                printf ("ERROR: can_send failed\n");
                // return error
                return -4;
        }

        // return success
        return 0;
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

First the parameter values should be checked in function ‘icanos_send’. That gives us the chance to reuse the code of this function also in other projects where it is not so easy to keep the survey whether parameters are correct or not.

After that the values for the CAN identifier, the RTR bit and the message length are copied to defined positions in a 16-bit descriptor value. Function 'IcSendReqBCAN' takes the function parameters and the descriptor value to fill the firmware request buffer. After that function 'can_send' transfers the contents to the ICANOS firmware which is responsible for transferring the needed data to the CAN controller.

Function 'IcSendReqBCAN' implements several possibilities to transmit CAN frames to the firmware. CAN frames can be transmitted with or without requesting a confirmation and a copy of the data to be sent back. For that the user defined value ‘myReqId’ can be used to uniquely mark a message. The value will also be included in confirmation messages so that message confirmations from the firmware can be uniquely assigned to messages sent from within the application.

Parameter spec defines how the firmware reacts:

- spec = 0; message is simply sent on the bus, parameter ‘myReqId’ is ignored.
- spec=1; a confirmation response is requested for each CAN message. The confirmation response contains the information whether the message was successfully sent. Parameter ‘myReqId’ can be used to assign confirmations to messages.
- spec=2; a complete copy of the sent message will also be returned to the message handler. This facility can be helpful for debugging purposes or for tracing sent and received messages in one and the same file. Parameter ‘myReqId’ will be ignored.
- spec=3; both, a confirmation and a copy message, will be returned.

Our example sends a message four times, one for each ‘spec’ value. We also implemented the corresponding handling in the message handler call-back function despite the fact we did not talk about how to write message handler code. This will be done in one of the next chapters. In the meantime take it like it is and be happy if it works. 

```
// create message for 'BusOn' message
printf ("\tSwitching bus on ... ");
IcBusOnBCAN(&msg);
// send 'BusOn' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot change bus status\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}
printf ("done\n");

printf ("\n\n\tReady for sending and receiving data ...\n\n");

// fill data buffer
memset (data, 0, 256);
data[0] = 0xaa;
data[1] = 0xbb;
data[2] = 0xcc;

id = 221;
rtr = FALSE;
myReqId = 92;
len = 3;

// ICANOS data transfer
printf ("ICANOS DATA TRANSFER:\n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
// use all available spec values ...
for (spec=0; spec<4; spec++)
{
        // set the first data byte to the used spec value
        data[0] = (unsigned char)spec;
        printf ("\n\t==> SENDING MESSAGE: id = %i\n", id);
        printf ("\t\t(RTR = %i, myReqId = %i, len = %i, spec = %i)\n",
                        rtr, myReqId+spec, len, spec);

        // invoke function icanos_send ...
        if (icanos_send(fd, id, rtr, spec, myReqId+spec, len, data) < 0)
        {
                printf ("\tERROR:\n");
                printf ("\tICANOS initialisation failed!\n");
                printf ("\tProgam aborted!\n\n");
                return (-3);
        }
        printf ("\t\tSuccessfully sent\n");

        // wait a short time between sending the separate messages
        sleep (2);
}

// wait a short time
sleep (5);

printf ("\n\n");
printf ("PROGRAM SHUTDOWN: \n");
```



For sample code please see directory :

code / chapter4 / icanos_send / icanos_send.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.2 How to write message handler code NEW!

We talked a couple of time about it but we never completely said, how to work with it: The message handler call-back function! Now we will make up for this!



Firmware services

The firmware exists of several parts which all send different firmware responses to the application. We call these parts ‘firmware services’. Each service uses several numbers to signal of which type a firmware response is. This number is used as identifier in the message handler. It is always contained in the cmd member of a ‘Message’ data structure. Therefore the firmware response type can always be determined by using the cmd value.

Additionally to this document also the ICAN3 Software Reference Manual is available. This manual describes communication between device and host from the firmware point of view. It also defines the service identifiers and the format of additional data coming in a firmware response of a specific type. Please see also this document for more detailed information about firmware services and firmware responses generated by them.

We recommend using a big ‘switch’ statement in the message handler to handle several service identifiers. The advantage is that you easily can reuse code, that you either implemented in other applications or that we provide you in examples. Below we printed an example for a quite simple message handler only handling the IDVERS response of our example above in a special manner. All other responses will be handled in a general way. We provide you function ‘showMessage’ to convert firmware response data into text format.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

It is absolutely impossible to write a path for each firmware response which might be returned by the firmware at any time in the future. We recommend writing a path only for those responses, which you would like to handle in a special manner. But it is surely a good idea to save the result of function 'showMessage' in the default statement at any place (a log file or something similar). So, you'll recognize when error indications will be signalled which you did not expect.

Adding a new path for a service identifier always work the same way. You simply add a 'case' statement with the corresponding macro as argument. Please do not use the numeric identifier values itself. It might lead to incompatibilities in the future! You'll find these macros in files 'dpm.h', 'mitop.h' and 'canopen.h'. Such macros always start with the prefix 'M_' followed by a short cut describing the service which generates it. The ICAN3 Software Reference Manual contains a table listing all available service identifiers, their corresponding macro and a short description.

```
void messageHandler (void *context, Message *msg)
{
        // extracting the command specifier of firmware response
        int     spec = msg->cmd;
        // setting the pointer to the first data byte in the response
        BYTE_t *d   = &msg->data[0];

        printf ("\t==> MESSAGE HANDLER:\n");

        // Implementing a filter for all wanted response messages and
        // handle them each in an own path of the following switch statement.
        switch (spec)
        {
        // IDVERS response, returning firmware version information
        case M_IDVERS:
                showMessage("", msg);
                break;

        // default handling, needs to be deleted, if unwanted response
        // shall be ignored !!
        default:
                printf ("\t\tUnhandled firmware request, spec=0x%x\n\n", spec);
                showMessage ("\t\tDATA: ", msg);
                break;
        }
}
```

The following code snippet can be used to add handling for message confirmations generated when spec 1 or 3 are used for function 'icanos_send':

```
        // Implementing a filter for all wanted response messages and
        // handle them each in an own path of the following switch statement.
        switch (spec)
        {
        // IDVERS response, returning firmware version information
        case M_IDVERS:
                showMessage("", msg);
                break;

        case M_BCAN_TX_con:
                printf ("\t\tMessage confirmation\n");
                printf ("\t\treqId=%i, result=", *(unsigned short *)&d[0]);
                switch (d[2])
                {
                case 0x00:
                        printf ("success\n");
                        break;
                case 0x01:
                        printf ("error: controller transmit buffer full\n");
                        break;
                case 0x02:
                        printf ("error: transmit failed\n");
                        break;
                default:
                        printf ("unknown value, res=%i\n", d[2]);
                        break;
                }
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        break;

// default handling, needs to be deleted, if unwanted response
// shall be ignored !!
default:
        printf ("\t\tUnhandled firmware request, spec=0x%x\n\n", spec);
        showMessage ("\t\tDATA: ", msg);
        break;
```



For sample code please see directory :

code / chapter4 / icanos_send / icanos_send.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.3 Receiving CAN frames UPDATED

If the device receives a CAN frame from the bus, the firmware will put it as an event message into a firmware response queue. An interrupt is used to notify the host, that firmware response messages need to be handled. Handling is already programmed in the library code. So, the library will call the message handler also for received CAN frames if it was setup.

Therefore the only thing we need to do is adding a corresponding path into the message handler function, which extracts the CAN frame data out of the firmware response data structure. That's all.

```
// response message for each received CAN frame
case M_BCAN_RX_ind:
        // get id value
        id = d[2]*8 + (d[3] >> 5);
        // get message length
        len = (d[3] & 0x0F);
        // get rtr bit
        rtr = ((d[3] & 0x10) >> 4);

        // printf message data
        printf ("Msg No.: %i\n", ++cnt);
        printf ("ICANOS: message received!\n");
        printf ("        message id = %i\n", id);
        printf ("        message length = %i Bytes\n", len);
        printf ("        message RTR bit = %i\n", rtr);
        // check message length
        if (len > 0)
        {
                // printf message data
                printf ("        message data = ");
                for (i=0; i<len; i++)
                {
                        printf ("%02X ", d[i+4]);
                }
                printf ("\n");
        }
        break;
```

In our example we simply extract message data and print it out. Of course, your work starts where we stopped writing our sample code. Here is the location where you need to implement your own CAN message handling!

Variable id contains the CAN message identifier, len get the data length of the CAN frame. Value rtr signals, whether a Remote Transmission Request was received.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter4 / icanos_recv / icanos_recv.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.4 Firmware error condition handling UPDATED

The firmware will signal error conditions also via firmware response messages. Adding a corresponding path to the message handler enables error handling in your applications.

```
// indication, that an event has occured in the firmware
case M_BCAN_EVENT_ind:
        // get message length
        len = (d[3] & 0x0F);
        // print error message
        printf ("ERROR INDICATION:\n");
        printf ("data length : %i Byte(s)\n", len);
        // check length
        if (len > 0)
        {
                // check error code in first data byte
                if (d[4] & 0x01)
                {
                        // printf error message
                        printf ("error type : error interrupt\n");
                }

                // check error code in first data byte
                if (d[4] & 0x02)
                {
                        // printf error message
                        printf ("error type : overrun interrupt\n");
                }

                // check error code in first data byte
                if (d[4] & 0x04)
                {
                        // printf error message
                        printf ("error type : interrupts lost\n");
                }

                // check error code in first data byte
                if (d[4] & 0x08)
                {
                        // printf error message
                        printf ("error type : send queue full\n");
                }

                // check error code in first data byte
                if (d[4] & 0x10)
                {
                        // printf error message
                        printf ("error type : CAN bus error\n");
                }

                // check whether additional data was sent
                if (len > 1)
                {
                        // printf message data
                        printf ("             message data = ");
                        for (i=1; i<len; i++)
                        {
                                printf ("%02X ", d[i+4]);
                        }
                        printf ("\n");
                }
        }
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        break;
```

The data length of an error message is always greater than 1. The first data byte contains a value which specifies the type of the error condition detected in the firmware. The table below lists valid values for this specifier with a short description. You will find more detailed information about error conditions in the ICAN3 Software Reference Manual.

Name	Contents of data byte 1	Description
Error interrupt	0x01	Occurs, when the error counter reaches warning level
Overrun interrupt	0x02	Occurs, when messages cannot be handled fast enough from the CPU of the VMOD_ICAN module.
Interrupts lost	0x04	Occurs, when interrupts occur, which cannot be handled by the application.
Send queue full	0x08	Signals, that the send queue has reached its maximum size.
Bus error	0x10	Signals a common CAN bus error, additional bus error handling features are described in an own chapter

Table 10: Firmware error conditions

In cases of an error interrupt (0x01) and a bus error (0x10) further information describing the error is stored inside the message.



For sample code please see directory :

code / chapter4 / icanos_event / icanos_event.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.5 Message filtering

Message filtering can be divided into two main parts, software message filtering and hardware message filtering. Software message filtering is a per message mechanism implemented in the ICANOS firmware. If a software filter was defined, each received CAN frame is compared with the filter settings stored in the firmware memory. Only if acceptance is signalled the frame will cause further actions in the firmware. Hardware message filtering is realized by the CAN controller which is normally a Philips SJA1000. In this case special registers of the CAN controller are configured, so that all messages are treated like they are not received in case the filter settings don't signal acceptance.

Software filtering is more convenient than the hardware filtering, because the programmer is able to define several ranges or single identifiers which have to be filtered out. But each frame, also unwanted frames need to be received from the firmware. So the firmware has to work with the frame even if they are filtered out.

Hardware filtering is a little bit more difficult because the programmer has to work directly with the acceptance code and the acceptance mask register of the SJA1000 CAN controller. So he needs more knowledge about how the controller uses these registers. Furthermore only one or two ranges of identifiers can be defined for messages which should be received. In this case all messages will be ignored except the messages for which a positive acceptance filter was defined in the CAN controller's

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

registers. The advantage in comparison with software filtering is that the firmware does not get notified about messages which are not accepted. So hardware filtering safes firmware resources!

The chapters below will explain a little bit more in detail, how hardware and software filtering works and what functions can be used to implement the corresponding filter mechanism.

4.5.1 Software message filtering

For activating software message filtering some few functions are contained in the MITOP library which help us creating corresponding firmware requests. We only need to give wanted filter settings to these functions and send the resulting firmware requests to the device. For default all identifiers are configured to be received. We will use the filter configuration functions to setup some identifier ranges not to be received.

The table below list available filter configuration functions contained in the library. You'll find further information in the ICAN3 Software Reference Manual.

Function	Description
IcOpenAfil	Enables the message receiving for one CAN identifier
IcCloseAfil	Disables the message receiving of one CAN identifier
IcRangeOpenAfil	Enables the message receiving for a complete range of CAN identifiers
IcRangeCloseAfil	Disables the message receiving for a complete range of CAN identifiers
IcSetAfil	Sets the filter mask for message receiving of one CAN identifier (important for 'Fast Host Interface')
IcRangeSetAfil	Sets the filter mask for message receiving of a complete range of CAN identifier (important for 'Fast Host Interface')

Table 11: Message filtering configuration functions

```
// create message for 'BusOn' message
printf ("\tSwitching bus on ... ");
IcBusOnBCAN(&msg);
// send 'BusOn' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot change bus status\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}
printf ("done\n");

// set message filter
printf ("\tIcRangeCloseAfil: Setting filter options for id range [221; 230] ... ");
IcRangeCloseAfil (&msg, 221, 230);
// send 'CloseFilter' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Cannot set filter mask\n");
        printf ("Progam aborted!\n\n");
        return (-3);
}
printf ("done\n");
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
// set message filter
printf ("\tIcRangeSetAfil: Setting filter options for id range [221; 230] ... ");
IcRangeSetAfil (&msg, 221, 230, 0);
// send 'CloseFilter' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Cannot set filter mask\n");
        printf ("Progam aborted!\n\n");
        return (-3);
}
printf ("done\n");

// set message filter
printf ("\tIcCloseAfil: Setting filter options for id range [221; 230] ... ");
for (id=221; (id<=230) && (rc>0); id++)
{
        IcCloseAfil (&msg, id);
        // send 'CloseFilter' message
        rc = can_send(fd, &msg);
}
if (rc<=0)
{
        printf ("failed\n");
        printf("ERROR : Cannot set filter mask\n");
        printf ("Progam aborted!\n\n");
        return (-3);
} else {
        printf ("done\n");
}

// set message filter
printf ("\tIcSetAfil: Setting filter options for id range [221; 230] ... ");
for (id=221; (id<=230) && (rc>0); id++)
{
        IcSetAfil (&msg, id, 0);
        // send 'CloseFilter' message
        rc = can_send(fd, &msg);
}
if (rc<=0)
{
        printf ("failed\n");
        printf("ERROR : Cannot set filter mask\n");
        printf ("Progam aborted!\n\n");
        return (-3);
} else {
        printf ("done\n");
}


printf ("\n\n\tReady for sending and receiving data ...\n\n");

// Wait a long time while data is received in message handler
// callback function
sleep (120);
```



For sample code please see directory :

code / chapter4 / icanos_swfilter / icanos_swfilter.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

We use four different methods in our example to define one and the same filter condition. We do this to show you how the different functions can be used. Of course, this is only for demo purposes. You should avoid doing this, because it will define several identical filter rules on the device. Each needless filter rule will waste performance in the firmware, because it needs one entry in a list which needs to be handled each time when a CAN frame has been received. The device cannot summarize

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

several overlapping areas defined in filter rules; it will take the rules like they are. The programmer is responsible for defining meaningful and performance optimized filter settings.

4.5.2 Hardware message filtering

The acceptance filter of the SJA1000 CAN controller compares the received CAN identifier with the acceptance filter register contents and decides whether a message should be accepted or not. Only in the event of a positive acceptance test, the complete message is forwarded to the CAN module's firmware.

The big advantage of this method is that the firmware is avoided from working with messages, which should be not accepted. This saves firmware resources. The disadvantage of this quite simple mechanism is, that only one (single filter mode) or two (dual filter mode) coherent areas of CAN identifiers can be defined, for which messages can pass through the filter. It is not possible to create several groups of identifiers for which messages are accepted and it is difficult to filter only some single IDs. For this the software filter mechanism has to be used. Software and hardware filter can be used in parallel! But it does not make much sense to define a software filter for a message identifier which can never pass through the hardware filter due to fact that the hardware filter mechanism will be the first filter to pass in any case.

With the help of the acceptance filter the CAN controller is able to allow passing of received messages only when the identifier bits of the received message are equal to the predefined ones within the acceptance filter registers. The acceptance filter is defined with an acceptance code (ACR = acceptance code register) and an acceptance mask (AMR = acceptance mask register). The bit patterns of CAN frames to be received are defined within the acceptance code. The corresponding acceptance mask allows defining certain bit positions to be 'don't care'. At the beginning (after a reset of the CAN module) all CAN message identifiers are defined to be accepted for default which means in fact, that the hardware filtering is disabled.

For enabling hardware acceptance filtering function 'IcWriteExtAcmBCAN' is used. This function gets a message buffer, a mode value, the value of the acceptance code register and the value of the acceptance mask register as function parameters. The function converts this data into the correct firmware request and puts the data into the buffer specified by the first function parameter. After that the firmware request data needs to be send to the CAN module's firmware with help of function 'can_send'.

Two different filter modes are selectable, the 'single filter mode' and the 'dual filter mode'. You can use either 0 (dual filter mode) or 1 (single filter mode) for setting the mode parameter in 'IcWriteExtAcmBCAN'. The meaning and the effect of the acceptance code and acceptance mask depends on the selected mode and the received identifier (extended or standard frame). Both is described in own chapters in the appendix of this document.

During the first steps we will first build a small application which shows you the use of function 'IcWriteExtAcmBCAN'. After that we will discuss the meaning and the results of different parameter values for receiving standard CAN identifiers in the two available filter modes.

In a second step we will expand the example application to be able to receive extended identifiers. This can only be done with the 'Fast Host Interface'. So we need to add some function calls to the existing example program for supporting this interface. Afterwards we discuss again how messages will be received in the different filter modes using extended identifier.

4.5.2.1 Hardware message filtering for standard CAN frames

For our example we take the code described in chapter ‘Receiving CAN messages’ (please see Softlink below) as basis. Here message reception is already implemented and we can fully concentrate on how to define hardware filter settings.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter4 / icanos_recv / icanos_recv.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

After device initialization we add a call of function 'IcWriteExtAcmBCAN'. It is not so difficult to setup the function call, but it is much more difficult to understand, how to set the values for ACR and AMR so that the wanted filter condition is reached.

```
// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// Congratulations, the device was initialized successfully and
// is greeting you:
printf ("\tHello, here is the Janz CAN device %s\n", argv[1]);

// setting mode
mode = 0; // dual filter mode

//      .--------------- Byte ACR[0], Bits 10-7, CAN identifier, filter 1
//      !.-------------- Byte ACR[0], Bits 6-3, CAN identifier, filter 1
//      !!.------------- Byte ACR[1], Bits 2-0, CAN identifier, RTR bit, filter 1
//      !!!.------------ Byte ACR[1], Bits 7-4, data byte 1, filter 1
//      !!!!.----------- Byte ACR[2], Bits 10-7, CAN identifier, filter 2
//      !!!!!.---------- Byte ACR[2], Bits 6-3, CAN identifier, filter 2
//      !!!!!!.--------- Byte ACR[3], Bits 2-0, CAN identifier, RTR bit, filter 2
//      !!!!!!!.-------- Byte ACR[3], Bits 3-0, data byte 1, filter 1
//      !!!!!!!!
//      vvvvvvvv
ACR = 0x00000000;

//      .--------------- Byte AMR[0], Bits 10-7, CAN identifier, filter 1
//      !.-------------- Byte AMR[0], Bits 6-3, CAN identifier, filter 1
//      !!.------------- Byte AMR[1], Bits 2-0, CAN identifier, RTR bit, filter 1
//      !!!.------------ Byte AMR[1], Bits 7-4, data byte 1, filter 1
//      !!!!.----------- Byte AMR[2], Bits 10-7, CAN identifier, filter 2
//      !!!!!.---------- Byte AMR[2], Bits 6-3, CAN identifier, filter 2
//      !!!!!!.--------- Byte AMR[3], Bits 2-0, CAN identifier, RTR bit, filter 2
//      !!!!!!!.-------- Byte AMR[3], Bits 3-0, data byte 1, filter 1
//      !!!!!!!!
//      vvvvvvvv
AMR = 0x00000000;

// create message for configuring hardware acceptance filtering
printf ("\tSetting hardware filter parameters ... ");
IcWriteExtAcmBCAN(&msg, mode, ACR, AMR);
// send message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Cannot change bus status\n");
        printf ("Progam aborted!\n\n");
        return (-2);
}
printf ("done\n");

// create message for baud rate settings
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
printf ("\tSetting baud rate to 1000 Bit/s ... ");
// set baud rate
IcWriteBtrBCAN(&msg, BTR_1MB);
// send 'baud rate' message
if (can_send(fd, &msg) <= 0)
{
       printf ("failed\n");
       printf ("ERROR : Cannot set baud rate\n");
       printf ("Progam aborted!\n\n");
       return (-5);
}
printf ("done\n");
```

We try to explain how to setup both values for different filtering purposes in the following chapters. So, please continue with the next chapters for seeing how this example works...



For sample code please see directory :

code / chapter4 / icanos_hwfilter / icanos_hwfilter.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.5.2.2 Filtering standard CAN frames in single filter mode

First we will describe how to set the parameter values ACR and AMR of function 'IcWriteExtAcmBCAN' for filtering **standard identifier** in **single filter mode**. Please refer to chapter 'Single filter mode' in the appendix for getting more detailed information about the mode itself.

The acceptance code value and the acceptance mask value do not correspond directly to the real register values we want to set. The value bytes are stored exactly vice versa in their byte order. So if we set byte 0 of the acceptance code value this data will end in byte 3 of the corresponding register after the message was transferred to the firmware. Byte 1 of the variable will end in byte 2 of the register ... etc.

In this example we want to configure the CAN controller for receiving only two standard identifiers, 256 and 257. We will configure the acceptance mask first, so that the data bytes of the CAN frame and the RTR bit are set to 'don't care'. Furthermore we want also bit 1 of the CAN identifier to be 'don't care'. This results in an acceptance mask value of 0x003FFFFF. Using this mask will cause the controller only to take bits 2-11 of an 11-bit CAN identifier (without RTR bit and data bits) for acceptance filter comparisons.

```
// setting mode
mode = 1; // single filter mode

//      .--------------- Byte ACR[0], Bits 10-7, CAN identifier, filter 1
//      !.-------------- Byte ACR[0], Bits 6-3, CAN identifier, filter 1
//      !!.------------- Byte ACR[1], Bits 2-0, CAN identifier, RTR bit, filter 1
//      !!!.------------ Byte ACR[1], Bits 7-4, data byte 1, filter 1
//      !!!!.----------- Byte ACR[2], Bits 10-7, CAN identifier, filter 2
//      !!!!!.---------- Byte ACR[2], Bits 6-3, CAN identifier, filter 2
//      !!!!!!.--------- Byte ACR[3], Bits 2-0, CAN identifier, RTR bit, filter 2
//      !!!!!!!.-------- Byte ACR[3], Bits 3-0, data byte 1, filter 1
//      !!!!!!!!
//      vvvvvvvv
ACR = 0x003FFFFF;

//      .--------------- Byte AMR[0], Bits 10-7, CAN identifier, filter 1
//      !.-------------- Byte AMR[0], Bits 6-3, CAN identifier, filter 1
//      !!.------------- Byte AMR[1], Bits 2-0, CAN identifier, RTR bit, filter 1
//      !!!.------------ Byte AMR[1], Bits 7-4, data byte 1, filter 1
//      !!!!.----------- Byte AMR[2], Bits 10-7, CAN identifier, filter 2
//      !!!!!.---------- Byte AMR[2], Bits 6-3, CAN identifier, filter 2
//      !!!!!!.--------- Byte AMR[3], Bits 2-0, CAN identifier, RTR bit, filter 2
//      !!!!!!!.-------- Byte AMR[3], Bits 3-0, data byte 1, filter 1
//      !!!!!!!!
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
//      vvvvvvvv
AMR = 0x20000000;
```

In the next step we will set the acceptance code, so that bit 9 of the CAN identifier has always to be set in a message which will be accepted. All bits which are set to 'don't care' in the acceptance mask can have any value in the acceptance code because these bits are not used for the bit comparisons of the acceptance filter. We will use 0 for these bits in our example. This results in value of 0x20000000 for the acceptance code.

Result:

Only messages will pass the filter which have bit 9 set in their identifier while the RTR bit, bit 1 of the identifier and all data bits are not used in the filter comparisons. This means that only messages with identifier 256 and 257 can pass the filter. It does not matter whether these messages are RTR messages or not! That's exactly what we want to configure in this chapter.

4.5.2.3 Filtering standard CAN frames in dual filter mode

In this chapter we want to use the **dual filter mode** (mode = 0) to define two separate **standard identifier** groups which will be accepted. The first group consists of the identifiers 256 and 257, so that we can partially take the settings already made for the single filter mode above. The second group contains the identifiers 248 up to 251. For this group we will define completely new settings.

We have to pay attention for the exceptional positioning of the bits in the controller's registers! The bit settings are not equal for the first and the second identifier group. The first group will use the bits of the CAN identifier, the RTR bit and the bits of the first data byte. The second filter is only able to use bits of the CAN identifier and the RTR bit for its acceptance comparisons. We will set the bits of the data byte in filter 1 and the RTR bit of both filter to 'don't care' in the acceptance mask register. This results in an acceptance mask register value of 0x003F007F.

```
// setting mode
mode = 0; // dual filter mode

//      .--------------- Byte ACR[0], Bits 10-7, CAN identifier, filter 1
//      !.-------------- Byte ACR[0], Bits 6-3, CAN identifier, filter 1
//      !!.------------- Byte ACR[1], Bits 2-0, CAN identifier, RTR bit, filter 1
//      !!!.------------ Byte ACR[1], Bits 7-4, data byte 1, filter 1
//      !!!!.----------- Byte ACR[2], Bits 10-7, CAN identifier, filter 2
//      !!!!!.---------- Byte ACR[2], Bits 6-3, CAN identifier, filter 2
//      !!!!!!.--------- Byte ACR[3], Bits 2-0, CAN identifier, RTR bit, filter 2
//      !!!!!!!.-------- Byte ACR[3], Bits 3-0, data byte 1, filter 1
//      !!!!!!!!
//      vvvvvvvv
ACR = 0x003F007F;

//      .--------------- Byte AMR[0], Bits 10-7, CAN identifier, filter 1
//      !.-------------- Byte AMR[0], Bits 6-3, CAN identifier, filter 1
//      !!.------------- Byte AMR[1], Bits 2-0, CAN identifier, RTR bit, filter 1
//      !!!.------------ Byte AMR[1], Bits 7-4, data byte 1, filter 1
//      !!!!.----------- Byte AMR[2], Bits 10-7, CAN identifier, filter 2
//      !!!!!.---------- Byte AMR[2], Bits 6-3, CAN identifier, filter 2
//      !!!!!!.--------- Byte AMR[3], Bits 2-0, CAN identifier, RTR bit, filter 2
//      !!!!!!!.-------- Byte AMR[3], Bits 3-0, data byte 1, filter 1
//      !!!!!!!!
//      vvvvvvvv
AMR = 0x20001F00;
```

For the acceptance code register of filter 1 we will use the same settings like already known from examples before (bit 9 set to 1, all other bits set to 0). Due to the fact, that the bit positions are different the acceptance code value also looks different. But for filter 1 the result is identical. For the second filter we have to find a bit mask which is common to all identifiers we want to accept (248 - 251). This is 0x1F0 if we include the RTR bit.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Result:

We only should receive messages in the first group, which have only identifier bit 9 set while bit 1 is ignored. This means that messages with identifier 256 and 257 will pass the filter. Furthermore message will be accepted in the second filter which have bits 8, 7, 6, 5 and 4 set while bit 3 needs to be unset and bits 2 and 1 are ignored. This means, that messages 248 up to 251 (inclusive) will also be able to pass the filter. That's the result we want to reach!

4.5.2.4 Hardware filtering for extended CAN frames

Because the standard host interface is not able to work with extended identifiers, our example application needs to be expanded so that the fast host interface will be initialized in parallel to the standard host interface. Regularity we do describe this in the next chapter. Therefore please accept the additional code lines like they are. Don't worry; you'll understand them better when reading the next chapter.

For a better understanding let us shortly explain how the application has been changed. We initialized the fast host interface which uses an own message handler call-back function. Therefore received messages will end up in the fast host message handler in this example! The configuration is absolutely identical, it only uses other values for the acceptance mode and acceptance code registers.

4.5.2.5 Filtering extended frames in single filter mode

There is no difference between filter settings for receiving standard (11-bit) CAN frames and extended (29-bit) CAN frame! The hardware filter settings will be used for both identifier types. Only the bit meanings are different.

Because of the length for extended identifiers nearly all bits in the acceptance configuration registers are needed to define settings for the CAN identifier. No data bytes are included in the bit comparisons. Only the lowest two bits are defined as unused. The identifier starts in bit 3, bit 2 will contains settings for the RTR bit. In this chapter we will configure exactly the same filter like already done for standard CAN frames in the chapters above.

```
// setting mode
mode = 1; // single filter mode

//      .--------------- Byte ACR[0], Bits 25-28, CAN identifier
//      !.-------------- Byte ACR[0], Bits 21-24, CAN identifier
//      !!.------------- Byte ACR[1], Bits 17-20, CAN identifier
//      !!!.------------ Byte ACR[1], Bits 13-16, CAN identifier
//      !!!!.----------- Byte ACR[2], Bits 9-12, CAN identifier
//      !!!!!.---------- Byte ACR[2], Bits 5-8, CAN identifier
//      !!!!!!.--------- Byte ACR[3], Bits 1-4, CAN identifier
//      !!!!!!!.-------- Byte ACR[3], Bits 0-1 unsued, RTR
//      !!!!!!!!
//      vvvvvvvv
ACR = 0x00000800;

//      .--------------- Byte AMR[0], Bits 25-28, CAN identifier
//      !.-------------- Byte AMR[0], Bits 21-24, CAN identifier
//      !!.------------- Byte AMR[1], Bits 17-20, CAN identifier
//      !!!.------------ Byte AMR[1], Bits 13-16, CAN identifier
//      !!!!.----------- Byte AMR[2], Bits 9-12, CAN identifier
//      !!!!!.---------- Byte AMR[2], Bits 5-8, CAN identifier
//      !!!!!!.--------- Byte AMR[3], Bits 1-4, CAN identifier
//      !!!!!!!.-------- Byte AMR[3], Bits 0-1 unsued, RTR
//      !!!!!!!!
//      vvvvvvvv
AMR = 0x0000000F;
```

For defining the same filter settings like already used for filtering messages with standard CAN identifiers, we only need to shift the bit some positions to the left, due to the fact, that some more bits are used for the identifier. But if you compare both acceptance code values and the acceptance mask

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

values you will see, that the same bits at the same identifier bit positions have the same values. Also for this example the unused and the RTR bits are masked off in the acceptance mask and bit 9 of the CAN identifier is defined to be 1 for acceptance in the acceptance code register.

Result:

The results are quite identical to the example which works with standard CAN identifiers. Only CAN message frames with identifier 256 and 257 (in 29-bit format) can pass through the filter. Of course, you are able to define a wider range of identifiers to be accepted as this would be possible for standard identifiers.



If you are using standard and extended CAN identifiers in parallel on your CAN bus, one and the same filter settings will have different effects for both. If we take a look to the just made settings in this chapter, these settings will also be used for standard identifiers if the CAN controller receives any. But in this case the bit meanings for standard identifiers are valid! So a completely other behaviour will be the result. In this case bits 2 and 3 of data byte 2 will be 'don't care' and bit 3 of data byte 1 has to be set in a message which will pass through the filter. The identifier always has to be 0.

You see, that it is nearly impossible to define reasonable filter settings which are valid for both, standard and extended identifiers. We recommend being very careful to use hardware filtering if there are messages with standard and extended CAN identifiers transferred over the bus.

4.5.2.6 Filtering extended frames in dual filter mode

Filtering CAN frames with extended identifiers includes much more restrictions than all examples before. Due to the fact, that 29-bit identifier has even 29-bit and that we only have 32 bits available in the acceptance mask and acceptance code register, it is only possible to define filter settings for groups of identifiers! Defining an acceptance filter for single CAN identifiers is impossible in this filter mode!

```
// setting mode
mode = 0; // dual filter mode

//      .--------------- Byte ACR[0], Bits 25-28, CAN identifier
//      !.-------------- Byte ACR[0], Bits 21-24, CAN identifier
//      !!.------------- Byte ACR[1], Bits 17-20, CAN identifier
//      !!!.------------ Byte ACR[1], Bits 13-16, CAN identifier
//      !!!!.----------- Byte ACR[2], Bits 9-12, CAN identifier
//      !!!!!.---------- Byte ACR[2], Bits 5-8, CAN identifier
//      !!!!!!.--------- Byte ACR[3], Bits 1-4, CAN identifier
//      !!!!!!!.-------- Byte ACR[3], Bits 0-1 unsued, RTR
//      !!!!!!!!
//      vvvvvvvv
ACR = 0x00000002;

//      .--------------- Byte AMR[0], Bits 25-28, CAN identifier
//      !.-------------- Byte AMR[0], Bits 21-24, CAN identifier
//      !!.------------- Byte AMR[1], Bits 17-20, CAN identifier
//      !!!.------------ Byte AMR[1], Bits 13-16, CAN identifier
//      !!!!.----------- Byte AMR[2], Bits 9-12, CAN identifier
//      !!!!!.---------- Byte AMR[2], Bits 5-8, CAN identifier
//      !!!!!!.--------- Byte AMR[3], Bits 1-4, CAN identifier
//      !!!!!!!.-------- Byte AMR[3], Bits 0-1 unsued, RTR
//      !!!!!!!!
//      vvvvvvvv
AMR = 0x00000001;
```

Both registers, the acceptance code and the acceptance mask, are divided into to ranges of 16 bit, one range for each filter. The first part of each register is used to define filter settings for the highest 16 identifier bits, which can be used from the first filter. The second part will be used from the second filter for the same purpose. This means, that bits 0 up to bit 12 of each identifier are set to 'don't care'

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

automatically and will be ignored in each bit comparison! Therefore filter can be defined for CAN message identifiers only in groups of 4096 or higher!

In our example we define two filter groups, one group of 4096 identifiers starting with identifier 0 and one group of 8192 identifiers starting with identifier 8192. For filter 1 we simply set all bytes in the acceptance mask and the acceptance code to 0. This means, that all bits above the 12th are used for filter comparison and have to be 0 for acceptance. In filter two the second bit of the acceptance code is defined to be 1 for acceptance. Bit 1 of the acceptance mask is set to 'don't care'.

Result:

CAN message frames will be accepted, if their identifiers are out of the interval [0; 4095] or [8192; 16302]. This filter mechanism is mostly used for very large CAN networks. If all members of a group of nodes use identifiers in same identifier range, the other identifiers can be filtered out with this mechanism. This way it is possible to define different sub networks which all transfer messages over the same cable, but which will not recognize messages from another groups. But to be able to use the hardware filter this way, you need to plan very well how nodes use identifiers for communication.

4.6 Cyclic sending of CAN frames

If the same messages shall be cyclically sent with the same time DELAY between two messages, you can request the ICANOS firmware to do this instead of programming a timer mechanism in your own program. For this purpose a list of CAN identifiers can be created and configured with a time value. All CAN messages which are added to this list will be sent in a block after the corresponding timer has elapsed. After that the timer starts running again. So these messages will be sent cyclically over the CAN bus until the list is destroyed.

For initializing the list and adding messages to it also firmware requests are used. MITOP library function ‘IcCreateCycSendList’ creates a firmware request for initialization of a message list. The example below creates this list with a time delay of 5000 milliseconds. That means, that all messages added to this list will be send over the bus each 5 seconds.

```
// create message for 'BusOn' message
printf ("\tSwitching bus on ... ");
IcBusOnBCAN(&msg);
// send 'BusOn' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot change bus status\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}
printf ("done\n");

// create message for 'BusOn' message
printf ("\tCreating list for cyclic sending of CAN frames ... ");
IcCreateCycSendList(&msg, 5000);
// send 'CreateList' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Cannot create list for cyclic sending\n");
        printf ("Progam aborted!\n\n");
        return (-3);
}
printf ("done\n");

// Wait until creation of send list was confirmed from the firmware
// Of course a semaphore mechanism is better and faster here, but we use
// this simple method to bring as less as possible system dependencies in
// our example program. Each customer may implement the synchronisation method
// he likes!!
printf ("\tWaiting until creation of send list was confirmed from firmware:\n");
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
sleep(1);
if (confirmed != 1) {
        printf ("\tERROR: Creation of send list not yet confirmed.\n");
        printf ("Progam aborted!\n\n");
        return (-2);
}

// initializing CAN message data
data[0] = 0xaa;
data[1] = 0xbb;
data[2] = 0xcc;
data[3] = 0xdd;
data[4] = 0xee;
data[5] = 0x0;
data[6] = 0x0;
data[7] = 0x0;
```

The problem is that the initialization needs a little bit of time and will be confirmed with a firmware response when successfully finished. Therefore we also need to add a corresponding path in our message handler. The response message will contain a handle value which is needed for adding CAN frames to the list. Several lists can be created in parallel for example with different time delays. Each list will have its own unique handle value for addressing. Without this handle value we cannot continue, therefore we have to wait until the response has been received.

Ok, we use a little bit rough method to wait for the confirmation in our main routine! We simple wait a second and check the value of a globally defined variable which will be changed when the confirmation arrives. Of course, you know 1000 methods, how to solve this much more elegant! It's your turn to do this … in your application.

```
// confirmation for creation request of a send list
case M_CYC_LIST_CREATE_con:

        // display messsages
        printf ("Cyclic sending request confirmation received\n");
        printf ("Error status = %i\n", d[0]);
        // check error status
        if (d[0] == 0)
        {
                // get list handle
                hCycList = d[1];
                printf ("List handle = %i\n", hCycList);
                // Set confirmation flag
                confirmed = 1;
        } else {
                printf ("error code = %i\n", d[1]);
                // set confirmation flag
                confirmed = 0;
        }

        break;
```

To demonstrate the cyclic sending of CAN messages we will add three CAN messages to the list each with the same data but using a different CAN identifier and a different message length. We use MITOP function 'IcAddMsgToCycSendList' which creates the needed firmware request for us.

```
// initializing CAN message data
data[0] = 0xaa;
data[1] = 0xbb;
data[2] = 0xcc;
data[3] = 0xdd;
data[4] = 0xee;
data[5] = 0x0;
data[6] = 0x0;
data[7] = 0x0;

printf ("Adding CAN frames to list ... ");
IcAddMsgToCycSendList(&msg, hCycList, 221, 3, 0, data);
if (can_send (fd, &msg) < 0)
{
        // TIMEOUT, abort execution
        printf ("failed\n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        printf ("Progam aborted!\n\n");
        return (-4);
}
printf ("1 ... ");

IcAddMsgToCycSendList(&msg, hCycList, 221, 4, 0, data);
if (can_send (fd, &msg) < 0)
{
        // TIMEOUT, abort execution
        printf ("failed\n");
        printf ("Progam aborted!\n\n");
        return (-4);
}
printf ("2 ... ");

IcAddMsgToCycSendList(&msg, hCycList, 221, 5, 0, data);
if (can_send (fd, &msg) < 0)
{
        // TIMEOUT, abort execution
        printf ("failed\n");
        printf ("Progam aborted!\n\n");
        return (-4);
}
printf ("3 ... done\n");

printf ("\n\n\tReady for receiving data ...\n\n");

// Wait a long time while data is received in message handler
// call-back function
sleep (20);
```

As response for these actions the message handler gets also a corresponding responses for each created entry in the list. Therefore we also need an additional path for this response type.

```
case M_CYC_MSG_INSERT_con:
        // display messsages
        printf ("\tConfirmation received for message which was ");
        printf ("inserted into a cyclic send list\n");
        printf ("\tError status = %i\n", d[0]);
        printf ("\tInsertion ");
        if (d[0] == 0) {
                printf ("was successful!\n");
                printf ("\tList handle value: %i\n", d[1]);
        } else {
                printf ("has failed!\n");
                printf ("\tError value value: %i\n", d[1]);
        }

        return;
```

List deletion is done with own firmware request. MITOP function ‘’ helps us creating the corresponding firmware request. The list handle value is needed, to enable the firmware to delete the correct list.

```
printf ("\n\n");
printf ("PROGRAM SHUTDOWN: \n");

// delete cyclic send list
printf ("Deleting list for cyclic sending of CAN frames ... ");
IcDeleteCycSendList (&msg, hCycList);
if (can_send (fd, &msg) < 0)
{
        // TIMEOUT, abort execution
        printf ("failed\n");
        printf ("Progam aborted!\n\n");
        return (-4);
}
printf ("done\n");

// create data for 'BusOff' message
printf ("\tSwitching bus off ... ");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



This feature is not available for ICANOS daemon!



For sample code please see directory :

code / chapter4 / icanos_cyclic / icanos_cyclic.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.7 CAN bus error handling NEW!

Without additional configuration CAN bus errors are signalled by the firmware only when the error counter reached error warning level or error passive level. That's normally enough to inform an application about a critical situation regarding bus communication. We do not forward more detailed error information to the application for default because CAN bus errors are not always critical. The bus itself is very error tolerant and is able to solve a lot of error situations without activities from the application side.

Indicating errors to the application works via firmware response messages sent to the application's message handler. Each response message uses one interrupt to be fetched from the host. Error situations can occur very frequently on the bus. Sending indications for all error situations can end in heavy interrupt load which can make a host system unable to react to anything! That's the reason why we are using a special method to transport error information to the application.

The host application can defined a value how many error indications are wanted at most. This value is called quota and must be a value between 0 and 255. Function 'IcSwitchBerrBCAN' is able to create a firmware request for setting this quota value on the device.

```
// create message for baud rate settings
printf ("\tSetting baud rate to 1000 Bit/s ... ");
// set baud rate
IcWriteBtrBCAN(&msg, BTR_1MB);
// send 'baud rate' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot set baud rate\n");
        printf ("Progam aborted!\n\n");
        return (-5);
}
printf ("done\n");

// enabling bus-error indications
printf ("\tEnabling bus error indications, quota = %i ... ", quota);
IcSwitchBerrBCAN(&msg, quota);
// send 'SwitchBerrBCAN' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot set error message quota\n");
        printf ("Progam aborted!\n\n");
        return (-7);
}
printf ("done\n");

// create message for 'BusOn' message
printf ("\tSwitching bus on ... ");
IcBusOnBCAN(&msg);
```

After this the firmware will send exactly as much error indications to the host as defined in this quota value. The quota value will be decreased for each indication sent to the host application. Furthermore each indication message will contain the actual quota value indicating how much error indications are

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

remaining. So, the host application can decide when and how much the quota should be increased. The firmware will stop sending error indications when the quota went to 0.

In corresponding example for this chapter we also provide several functions making interpreting error information a little bit easier. Function 'ShowSJA1000ModeRegisterMeanings' converts the value of the CAN controller mode register into readable text. Function 'ShowSJA1000ErrorRegisterMeanings' does the same with the SJA1000 error register value contained in an error indication message. For interpreting SJA1000 status register values function 'ShowSJA1000StatusRegisterMeanings' is provided. Feel free to change these functions to your own needs and to use them in your own applications.



For sample code please see directory :

code / chapter4 / icanos_err_quota / icanos_err_quota.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.8 Bulk buffer

The bulk buffer is an effective way for receiving CAN frames with a minimum of interrupt load for the host system. A storage area is defined for it in the firmware memory where CAN frames are put until the buffer is full or a timer has elapsed. This reduces the interrupt load for the host system, because an interrupt is not raised for each incoming CAN frame but only for a bulk of them. This makes this mechanism to an ideal method to receive CAN frames when having high data traffic on the bus.

To be able to use the bulk buffer, the buffer needs to be initialized in the firmware. Function 'InitBulkBuffer' or 'InitBulkBufferTimeout' can be used for this. Whereby function 'InitBulkBufferTimeout' is the easier way, because it uses default settings for the storage management parameters of function 'InitBulkBuffer'. Please see the ICAN3 Software Reference Manual to get more information about parameters of function 'InitBulkBuffer'. The timeout value parameter used in function 'InitBulkBufferTimeout' needs to be given in 10ms steps, so that 100 used in our example below defines a timeout value of 1 second. A value of 0 will disable the bulk buffer.

```
// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// Create bulk buffer initialization message using
// a timeout of 100 * 10ms
printf ("\tInitializing bulk buffer ... ");
InitBulkBufferTimeout (&msg, 100);
// send initialization message to the firmware
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("\tERROR : Cannot initialize bulk buffer, can_send failed\n");
        return (-1);
}
printf ("done\n");

// send initialization message to the firmware
printf ("\tConfiguring messages from ID range 0 to ");
printf ("2000 to be received in the bulk buffer ... ");
```

 © **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
IcRangeSetAfil (&msg, 0, 2047, 1);
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("\tERROR : Cannot configure bulk buffer, can_send failed\n");
        return (-1);
}
printf ("done\n");

// BusOn request

// create message for baud rate settings
printf ("\tSetting baud rate to 1000 Bit/s ... ");
// set baud rate
IcWriteBtrBCAN(&msg, BTR_1MB);
// send 'baud rate' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot set baud rate\n");
        printf ("Progam aborted!\n\n");
        return (-5);
}
printf ("done\n");
```

CAN frames are not redirected to the bulk buffer for default! We need also need to configure this. On the one hand this is additional work to do. On the other hand we have the possibility to use a bulk buffer and regular CAN frame reception in parallel, regular CAN frame reception for time critical CAN frames and bulk buffer for frames not needed once.

For configuration we use function ‘IcSetAfil’ or ‘IcSetAfilRange’ which are already known from the software filter configuration. In fact, redirecting CAN frames to another interface is quite similar to filtering CAN frames. The filter engine of the firmware decides whether a frame needs to be dropped or whether it needs to be redirected to another interface. For redirecting CAN frames to the bulk buffer we use 1 as argument for this function.

The message handler needs also to be expanded for receiving firmware responses from bulk buffer service. Such an indication will possibly contain more than one received CAN frame in its body. The table below describes the format of the CAN frame data inside a bulk buffer indication message.

i	data[i]	Description
0	Length	Length of the complete message
1	Command	Command code, should be always 2
2	Descriptor word (LSB)	Descriptor containing identifier, rtr bit and data length code (DLC)
3	Descriptor word (MSB)	Descriptor containing identifier, rtr bit and data length code (DLC)
4	Data byte 0	Data byte 0 of the original CAN frame, if the message contains data (DLC > 0)
:	:	:
4 + DLC	Data byte	Last data byte of the original CAN frame

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Table 12: Data structure of a bulk buffer indication message

The following code lines were added to the message to extract the message data out of the firmware indication message. Please note, that the end of the buffer is signalled with values 0xFF for the command and length member.

```
// message(s) present in bulk buffer
case M_BCAN_BULK_ind:
        // set a pointer to the starting point of the message buffer
        msgPtr = (void *) &msg->data[0];

        // Bulk buffer message indication, bulk buffer is full or timeout is elapsed
        printf ("BULK BUFFER: messages received\n");

        // run until all messages are maintained
        while (!end)
        {
                // get the message length
                len = *msgPtr;
                // increase pointer position
                msgPtr++;
                // get command value, should be always 2
                cmd = (unsigned char) *msgPtr;

                // print out header of each message consisting of
                // length and command information
                printf ("message header information : cmd=%i, len=%i\n", cmd, len);

                // check whether end of bulk buffer was reached
                if ((cmd == 0xFF) && (len == 0xFF))
                {
                        printf ("\t\t => end reached\n");

                        // set end flag to TRUE => stop this loop,
                        // end-of-messages reached
                        end = TRUE;
                } else {
                        // increase pointer position
                        msgPtr++;
                        // get first descriptor bytes
                        desc1 = (unsigned char) *msgPtr;
                        // increase pointer position
                        msgPtr++;
                        // get second descriptor bytes
                        desc2 = (unsigned char) *msgPtr;

                        // get data length code out of descriptor bytes
                        msglen = (desc2 & 0x0F);

                        // get id out of descriptor bytes
                        id = desc1*8 + (desc2 >> 5);

                        // get RTR bit out of descrptor bytes
                        rtr = ((desc2 & 0x10) >> 4);


                        // print out message data
                        printf ("        message: id=%i, l=%i, rtr=%i", id, len, rtr);
                        for (i=0; i<len-4; i++)
                        {
                                // increase pointer position
                                msgPtr++;
                                dataByte = *msgPtr;
                                printf (", d%i=%i", i, dataByte);
                        }
                        printf ("\n");

                        // set end flag to FALSE => continue with next message in buffer
                        end = FALSE;

                        // increase pointer position
                        msgPtr++;
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
            }
        }

        break;
```



This feature is not available for ICANOS daemon!



For sample code please see directory :

code / chapter4 / bulk_buffer/ bulk_buffer.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

4.9 SNIFF buffer

The SNIFF buffer was originally implemented for debug purposes only. It works quite similar to the bulk buffer. It enables an application to get information about when a CAN frame was received, because a time stamp is added to each frame buffer at receive time. CAN frames will be collected in a buffer and transferred to the host when either the buffer is full or a configured timeout has occurred.



This feature is not available for ICANOS daemon!

4.9.1 SNIFF buffer initialization

Like for the bulk buffer also two initialization functions are available for the SNIFF buffer, ‘InitSniffBuffer’ and ‘InitSniffBufferTimeout’. Function ‘InitSniffBufferTimeout’ sets default values for the most parameters available for function ‘InitSniffBuffer’. When starting this should be the function of your choice. You will find more detailed information about Function ‘InitSniffBuffer’ in the ICAN3 Software Reference manual. The timeout value parameter used in function ‘InitSniffBufferTimeout’ needs to be given in 10ms steps, so that 100 used in our example below defines a timeout value of 1 second. A value of 0 will disable the bulk buffer. The special value 0xffff, makes SNIFF buffer messages be delivered immediately (with a timeout of 0).

```
// call the one-in-all initialization function
printf ("\tInitializing device .... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// Initializing the SNIFF buffer mechanism
printf ("\tInitializing SNIFF buffer ... ");
// InitSniffBuffer (&msg, 100, 252, 4);
InitSniffBufferTimeout (&msg, 100);
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Function 'can_send' failed\n");
        printf ("Progam aborted!\n\n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        return (-1);
}
printf ("done\n");

// Configuring the SNIFF buffer mechanism
printf ("\tConfiguring SNIFF buffer ... ");
```

4.9.2 SNIFF buffer configuration

The next step will be configuring the SNIFF buffer. We have to specify for which kind of CAN frames the SNIFF buffer shall be used and which CAN identifier shall pass the prefixed filter.

Function 'ConfigSniffBufferEcho' has to be used for setting some configuration parameters for the SNIFF buffer. There are four macro values defined in header file 'bcan.h'. You can specify one of these values or a combination of any separated by a logical OR operator.

- **SNIFF_RX_FROM_CAN** - Puts a copy of all CAN frames received over the CAN bus into the SNIFF buffer.
- **SNIFF_ECHO_FROM_CANLOOK** - Puts a copy of all CAN frames sent from CANlook into the SNIFF buffer (obsolete, CANlook does not longer use this interface!)
- **SNIFF_ECHO_FROM_PLAINQ** - Puts a copy of all CAN frames sent over the standard host interface (former known as ‘plain queue’) into the SNIFF buffer
- **SNIFF_ECHO_FROM_FASTQ** - Puts a copy of all CAN frames sent over the 'Fast Host Interface' into the SNIFF buffer

In our example we use all macros at the same time, so that we will receive a copy of each transferred message in our SNIFF buffer.

```
// Initializing the SNIFF buffer mechanism
printf ("\tInitializing SNIFF buffer ... ");
// InitSniffBuffer (&msg, 100, 252, 4);
InitSniffBufferTimeout (&msg, 100);
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Function 'can_send' failed\n");
        printf ("Progam aborted!\n\n");
        return (-1);
}
printf ("done\n");

// Configuring the SNIFF buffer mechanism
printf ("\tConfiguring SNIFF buffer ... ");
ConfigSniffBufferEcho (&msg,  SNIFF_RX_FROM_CAN | SNIFF_ECHO_FROM_CANLOOK |
                              SNIFF_ECHO_FROM_PLAINQ | SNIFF_ECHO_FROM_FASTQ);
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Function 'can_send' failed\n");
        printf ("Progam aborted!\n\n");
        return (-1);
}
printf ("done\n");

// Configuring the SNIFF buffer mechanism
printf ("\tConfiguring SNIFF buffer filter settings for std messages ... ");
```

Furthermore we have the possibility to define a message filter separately for the SNIFF buffer. With this filter we can specify ranges of CAN identifier to be received or rejected from the SNIFF buffer engine in the firmware. Depending on whether you are working with standard (11-bit) or extended (29-bit) CAN identifier, you have to use function 'ConfigSniffBufferAfilStd' or 'ConfigSniffBufferAfilExt'. In our example we define a range from ID 10 to ID 500 of standard CAN identifier to be rejected.

```
// Configuring the SNIFF buffer mechanism
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
printf ("\tConfiguring SNIFF buffer ... ");
ConfigSniffBufferEcho (&msg,  SNIFF_RX_FROM_CAN | SNIFF_ECHO_FROM_CANLOOK |
                              SNIFF_ECHO_FROM_PLAINQ | SNIFF_ECHO_FROM_FASTQ);
if (can_send(fd, &msg) <= 0)
{
       printf ("failed\n");
       printf("ERROR : Function 'can_send' failed\n");
       printf ("Progam aborted!\n\n");
       return (-1);
}
printf ("done\n");

// Configuring the SNIFF buffer mechanism
printf ("\tConfiguring SNIFF buffer filter settings for std messages ... ");
ConfigSniffBufferAfilStd (&msg, 10, 500, 0);
if (can_send(fd, &msg) <= 0)
{
       printf ("failed\n");
       printf("ERROR : Function 'can_send' failed\n");
       printf ("Progam aborted!\n\n");
       return (-1);
}
printf ("done\n");

// we set the message filter, so that all reception over standard interface
// is disabled. So messages will be received only over the SNIFF buffer
// mechanism !!
printf ("\tSetting filter options for disabling reception of all messages ... ");
IcRangeSetAfil (&msg, 0, 2047, 0);
// send 'CloseFilter' message
if (can_send(fd, &msg) <= 0)
{
       printf ("failed\n");
       printf("ERROR : Cannot set filter mask\n");
       printf ("Progam aborted!\n\n");
       return (-3);
}
printf ("done\n");

// BusOn request

// create message for baud rate settings
printf ("\tSetting baud rate to 1000 Bit/s ... ");
// set baud rate
IcWriteBtrBCAN(&msg, BTR_1MB);
// send 'baud rate' message
if (can_send(fd, &msg) <= 0)
{
       printf ("failed\n");
       printf ("ERROR : Cannot set baud rate\n");
       printf ("Progam aborted!\n\n");
       return (-5);
}
printf ("done\n");
```



If you are working with extended CAN identifier the filter can only be set for groups of CAN identifier. Please take a look into the ICAN software reference manual for getting more details about the valid parameter values for function 'ConfigSniffBufferAfilExt'.

4.9.3 SNIFF buffer message handling

Message handling also works quite equal to the bulk buffer message handling. CAN frames also were transmitted in a bulk to the host application's message handler. Additionally to the bulk buffer each CAN frame has a type indicator and timestamp data included. How to extract message data is shown in the message handler path below.

```
// SNIFF BUFFER response, either timeout occurred or SNIFF
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
// buffer was full in firmware
case M_BCAN_SNIFF_ind:
        // Work while unhandled messages are present
        // (end mark 0xFFFF was not reached)
        while(!((d[0] == 0xff) && (d[1] == 0xff)))
        {
                // check whether end of response message buffer was reached
                if ((d - dSave) >= MSGLEN)
                {
                        // abort
                        src = unknown;
                        id = 0;
                        break;
                }

                // use the data depending on the second message
                // byte holding a message type specifier
                switch (d[1])
                {
                // StdID sent over plain queue
                case SNIFF_MS_ECHO_FROM_PLAIN_STD:
                        src = eps;
                        id = (d[4]<<5)+(d[5]>>3);
                        break;
                // ExtID sent over plain queue
                case SNIFF_MS_ECHO_FROM_PLAIN_XTD:
                        src = epx;
                        id = (d[4]<<21)+(d[5]<<13)+(d[6]<<5)+(d[7]>>3);
                        break;
                // StdID sent over 'Fast Host Interface'
                case SNIFF_MS_ECHO_FROM_FAST_STD:
                        src = efs;
                        id = (d[4]<<5)+(d[5]>>3);
                        break;
                // ExtID sent over 'Fast Host Interface'
                case SNIFF_MS_ECHO_FROM_FAST_XTD:
                        src = efx;
                        id = (d[4]<<21)+(d[5]<<13)+(d[6]<<5)+(d[7]>>3);
                        break;
                // StdID sent from CANlook
                case SNIFF_MS_ECHO_FROM_CANLOOK_STD:
                        src = ecs;
                        id = (d[4]<<5)+(d[5]>>3);
                        break;
                // ExtID sent from CANlook
                case SNIFF_MS_ECHO_FROM_CANLOOK_XTD:
                        src = ecx;
                        id = (d[4]<<21)+(d[5]<<13)+(d[6]<<5)+(d[7]>>3);
                        break;
                // StdID CAN message received
                case SNIFF_MS_CAN_RX_STD:
                        src = rxs;
                        id = (d[4]<<5)+(d[5]>>3);
                        break;
                // ExtID CAN message received
                case SNIFF_MS_CAN_RX_XTD:
                        src = rxx;
                        id = (d[4]<<21)+(d[5]<<13)+(d[6]<<5)+(d[7]>>3);
                        break;
                // error situation, messages lost
                case SNIFF_MS_MSG_LOST:
                        lostMsg = d[2] + (d[3] << 8);
                        break;
                // error situation, unknown specifier byte value
                case SNIFF_MS_UNKNOWN:
                default:
                        src = unknown;
                        id = 0;
                        break;
                }

                // if no messages were lost
                if (lostMsg == 0)
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                {
                        // extract message data
                        rtr = (d[2] >> 6) & 1;
                        len = d[2] & 0xf;
                        dezi = d[16] + (d[17]<<8) + (d[18]<<16) + (d[19]<<24);
                        hr = d[20] + (d[21]<<8);

                        // create timestamp
                        s  = dezi / 1000000;
                        min = s / 60;
                        s = s % 60;
                        ms = (dezi % 1000000) / 1000;
                        us = (dezi % 1000);

                        // print text into buffer
                        n = sprintf(buff,"%s %.5d:%.2d:%.2d.%.3d.%.3d  %.9d %d",
                                src, hr, min, s, ms, us, id, rtr);
                        if ( rtr != 1)
                        {
                                x = sprintf(&buff[n], " %d", len);
                                n += x;
                                for (i=0; i<len; i++)
                                {
                                        x = sprintf(&buff[n], " %2.2x", (int)d[i+8]);
                                        n += x;
                                }
                        }
                } else {
                        // print out an error message
                        n = sprintf(buff,"%s <%d messages lost on CAN-module>",
                                service, lostMsg);
                        lostMsg = 0;
                }

                // set null terminator
                buff[n] = 0;

                // print the message
                printf ("\tSNIFF: %s\n", buff);

                // set pointer to next message in buffer
                d += d[0];
        }
        break;
```



For sample code please see directory :

code / chapter4 / sniff_buffer/ sniff_buffer.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

5 Fast Host Interface



Fast Host Interface

The ‘Fast Host Interface’ is a software interface which provides a CAN frame data transfer mechanism on a faster and convenient way than the Standard Host Interface. The Fast Host Interface completely works on a CAN frame basis. That means that we do not transmit firmware requests to the firmware and do not receive firmware responses from the firmware in the corresponding message handler with this interface! That’s the big difference to the Standard Host Interface where CAN frames and administrative command data are included into firmware requests and responses/indications. And that’s the reason why we need to install an own message handler call-back function for the Fast Host Interface. So, administrative information like confirmations, errors and warnings will still be forwarded to Standard Host Interface call-back function. Therefore the standard host interface is needed in parallel when working with the Fast Host Interface.

The 'Fast Host Interface' is provided by both firmware types, CANopen and ICANOS. It provides some additional capabilities which the Standard Host Interface does not have:

- More than 3 prioritized queues for CAN frame sending
- Capabilities to access a second CAN controller on devices, which only has one microcontroller on board (CAN-PCI2/2, CAN-104/2).
- Receiving CAN frames with timestamps.
- Receiving CAN frames with 29-bit identifiers.
- 29-bit identifier message filtering.

This interface does not support a higher level protocol. So, only raw CAN (Layer-2) frame data can be transferred.

5.1 Initializing the Fast Host Interface

The Fast Host Interface is not activated for default but the initialization is not very complicated. The programmer only needs to enter the queue sizes which shall be used for exchanging CAN frames with the device into the device information structure. For this the member variables 'fast_if_wbuffers' and 'fast_if_rbuffers' are used. If one of these members is set to a value larger than 0, the fast host interface will be automatically initialized in function ‘can_device_setup’.

Member 'fast_if_wbuffers' contains the buffer size for CAN frames transferred from the host to the firmware. 'fast_if_rbuffers' is needed for CAN frames transmission from the firmware to our application. If only member 'fast_if_wbuffers' is set, ‘can_device_setup’ initializes the Fast Host Interface without a message handler call-back function.

If member 'fast_if_rbuffers' is set to a value greater than 0, function ‘can_device_setup’ will also try to install a message handler call-back function. Because of the slightly different functioning of the Fast Host Interface an own message handler call-back function is used for it. The pointer to this function needs to be given in member ‘fast_if_handler’ of the device information structure. ‘can_device_setup’ will return with an error, if read buffers were defined and member ‘fast_if_handler’ is set to NULL, because defining a buffer for receiving CAN frames makes less sense when there is no possibility to notify the application about incoming CAN data.

```
// print arguments
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        printf ("PROGRAM ARGUMENTS: \n");
        printf ("\tDevice access string     : %s\n", argv[1]);
        printf ("\n\n");
        printf ("PROGRAM ACTIONS: \n");

        printf ("\tInitializing device information structure ... ");
        // initialize device data structure with default values
        can_device_init_struct (&devInfo);
        // fill in needed data into the device data structure
        devInfo.devname = argv[1];
        // set version information for requested interface
        devInfo.dpm_dev_t_rev = DPM_DEV_T_REV;
        // let the library check, that the correct firmware is installed
        devInfo.fw_check_string = "ICANOS";

        // setting pointer to callback function
        devInfo.std_if_handler = messageHandler;
        devInfo.std_if_context = NULL;

        // setting pointer to callback function
        devInfo.fast_if_handler = fastMessageHandler;
        devInfo.fast_if_context = NULL;

        // initializing the fast host interface
        devInfo.fast_if_wbuffers = 10;
        devInfo.fast_if_rbuffers = 20;

        printf ("done\n");

        // call the one-in-all initialization function
        printf ("\tInitializing device ... ");
        fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
        if (fd < 0)
        {
```

Like already described for the standard host interface, a message handler function is a call-back function asynchronously called from within the DPM library. In our first example we use a really short and simple message handler function.

```
////////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
////////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    fastMessageHandler
//
// PURPOSE:
//    function is called asynchronously from the library if messages need
//    to be handled for the fast host interface.
//
// PARAMETERS:
//    <context> - Pointer given to the library in the device information
//                structure during the initialization (can_device_setup)
//    <msg>     - Pointer to a FastMessage data structure containing the data
//                of a CAN message
//
// RETURN VALUE:
//    none

void fastMessageHandler (void *context, FastMessage *msg)
{
       printf ("\tMessage received on fast host interface\n");
       return;

}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter5 / fastif_init / fastif_init.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

5.2 Sending CAN frames

After the initialization has been done, sending a CAN frame is possible directly after the BusOn request (please see corresponding chapter above). We added a function ‘icanos_fastif_send’ to our example code which has nearly the same parameter as function ‘icanos_send’ used for explaining sending CAN frames via standard host interface. We do not need to create a firmware request, we only need to put CAN frame data into a data buffer and send it to the firmware with help of function ‘can_fast_send’.

```
////////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
////////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    icanos_fastif_send
//
// PURPOSE:
//    function for sending CAN message data via the fast host interface
//
// PARAMETERS:
//    <fd>          - device connection specifier
//    <lId>         - CAN message id
//    <lLen>        - CAN mesasge length
//    <lRtr>        - RTR flag, 0 or 1
//    <lExt>        - extended CAN identifier flag, 0 or 1
//    <lData>       - Pointer to memory where CAN data byte are stored
//    <lSingleShot> - Single shot flag, 0 or 1
//    <lEcho>       - echo flag, 0 or 1
//
// RETURN VALUE:
//    1 in case of success otherwise 0

long icanos_fastif_send (     int fd, long lId, char lLen,
                              char lRtr, char lExt, char*lData, char lSingleShot, char lEcho)
{
        // descriptors
        WORD_t lDesc0, lDesc1;
        // result code
        long rc = 1;
        // Data buffer
        FastMessage fMsg;

        // create new descriptor 0 with extended identifier
        // bit, RTR bit and message length
        lDesc0 = (lExt << 7) | (lRtr << 6) | lLen;
        // create descriptor 1 with echo and single shot bit
        lDesc1 = (lEcho << 4) | (lSingleShot << 1);

        // prepare FastMessage data structure for sending data in format 1
        // suppotring standard CAN identifier as well as extended identfier
        fMsg.cmd = 1;
        fMsg.data[0] = (BYTE_t) lDesc0;
        fMsg.data[1] = (BYTE_t) lDesc1;
        fMsg.data[2] = (BYTE_t) (lId/8);
        fMsg.data[3] = (BYTE_t) ((lId << 5) & 0xff);
        fMsg.data[4] = UNUSED;
        fMsg.data[5] = UNUSED;
        fMsg.data[6] = lData[0];
        fMsg.data[7] = lData[1];
        fMsg.data[8] = lData[2];
        fMsg.data[9] = lData[3];
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        fMsg.data[10] = lData[4];
        fMsg.data[11] = lData[5];
        fMsg.data[12] = lData[6];
        fMsg.data[13] = lData[7];

        // send message on fast queue
        if (can_fast_send (fd, &fMsg) <= 0)
        {
                // set error result
                rc = 0;
        }

        // return result
        return rc;
}
```

Parameter ‘fd’ gets the device connection descriptor returned by function ‘can_device_setup’. ‘lId’ specifies the CAN message identifier. ‘lLen’ should contain the CAN message length. Parameter ‘lRtr’ can contain either a 1 or a 0. Using 1 causes the device to send the message as Remote Transmission Request. Using 0 will result in sending a normal CAN message. Flag ‘lExt’ specifies whether to use standard identifier format (lExt=0) or extended identifier format (lExt=1). ‘lData’ points to the CAN message data buffer which is 8 byte long at most.

‘lSingleShot’ and ‘lEcho’ are flags enabling some additional features of the fast host interface. If ‘lSingleShot’ is set to 1, the firmware will try to send the CAN frame only once. No retry is made if this one attempt failed. In normal case the firmware would try to resend the frame up to three times when message fails because of arbitration lost for instance. When single shot mode is used, the firmware will return an error when the first attempt failed. If ‘lEcho’ is set to 1, the firmware will forward a copy of the CAN frame data to the application’s message handler when sending was successful.



For sample code please see directory :

code / chapter5 / fastif_send / fastif_send.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

5.3 Receiving CAN frames

Receiving CAN frames works quite similar to the standard host interface. Because standard host interface and fast host interface works in parallel, the firmware needs to know which CAN frame should be forwarded to the fast host interface and which is expected on the standard host interface. So, we need to give this information to the firmware in a further step.

The firmware uses its filter engine for that. We can use MITOP functions ‘IcSetAfil’ and ‘IcRangeSetAfil’ for configuration which are already known from standard host filter configuration. Using parameter 2 as filter parameter signals the firmware, that the corresponding CAN frames need to be redirected to the fast host interface.

```
        // create message for 'BusOn' message
        printf ("\tSwitching bus on ... ");
        IcBusOnBCAN(&msg);
        // send 'BusOn' message
        if (can_send(fd, &msg) <= 0)
        {
                printf ("failed\n");
                printf ("ERROR : Cannot change bus status\n");
                printf ("Progam aborted!\n\n");
                return (-6);
        }
        printf ("done\n");

        // Wait a short time to guarantee that BusOn has been completed
        sleep(1);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        // let fast messages with ID 221 pass...
        printf ("\tConfiguring ID 221 to be received on fast host interface ... ");
        IcSetAfil(&msg, 221, 2);
        if (can_send(fd, &msg) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot set message filter\n");
                printf ("Progam aborted!\n\n");
                return (-5);
        }
        printf ("done\n");

        // Wait a little bit to be able to recognize messages
        // received in the standard message handler
        printf ("\n\tWaiting for messages on 'Fast Host Interface':\n\n");
        sleep(120);
```



CAN frames with extended identifier (29-bit) are always received via the Fast Host Interface. They are all enabled for default. If no message handler call-back function is available, CAN frames with extended identifier will be ignored.

Because only CAN frame data will be forwarded to the fast host message handler, its structure is quite simple. The ‘FastMessage’ data structure exists of two versions using the same data structure but having slightly different bit meanings! A fast host message handler needs to be able to work with both formats, because standard identifier CAN frames will be transmitted in another format (cmd=0) than the extended identifier CAN frames (cmd=1). This has historical reasons and cannot be changed to avoid compatibility problems with older customer applications! Please see the message handler code to get the information how CAN frame data can be extracted o9ut of the ‘FastMessage’ data structure.

```
//////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
//////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    fastMessageHandler
//
// PURPOSE:
//    function is called asynchronously from the library if messages need
//    to be handled for the 'Fast Host Interface'.
//
// PARAMETERS:
//    <context> - Pointer given to the library in the device information
//                structure during the initialization (can_device_setup)
//    <msg>     - Pointer to a FastMessage data structure containing the data
//                the firmware response
//
// RETURN VALUE:
//    none

void fastMessageHandler (void *context, FastMessage *msg)
{
        // CAN identifier
        long lId;
        // message length
        BYTE_t lLen;
        // RTR bit
        BYTE_t lRtr;
        // extended message identifier flag
        BYTE_t lExt;
        // echo flag
        BYTE_t *data;

        printf ("\tMessage received on 'Fast Host Interface', cmd=%i\n", msg->cmd);

        // extracting information from first data bytes depending on the
        // transferred command code
        switch (msg->cmd)
        {
        case 0:
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                lId = (msg->data[0] << 3) + ((msg->data[1] & 0xE0) >> 5);
                lRtr = ((msg->data[1] & 0x10) >> 4);
                lLen = (msg->data[1] & 0x0F);

                // set pointer to CAN data area
                data = &msg->data[2];

                // issue text messages
                printf ("\tID = %i, RTR=%i, len=%i\n", lId, lRtr, lLen);
                printf ("\tData: 0x%X, 0x%X, 0x%X, 0x%X, 0x%X, 0x%X, 0x%X, 0x%X\n",
                        data[0], data[1], data[2], data[3], data[4], data[5], data[6], data[7]);

                break;

        case 1:
                lExt = ((msg->data[0] & 0x80) >> 7);
                lRtr = ((msg->data[0] & 0x40) >> 6);
                lLen = (msg->data[0] & 0x0F);

                // check whether extended identifier were used
                if (lExt == 1)
                {
                        lId =   (msg->data[2] << 20);
                        lId +=  (msg->data[3] << 12);
                        lId +=  (msg->data[4] <<  4);
                        lId +=  ((msg->data[5] & 0xF0) >>  4);
                } else {
                        lId =   (msg->data[2] << 3);
                        lId +=  ((msg->data[3] & 0x70) >>  5);
                }

                // set pointer to CAN data area
                data = &msg->data[6];

                // issue text messages
                printf ("\tID = %i, RTR=%i, len=%i, ext=%i\n", lId, lRtr, lLen, lExt);
                printf ("\tData: 0x%X, 0x%X, 0x%X, 0x%X, 0x%X, 0x%X, 0x%X, 0x%X\n",
                        data[0], data[1], data[2], data[3], data[4], data[5], data[6], data[7]);

                break;
        default:
                printf ("\tUnknown message command code!! Message ignored\n\n");
                break;
        }
}
```



For sample code please see directory :

code / chapter5 / fastif_recv / fastif_recv.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

5.4 Special features for using a second CAN channel UPDATED

Janz provides several CAN adapter devices which can optionally be equipped with a second CAN controller / second CAN channel but which have only one microcontroller to handle both. In this case both CAN controllers will be accessed via only one device connection from the application. The table below lists the corresponding CAN adapter devices for which an optional second CAN channel is available from the hardware side.

If a second CAN channel is available it can be used in two ways. On the one hand the second CAN channel can be fully accessed over the fast host interface. All capabilities available for the fast host interface can be used on both channels independently. On the other hand the second CAN channel can be used to have a higher reliability. For example, a communication could be designed in a redundant manner. In such case the Janz software provides an easy way to change from one channel to the other when communication errors are recognized.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Picture	Product name	Description
	CAN-104	ISA carrier board in PC-104 format with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with a second CAN channel.
	CAN-PCI2	PCI carrier with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with a second CAN channel.
	CAN-PMC2	PCI carrier board in PMC format with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with a second CAN channel.
	CAN-PCIH	PCI carrier board in PMC format with a VMOD-ICAN3 module integrated on MODULbus slot 0. Optionally it can be delivered with four CAN channels. DMA master transfer can be used if the device driver is prepared for that.
	CANcard	PCMCIA carrier board with an VMOD-ICAN3 MODULbus module integrated on MODULbus slot 0

Table 13: CAN adapter devices for which an optional second CAN channel is available

5.4.1 Handling the second CAN channel with the 'Fast Host Interface'

Before the device is able to work with the second CAN channel, we have to inform the firmware, that there is a second CAN controller available and that we want to use it. This is done with help of a firmware request which can be created with help of MITOP function 'SwitchCanControllerInterpretation'. Use 1 as parameter to activate CAN controller interpretation. Parameter 0 will deactivate this feature.

```
// call the one-in-all initialization function
printf ("\tInitializing device ... ");
fd = can_device_setup (&devInfo, errBuff, ERROR_MSG_BUFFER_SIZE);
if (fd < 0)
{
        // error, abort program execution
        printf ("failed\n");
        printf ("\tERROR: %s\n", errBuff);
        printf ("\n\tProgram aborted\n\n");
        return (-1);
} else {
        printf ("done\n");
}

// Enable second CAN controller facilities
printf ("\tEnable 2nd CAN controller usage ... ");
SwitchCanControllerInterpretation(&msg, 1);
// send 'baud rate' message
if (can_send(fd, &msg) <= 0)
{
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        printf ("failed\n");
        printf ("ERROR : Unable to enable 2nd CAN controller\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}
printf ("done\n");

// Now we have two CAN channels available and we also need
// to configure two CAN channels !!!
for (channel=0; channel<CHANNEL_NO; channel++)
{
```

After that the CAN controller number for which the message is for needs to be coded in each firmware request. We provide an additional function ‘patchCCNo’ which will help us to do this. This function needs to called after the firmware request was built with help of a MITOP library function. It will write the CAN channel number to the right location in the firmware request buffer.

Because we have two CAN controllers available now, we also need to initialize two CAN networks. But be careful, we only have one device connection! In our example we do this by using a 'for'-loop.

```
// Enable second CAN controller facilities
printf ("\tEnable 2nd CAN controller usage ... ");
SwitchCanControllerInterpretation(&msg, 1);
// send 'baud rate' message
if (can_send(fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Unable to enable 2nd CAN controller\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}
printf ("done\n");

// Now we have two CAN channels available and we also need
// to configure two CAN channels !!!
for (channel=0; channel<CHANNEL_NO; channel++)
{
        // create message for baud rate settings
        printf ("\tSetting baud rate to %i Bit/s for channel %i ... ",
                baudrate[channel], channel);
        IcWriteBtrBCAN(&msg, baudrate[channel]);
        // set CAN controller number in 'Message' data structure
        patchCcNo (&msg, channel);
        // send 'baud rate' message
        if (can_send(fd, &msg) <= 0)
        {
                printf ("failed\n");
                printf ("ERROR : Cannot set baud rate\n");
                printf ("Progam aborted!\n\n");
                return (-5);
        }
        printf ("done\n");

        // create message for 'BusOn' message
        printf ("\tSwitching bus on ... ");
        IcBusOnBCAN(&msg);
        // set CAN controller number in 'Message' data structure
        patchCcNo (&msg, channel);
        // send 'BusOn' message
        if (can_send(fd, &msg) <= 0)
        {
                printf ("failed\n");
                printf ("ERROR : Cannot change bus status\n");
                printf ("Progam aborted!\n\n");
                return (-6);
        }
        printf ("done\n");

        // create message for redirecting messages to the FAST HOST interface
        printf ("\tRedirect all messages to FAST HOST interface for channel %i ... ",
                channel);
        IcRangeSetAfil(&msg, 0, 2047, 2);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                // set CAN controller number in 'Message' data structure
                patchCcNo (&msg, channel);
                // send 'BusOn' message
                if (can_send(fd, &msg) <= 0)
                {
                        printf ("failed\n");
                        printf ("ERROR : Cannot change bus status\n");
                        printf ("Progam aborted!\n\n");
                        return (-6);
                }
                printf ("done\n");
        }

        // Wait a little bit to be able to recognize messages
        // received in the standard message handler
        printf ("\n\tWaiting for messages on 'Fast Host Interface':\n\n");
        sleep(20);
```

Sending CAN frames on one of the available channels also requires setting a CAN channel number now! We changed our function ‘icanos_fast_send’ from earlier examples in this document slightly to show how the CAN controller number is coded in the ‘FastMessage’ data structure.

```
////////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
////////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    icanos_fastif_send
//
// PURPOSE:
//    function for sending CAN message data via the fast host interface
//
// PARAMETERS:
//    <fd>          - device connection specifier
//    <lChannel>    - CAN channel number, either 0 or 1
//    <lId>         - CAN message id
//    <lLen>        - CAN mesasge length
//    <lRtr>        - RTR flag, 0 or 1
//    <lExt>        - extended CAN identifier flag, 0 or 1
//    <lData>       - Pointer to memory where CAN data byte are stored
//    <lSingleShot> - Single shot flag, 0 or 1
//    <lEcho>       - echo flag, 0 or 1
//
// RETURN VALUE:
//    1 in case of success otherwise 0

long icanos_fastif_send (     int fd,
                              char lChannel,
                              long lId,
                              char lLen,
                              char lRtr,
                              char*lData,
                              char lExt,
                              char lSingleShot,
                              char lEcho)
{
        // descriptors
        WORD_t lDesc0, lDesc1;
        // result code
        long rc = 1;
        // Data buffer
        FastMessage fMsg;

        // create new descriptor 0 with extended identifier
        // bit, RTR bit and message length
        lDesc0 = (lExt << 7) | (lRtr << 6) | lLen;
        // create descriptor 1 with echo and single shot bit
        lDesc1 = (lEcho << 4) | (lSingleShot << 1);

        // prepare FastMessage data structure for sending data in format 1
        // suppotring standard CAN identifier as well as extended identfier
        fMsg.cmd = 1;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        // set CAN channel number
        fMsg.cmd |= ((lChannel&0x7)<<4);

        fMsg.data[0] = (BYTE_t) lDesc0;
        fMsg.data[1] = (BYTE_t) lDesc1;
        fMsg.data[2] = (BYTE_t) (lId/8);
        fMsg.data[3] = (BYTE_t) ((lId << 5) & 0xff);
        fMsg.data[4] = UNUSED;
        fMsg.data[5] = UNUSED;
        fMsg.data[6] = lData[0];
        fMsg.data[7] = lData[1];
        fMsg.data[8] = lData[2];
        fMsg.data[9] = lData[3];
        fMsg.data[10] = lData[4];
        fMsg.data[11] = lData[5];
        fMsg.data[12] = lData[6];
        fMsg.data[13] = lData[7];

        // send message on fast queue
        if (can_fast_send (fd, &fMsg) <= 0)
        {
                // set error result
                rc = 0;
        }

        // return result
        return rc;
}
```

Taking a look to the code snippet above it's easy to find out, that the CAN controller number is set in the upper nibble of the cmd member in data structure 'FastMessage' when sending over the fast host interface.

```
        // set CAN controller channel number (0 or 1)
        channel = 0;

        // set data bytes
        lData[0] = 0xaa;
        lData[1] = 0xbb;
        lData[2] = 0xcc;
        lData[3] = 0x0;
        lData[4] = 0x0;
        lData[5] = 0x0;
        lData[6] = 0x0;
        lData[7] = 0x0;

        // implement a while loop for sending a message a couple of times
        while (n < 20)
        {
                // send messages on both CAN controllers
                for (channel=0; channel<CHANNEL_NO; channel++)
                {
                        // send message on fast queue
                        printf ("\tSending message on fast host interface with channel=%i ... ",
                                channel);

                        if (icanos_fastif_send (fd, channel, 221, 3, 0, lData, 0, 0, 0) < 0)
                        {
                                // error
                                printf ("failed\n");
                        } else {
                                // success
                                printf ("done\n");
                        }
                }


                // wait for message handler outputs ...
                sleep (1);

                // increase counter
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                n++;
        }
```

Of course, we also have to take care in our message handler call-back function after enabling the second CAN controller. CAN frames from both CAN channels will end up in the same call-back function call. But the CAN channel number is coded in the first data byte.

In our example we wrote an own message handler function for each CAN channel and used a pre-handler function for getting the CAN channel number and forwarding the CAN frame to the channel specific handler function.

```
//////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
//////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    fastMessageHandler
//
// PURPOSE:
//    function is called asynchronously from the library if messages need
//    to be handled for the 'Fast Host Interface'.
//
// PARAMETERS:
//    <context> - Pointer given to the library in the device information
//                structure during the initialization (can_device_setup)
//    <msg>     - Pointer to a FastMessage data structure containing the data
//                the firmware response
//
// RETURN VALUE:
//    none

void fastMessageHandler (void *context, FastMessage *msg)
{
        // CAN controller number
        int ctrlNum;

        printf ("\tMessage received on 'Fast Host Interface', cmd=%i\n", msg->cmd);

        // Extract controller number
        ctrlNum = (msg->data[1]&0x70) >> 4;

        // Call corresponding message handler depending on
        // CAN controller number
        switch (ctrlNum)
        {
        case 0:
                // call message handler function for CAN controller 0
                fastMessageHandler_CC0 (context, msg);
                break;
        case 1:
                // call message handler function for CAN controller 1
                fastMessageHandler_CC1 (context, msg);
                break;
        default:
                printf ("\tERROR: Wrong CAN controller number (%i), message ignored\n",
ctrlNum);
        }

        // return into the deepness of CAN libraries ...
        return;
}
```

The message content is nearly identical with the content received when only one CAN controller is used. So please refer to the chapters above this to get information about how to work with the separate indications received in the message handler call-back routine.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter5 / snd_canctrl_fhif / snd_canctrl_fhif.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

5.4.2 Swapping between both CAN controllers

If you need the second CAN channel for safety purposes, you can use an additional feature of the Janz CAN libraries. It is possible to swap the functionality from the first CAN channel to the second and vice versa. For this feature no additional configuration work is needed. You simply need to send a corresponding request to the firmware.

We provide you a CANopen example as well as an ICANOS example in which the CAN channel is automatically changed when detecting that the error limit reaches error warning limit. This example contains function ‘swap_cc’ which is called from within the message handler when reaching the error limit was indicated.

```
////////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
////////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    swap_cc
//
// PURPOSE:
//    function is called from within the message handler for executing
//    a CAN controller swap.
//
// PARAMETERS:
//    none
//
// RETURN VALUE:
//    The function returns 0 if swapping was successful. otherwise
//    a negative value is returned

int swap_cc(int fd)
{
       // message buffer
       Message msg;

       // invert current CAN controller number
       printf ("\n\t\tPrepare application for swapping CAN controller from %i to %i ... ",
                     currentCc, !currentCc);
       currentCc = !currentCc;
       printf ("done\n");

       // calling function for building firmware request
       printf ("\t\tSwapping CAN controller ... ");
       SwapCanController (&msg, currentCc);

       // send firmware request
       if (can_send (fd, &msg) < 0)
       {
              // print error message
              printf ("failed, can_send failed\n");
              // return error
              return -1;
       }
       printf ("done\n");

       // always return success
       return 0;
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

This example shows that CAN channel swapping can be done with ICANOS firmware as well as with CANopen firmware installed on the device. It also works independent from the used interface in your application.



For sample code please see directory :

code / chapter5 / snd_canctrl_swap / snd_canctrl_swap.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



For sample code please see directory :

code / chapter6 / snd_canctrl_swap / snd_canctrl_swap.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

6 CANopen

In this chapter we would like to explain how Janz CAN adapter can be used as a CANopen node. As already explained at the beginning of this tutorial the reader should know the CANopen specifications.

6.1 Device configuration file (DCF)

Nearly all configuration settings which are needed for working with the Janz CAN adapter as a CANopen node are stored in the device configuration file (DCF). This is a text file which is parsed by a function of the CANopen host library and after that the configuration options are given to the CANopen firmware.

The file is structured like an INI file. The configuration parameters can be stored in sections which names are enclosed in brackets ([1004]). Below them the property values like data type, access rights, parameter name, etc. regarding the actual section can be given in text format. Also a default value can be set which is the parameter value itself if the parameter is a read only entry. A parameter is set through a key value followed by an equal sign and the parameter value. This value is set only for the section defined before.

Except some settings regarding the DCF itself mainly the contents of the object dictionary for the Janz CAN module is defined in the DCF. Indices and Sub-Indices are defined as section names in hexadecimal format. Following you will find an example how the index 0x1004 and its specified sub-indices are defined in a DCF. In the CANopen specification is defined, that index 0x1004 specifies the number of supported Process Data Objects (PDOs). Number and meaning of each index in the object dictionary is defined in the CANopen specifications.

```
[1004]
SubNumber=3
ParameterName=number of PDOs supported

[1004sub0]
ParameterName=number of PDOs supported
ObjectType=0x0008
DataType=0x07
AccessType=ro
ParameterValue=0x00020002
DefaultValue=0x00020002
PDOMapping=0

[1004sub1]
ParameterName=number of synchronous PDOs
ObjectType=0x0008
DataType=0x07
AccessType=ro
ParameterValue=0x00010001
DefaultValue=0x00010001
PDOMapping=0

[1004sub2]
ParameterName=number of asynchronous PDOs
ObjectType=0x0008
DataType=0x07
AccessType=ro
ParameterValue=0x00010001
DefaultValue=0x00010001
PDOMapping=0
```

This tutorial works with an sample DCF 'default.dcf' which is used for all following CANopen sample programs. Except basic definitions regarding bus communication it contains the communication and

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

mapping parameter of two process data objects which are needed for writing data to and reading data from the CAN bus in these examples.

6.2 Communication objects

For transferring data between CAN nodes so called communication objects are used. This is a logical unit in which data is packed together with a unique CAN identifier. Its data transfer direction is specified through its transmission type. Depending on the used protocol and service different communication objects are defined and used. The following table gives an overview what types of communication objects are known and for which purpose they are for.

Type of communication object	Usage
Process data object (PDO)	Transferring process data during runtime
Service data object (SDO)	Transferring configuration data during configuration phase
Synchronisation object (SYNC)	Signalling that synchronized process data shall be transferred during runtime.
Timestamp (TIME)	Transferring time information during runtime
Emergency object (EMCY)	Transferring error messages during runtime
Network management object	Transferring network management messages during runtime

Table 14: CAN adapter devices for which an optional second CAN channel is available

You will find more detailed information about communication objects in the CANopen specifications and in CANopen standard literature.

6.3 Object dictionary

The object dictionary is a dual port memory range maintained by CANopen firmware in which values are stored as entries. These entries are called indices and are organized in a hierarchical manner. Each index which is a hexadecimal value between 0x0000 to 0xFFFF can have up to 256 (0x00 to 0xFF) sub-indices. The structure of the object dictionary is also defined in the DCF. Configuration data (SDOs) as well as process data (PDOs) and other communication objects are stored in specific areas of the object dictionary and can be written and read from application side as well as from the CAN bus side.

At the beginning each entry is initialized with its default value. Default values are not written on the CAN bus. Values in the object dictionary will have this default value as long as they are changed either from application side or from the CAN bus side.

Please take a look into the standard literature to get more information about the object dictionary, about what indices are available and for what they are used for.

6.4 Initializing a slave node

In our first example we want to create a program which works as a CANopen slave node on the bus. We use the template created in chapter 3 as starting point. A name of DCF file will be given as program argument and needs to be parsed at the beginning of the main routine. Furthermore we added a function ‘canopen_init’ in which the CANopen specific initialization work is done.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
//////////////////////////////////////////////////////////////////////
// CAN TUTORIAL EXAMPLE APPLICATION
//////////////////////////////////////////////////////////////////////
//
// FUNCTION:
//    canopen_init
//
// PURPOSE:
//    initialisation function for VMOD-ICAN3 with CANOPEN firmware
//
// PARAMETERS:
//    <dcfName>       - pointer to strung buffer with DCF name
//
// RETURN VALUE:
//    int - return code
//     0  - no error
//    -1  - invalid argument pointer
//    -2  - cannot read DCF
//    -3  - can_send failed

int canopen_init (char *dcfName)
{
        // check pointer to DCF file name
        printf ("\tChecking argument pointer ... ");
        if (dcfName == NULL)
        {
                // invalid pointer
                printf ("invalid\n");
                return (-1);
        }
        printf ("valid\n");

        // read DCF to local buffer
        printf ("\tReading DCF to local buffer ... ");
        if (readDcfToLocalBufferAlt(dcfName) < 0)
        {
                // DCF could not be read
                printf ("failed\n");
                printf("ERROR : Cannot read DCF %s\n", dcfName);
                return (-2);
        }
        printf ("done\n");

        // Link libraries' driver functions for reading/writing directly
        // to/from the process picture into the object directory
        // DPM-accessing locations
        myVars.drvRead = ican_read_pp;
        myVars.drvWrite = ican_write_pp;

        // create message to configure CMS service
        printf ("\tConfiguring CMS service ... ");
        CmsConfig(&myVars.msgBufferTx, CMS_USE_HOST_ID);
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot create list for cyclic sending\n");
                printf ("Progam aborted!\n\n");
                return (-3);
        }
        printf ("done\n");

        // define timeout value in ms for CMS objects
        myVars.cmsDefinitionTimeout = 50;

        // Set endianess of the host machine
        printf ("\tSetting endianess of host machine ... ");
        SetHostEndianess(&myVars.msgBufferTx, COP_DPM_LITTLE_ENDIAN);
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot create list for cyclic sending\n");
                printf ("Progam aborted!\n\n");
                return (-3);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        }
        printf ("done\n");

        // initialize VMOD-ICAN3 module as CANopen slave
        initCanOpenAlt(&myVars);

        // return success
        return 0;
}
```

In function 'canopen_init' a DCF is opened and read into the local memory with help of function 'readDcfToLocalBufferAlt'. Functions of the CANopen host library are responsible for the memory management for storing the DCF file contents in this case so that the programmer do not pay attention for this. The function only needs the path and the name in text format where the DCF can be found. After that the function pointer of functions 'ican_read_pp' and 'ican_write_pp' have be stored in the global variable myVars. This enables functions in the CANopen library to call them directly each time it is needed.

With the function 'CmsConfig' the CMS service in the CANopen firmware is configured. Function 'SetHostEndianess' informs the CANopen firmware about the endian type of our host system. So it is possible to get each data in the expected byte order specifying the needed endian type. 'initCanOpenAlt' does the main initialisation work.

Function 'canopen_init is called from the 'main' routine after the message handler was started successfully and before we switch the device ‘BusOn’ with help of function ‘LsBusOnReq’ (please see chapter 3).

If this sample program is started something unexpected occurs after the initialisation has finished. The message handler detects some 'unhandled messages'. These messages are asynchronously received confirmations for successfully finished CANopen initialisation requests. To work with this confirmation messages in the correct way some code lines need to be added to the message handler. The CANopen library provides two functions ‘copLibHandleMsgCtrl’ and ‘copLibHandleMsgData’ for default handling. Add these two functions always at the beginning of your CANopen application message handler. They will do a lot of standard work for you! If a request was already handled in one of these functions there is no need to do this also in the message handler. Simply return in this case.

```
void messageHandler (void *context, Message *msg)
{
        // extracting the command specifier of firmware response
        int                   spec = msg->cmd;
        // setting the pointer to the first data byte in the response
        BYTE_t          *d   = &msg->data[0];
        // getting a sub-spec value from first and second message byte
        WORD_t          subspec = word_from_bytes(d[0],d[1]);

        // check if standard handler can handle the incoming message
        if (copLibHandleMsgCtrl (&myVars, msg))
        {
                // message was handled from standard handler, return without further processing
                return;
        }

        // check if standard handler can handle the incoming message
        if (copLibHandleMsgData (&myVars, msg))
        {
                // message was handled from standard handler, return without further processing
                return;
        }

        // Implementing a filter for all wanted response messages and
        // handle them each in an own path of the following switch statement.
        switch (spec)
        {
        // IDVERS response, returning firmware version information
        case M_IDVERS:
                showMessage("", msg);
                return;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

After adding these code lines to the message handler it should work without problems.

We also added a further path to our switch statement in the message handler to handle node state indication. Calling function 'initCANopenAlt' causes the CANopen firmware to set our node into status PRE-OPERATIONAL. This is signalled with a firmware indication form the NMT service. The NMT service uses the specifier M_NMTS. It uses sub-specifiers to provide more detailed information about the indication type. NMT state change indications will be sent with sub-specifier 'MS_NMTS_STATE_ind'. This indication type also contains the actual status of the device. If such an indication occurs we have to copy the status value into global variable myVars. So the status will be always accessible and up to date, because each status change will also cause such an indication message from the NMT service and the actual status value is always asynchronously copied into the global variable.

```
// Messages from the NMT-Slave
case M_NMTS:
        // check subspec value
        switch (subspec) {

        // state change indications of node forced by network NMT-master
        case MS_NMTS_STATE_ind:
                printf ("\tNode state indication, state = ");
                switch (myVars.nodeState)
                {
                case DISCONNECTED:
                        printf ("DISCONNECTED\n");
                        break;
                case CONNECTING:
                        printf ("CONNECTING\n");
                        break;
                case PREPARING:
                        printf ("PREPARING\n");
                        break;
                case PREPARED:
                        printf ("PREPARED\n");
                        break;
                case OPERATIONAL:
                        printf ("OPERATIONAL\n");
                        break;
                case PRE_OPERATIONAL:
                        printf ("PRE-OPERATIONAL\n");
                        break;
                default:
                        printf ("UNKNOWN\n");
                }
                break;


        // default processing
        default:
                showMessage ("Unhandled NMT slave message :\n", msg);
        }
        break;
```

After that the program is ready for sending and receiving data over the object dictionary as a CANopen slave node. How to do this is explained in further chapters of this document.

In addition we added some code lines to our example which let the main routine wait for the NMT state change indication after device initialization. This is helpful, because application work should not start before the device was set to the OPERATIONAL state. We are not master, that means switching the node to OPERATIONAL will be done from another node on the bus. Our application has to wait until the corresponding state change indication was received. Waiting will timeout after 30 seconds in our example.

```
printf ("\n\n\tReady for sending and receiving data !\n\n");
printf ("IMPORTANT:\n");
printf ("\tBecause we are not CANopen master, we have to wait until\n");
printf ("\tthe master switches the bus into the OPERATIONAL state!!\n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
printf ("\tSo lets wait ... \n\n");

// Waiting on OPERATIONAL state
// ============================

// Waiting on OPERATIONAL state switch is done with a while loop, which
// is getting the state cyclically. The device state is maintained from
// the CANopen library automatically!

while ((myVars.nodeState != OPERATIONAL) && (timeout < 30)) {
        // wait a further second and try again
        sleep(1);
        // increase timeout counter
        timeout++;
}

// check whether timeout occurred
if (timeout < 30) {
        printf ("\n\tState now is OPERATIONAL, so lets continue!\n");
} else {
        printf ("Application timed out!!\n");
        printf ("ERROR : No NMT state change notified before timeout\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}

// It's time to work with CANopen communication objects now (PDOs)
// In this example we only wait a short time for signaling, that
// the program is active now.
sleep(20);
```



For sample code please see directory :

code / chapter6 / conventional / init_slave / init_slave.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.5 Process data objects

Process data which shall be sent to or received from the CAN bus during runtime will be transferred with so called process data objects (PDOs). Properties and data of a PDO are defined in the DCF and stored in the object dictionary. Depending on how the property 'transmission type' is configured, data of a PDO is transferred over the CAN bus on a different manner. On the one hand this property value defines whether data is send synchronously or asynchronously. Values stored in asynchronous PDOs are sent to the CAN bus directly after the data has changed in the object dictionary. Values stored in synchronous PDOs are sent to the CAN bus only when a synchronisation message (SYNC-Message) has occurred before. Furthermore this property defines whether data sending is controlled by an event, whether data is sent cyclically or whether data sending has to be requested from another node. There are two different types of PDOs, receive PDOs (RPDO) for receiving data and transmit PDOs (TPDO) for sending data.

But data cannot be given directly to a PDO. Data entries are available in the object dictionary to which data needs to be written. These data points are connected with a PDO via the mapping parameter inside the PDO configuration.

PDOs are configured in the Device Configuration File (DCF). Depending on the type of the PDO (Receive or Transmit) the PDO configuration parameter (communication and mapping parameter) are stored in different identifier areas of the object dictionary. The following table lists the number ranges which are used for the different PDO types.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Index range	Configuration
0x1005	Synchronisation object (SYNC) parameter
0x1014, 0x1015	Emergency object (EMCY) parameter
0x1400 .. 0x15FF	RPDO communication parameter
0x1600 .. 0x17FF	RPDO mapping parameter
0x1800 .. 0x19FF	TPDO communication parameter
0x1A00 .. 0x1BFF	TPDO mapping parameter
0x1200 .. 0x127F	Server SDO parameter
0x1280 .. 0x12FF	Client SDO parameter
0x1300 .. 0x137F	Safety relevant data object (SRDO) communication parameter

Table 15: Identifier ranges for different communication objects

For example the communication parameter of the first RPDO (ReceivePDO) can be found in sub-indices of index 0x1400, the communication parameter for the second RPDO are stored in sub-indices of index 0x1401, etc. For each property value of a PDO an own sub-index value has to be used. For each PDO an own unique CAN identifier has be defined. During the communication only the identifier and the process value are transferred over the bus. So a higher effort is needed during the configuration of a CAN network but a higher performance is reached during runtime with this mechanism, because the overhead in the transferred data package is as small as it can be! More detailed information about configuration of PDOs is given in the CANopen specifications.

If the configuration of a network has finished, sending data is quite similar in each case for an application programmer. Data is only transferred to the corresponding data entry in the object dictionary. After that the corresponding PDO needs to be triggered with a further function of the CANopen host library. Depending on which transmission type was configured for this PDO the CANopen firmware chooses and executes the transfer mechanism.



More than one data entry can be mapped in one PDO! The PDO will send the complete PDO data for all mapped data entries each time it is triggered. The programmer has to be sure, that all data variables mapped into one and the same PDO were up to date before the corresponding PDO is triggered. Triggering this PDO each time when one of its data entries has changed may end in sending or requesting PDO data twice or triply.

Write function	Description
getOffsetToPp	Gets the memory offset of a process value inside the object dictionary. This value is needed as a parameter for function 'ican_read_pp' and 'ican_write_pp'.
ican_write_pp	Writes raw data to a place in the memory of the object dictionary
writeOD	Writes raw data to an entry in the object dictionary

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

writeODUnsigned8	Writes an 'unsigned integer' value with a length of 1 byte to an entry of the object dictionary
writeODUnsigned16	Writes an 'unsigned integer' value with a length of 2 bytes to an entry of the object dictionary
writeODUnsigned24	Writes an 'unsigned integer' value with a length of 3 bytes to an entry of the object dictionary
writeODUnsigned32	Writes an 'unsigned integer' value with a length of 4 bytes to an entry of the object dictionary
writeODUnsigned40	Writes an 'unsigned integer' value with a length of 5 bytes to an entry of the object dictionary
writeODUnsigned48	Writes an 'unsigned integer' value with a length of 6 bytes to an entry of the object dictionary
writeODUnsigned56	Writes an 'unsigned integer' value with a length of 7 bytes to an entry of the object dictionary
writeODUnsigned64	Writes an 'unsigned integer' value with a length of 8 bytes to an entry of the object dictionary
writeODInteger8	Write a 'signed integer' value with a length of 1 byte to an entry of the object dictionary
writeODInteger16	Write a 'signed integer' value with a length of 2 bytes to an entry of the object dictionary
writeODInteger24	Write a 'signed integer' value with a length of 3 bytes to an entry of the object dictionary
writeODInteger32	Writes a 'signed integer' value with a length of 4 bytes to entry of the object dictionary
writeODInteger40	Writes a 'signed integer' value with a length of 5 bytes to entry of the object dictionary
writeODInteger48	Writes a 'signed integer' value with a length of 6 bytes to entry of the object dictionary
writeODInteger56	Writes a 'signed integer' value with a length of 7 bytes to entry of the object dictionary
writeODInteger64	Writes a 'signed integer' value with a length of 8 bytes to an entry of the object dictionary
writeFloatingPoint	Writes a 'floating point' value to an entry of the object dictionary
writeVisString	Writes a string to an entry of the object dictionary

Table 16: Functions for writing PDO data

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Depending on the data type of a data entry different wrapper function are available in the CANopen library for writing a value. The following table lists some of these functions. Most of the listed functions will be used and described in the following chapters. Meaning and parameters of the listed functions are also explained in the ICAN3 Software Reference Manual.

For receiving data some additional work has to be done. Like explained in more than one cases above the firmware normally gives received data from the CAN bus directly to the message handler. The transmission type defines how the data is received and with which specifier the corresponding firmware indication is transferred. Asynchronous, event-based PDOs are automatically transferred to the message handler and have the specifier 'CMS_DATA_ind'.

Instead of transferring synchronous PDOs to the message handler the handler gets only the information that a synchronisation message was recognized. After that the application has to decide whether and which items have to be read from the object dictionary.

Request-controlled PDOs are not sent to CAN bus automatically. To update the data from a PDO with this transmission type the application has to send a request to the firmware. After that the firmware will send the corresponding RTR frame on the bus. If there is a node in the network which is a provider for this PDO the data of the PDO is sent from this node after the request was received. This CAN frame is forwarded to the message handler like explained for event-based PDOs.

The following table lists functions which can be used for reading data from the object dictionary. Most of these functions will be explained in more detail in the following steps of this tutorial. These and further functions are also described in the ICAN3 Software Reference Manual.

Read function	Description
getOffsetToPp	Gets the memory offset of a process value inside the object dictionary. This value is needed as a parameter for function 'ican_read_pp' and 'ican_write_pp'.
ican_read_pp	Reads raw data from a place in the memory of the object dictionary
readOD	Reads raw data from an entry of the object dictionary into a data buffer
readODUnsigned8	Reads 'unsigned integer' value with a length of 1 byte from an entry of the object dictionary into a data buffer
readODUnsigned16	Reads an 'unsigned integer' value with a length of 2 byte from an entry of the object dictionary into a data buffer
readODUnsigned24	Reads an 'unsigned integer' value with a length of 3 byte from an entry of the object dictionary into a data buffer
readODUnsigned32	Reads an unsigned integer' value with a length of 4 byte from an entry of the object dictionary into a data buffer
readODUnsigned40	Reads an unsigned integer' value with a length of 5 byte from an entry of the object dictionary into a data buffer
readODUnsigned48	Reads an unsigned integer' value with a length of 6 byte from an entry of the object dictionary into a data buffer
readODUnsigned56	Reads an unsigned integer' value with a length of 7 byte from an entry of the object dictionary into a data buffer

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

readODUnsigned64	Reads an 'unsigned integer' value with a length of 8 byte from an entry of the object dictionary into a data buffer
readODInteger8	Reads a 'signed integer' value with a length of 1 byte from an entry of the object dictionary into a data buffer
readODInteger16	Reads a 'signed integer' value with a length of 2 byte from an entry of the object dictionary into a data buffer
readODInteger24	Reads a 'signed integer' value with a length of 3 byte from an entry of the object dictionary into a data buffer
readODInteger32	Reads a 'signed integer' value with a length of 4 byte from an entry of the object dictionary into a data buffer
readODInteger40	Reads a 'signed integer' value with a length of 5 byte from an entry of the object dictionary into a data buffer
readODInteger48	Reads a 'signed integer' value with a length of 6 byte from an entry of the object dictionary into a data buffer
readODInteger56	Reads a 'signed integer' value with a length of 7 byte from an entry of the object dictionary into a data buffer
readODInteger64	Reads a 'signed integer' value with a length of 8 byte from an entry of the object dictionary into a data buffer
readFloatingPoint	Reads a floating point value from an entry of the object dictionary into a data buffer
readVisString	Reads a visible string value from an entry of the object dictionary into a data buffer

Table 17: Functions for reading PDO data

6.6 Sending data of a asynchronous PDO

In the following example we used the slave application built in the chapter before to implement sending data of an asynchronous PDO. For this a transmit PDO (TPDO) has been configured in the DCF 'default.dcf'. Two 16 bit integer values were mapped into to this TPDO. These integer values reside on index 0x6401 in sub-index 1 and 2. So writing on the value stored under 0x6401SUB1 and triggering the first TPDO should generate a corresponding CAN message on the bus having both values contained.

We use function 'writeODUnsigned8' for writing data to the object dictionary entry. This function needs a pointer to a buffer, the value which has to be written, the index and the sub-index value as function parameters. With function 'triggerTPDO' the corresponding TPDO (here TPDO No. 1) is triggered and the process value is sent in a CAN frame over the CAN bus before the program will be finished.

```
// check whether timeout occurred
if (timeout < 30) {
        printf ("\n\tState now is OPERATIONAL, so lets continue!\n");
} else {
        printf ("Application timed out!!\n");
        printf ("ERROR : No NMT state change notified before timeout\n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        printf ("Progam aborted!\n\n");
        return (-6);
}

// write a 1 byte value to index 0x2100SUB1 ten times
printf ("\nWRITING DATA\n");
for (i=0; i<10; i++)
{
        printf ("\tCycle #%i: ", i);
        // writing process value to object dictionary
        if (!writeODUnsigned16 (&myVars, 0x6401, 1, i, LOCAL_ACCESS))
        {
                printf ("failed to write value to index 0x6401SUB1\n");
        } else {
                printf ("value %i sucessfully written to index 0x6401SUB1\n", i);
        }

        printf ("\tCycle #%i: ", i);
        // writing process value to object dictionary
        if (!writeODUnsigned16 (&myVars, 0x6401, 2, i, LOCAL_ACCESS))
        {
                printf ("failed to write value to index 0x6401SUB2\n");
        } else {
                printf ("value %i sucessfully written to index 0x6401SUB2\n", i);
        }

        // trigger corresponding PDO
        triggerTPDO (&myVars, 1);
}

printf ("\n\n");
printf ("PROGRAM SHUTDOWN: \n");
```

Of course, it is a disadvantage that the bus has to be switched into the 'OPERATIONAL' status from another master node. Slave nodes are not available to do this themselves! That's the reason why they are slave nodes! Writing code for a master node is also possible and not much more complex. But among all network nodes in a CAN network there should only be one node which has master capabilities. This node works as administrator all over the network. So in many cases it will be necessary to write a slave node, because there is already another node in the network which has master capabilities.



For sample code please see directory :

code / chapter6 / conventional / pdo_send / pdo _send.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.7 Receiving data of a asynchronous, event-based PDO

Like explained in chapters before, receiving PDO data can be done in two different manners. You can either get indications in the message handler each time when new PDO data has arrived or you can simply work with the data stored in the object dictionary. You can configure the wanted manner by setting the transmission type of the PDO. Data of asynchronous PDOs will be stored automatically in the object dictionary and additionally forwarded to the application's message handler. Data of synchronous PDOs will be stored automatically in the object dictionary and can be read with an object dictionary access function each time when it is needed in the application.

Data in the object dictionary is updated automatically, so that the application can read the most actual value each time it accesses it. But the application will get no notification about when the updated value arrived! We will spend no further time with explaining, how to work with synchronous PDOs, because it is simple. You only need to use the access functions like 'readODUnsigned32' for reading the values from the object dictionary. You'll find a list of access functions in chapter above.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

You will get an indication for received PDO data when you are using an asynchronous PDO. But in this case only the raw data transferred in the CAN frame will be available in the message handler notification. But we will show you in this chapter, how to get information about the corresponding PDO and mapped values.

The sample program created in the chapter above is already able to receive PDO data from the CAN bus. The DCF also contains the line for defining a receive PDO, index 0x1400 contains the communication parameter, index 0x1600 contains the mapping parameter. We defined CAN identifier 181_{hex} (385_{dez}) for the PDO. Only a corresponding path in our message handler is missing.

```
// Message which handles received SDOs/(asynchronous/acyclic)PDOs
case M_CMS_DATA:
        id = word_from_bytes(d[2],d[3]);

        switch (subspec) {
        // In case of asynchronous Receive PDOs,
        // we get an indication. Nevertheless the
        // data inside the PDO will be mapped to
        // the corresponding DPM-locations and will
        // be available for DPM-access functions.
        case MS_CMS_DATA_ind:
                // Depending on the returned ID ...
                if ( OBJ_DIC_RANGE_RPDO(id) )
                {
                        // We can calculate the OD entry with the corresponding mapping
                        // information
                        printf ("\tAsynchronous RPDO #%d notification\n",
                                  (id - 0x13ff));
                        printf ("\t\tData: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x,",
                                  d[4], d[5], d[6], d[7], d[8], d[9], d[10]);
                        printf ("0x%x\n", d[11] );

                        // Checking which objects are mapped to this PDO
                        pdoData = 4;
                        subIndex = 1;

                        while ( (retVal = readODUnsigned32(&myVars,
                                  (WORD_t)(id + 0x200), subIndex++, &mapping)) == TRUE)
                        {
                                printf ("\t\tData belongs to mapped object at");
                                printf ("\n\t\tindex 0x%x, subindex 0x%x",
                                        (int)((mapping >> 16) & 0xffff),
                                        (int)((mapping >> 8) & 0xff));
                                printf (" (%d valid byte(s))\n",
                                        (int)((mapping & 0xff) / 8));

                        }
                } else {
                        showMessage ("\tUnhandled CMS data indication received :\n\t",
                                         msg);
                }
                break;
        default:
                showMessage ("\tUnhandled CMS data message received :\n\t", msg);
        }

        break;
```

The CMS service is responsible for transferring PDO data to the host. It uses indicator M_CMS_DATA in its firmware indications. The sub-specifier MS_CMS_DATA_ind is used to signal incoming data for a PDO. In this case Bytes 3 and 4 contain the object dictionary entry in which the communication parameters of the corresponding PDO are defined. With macro OBJ_DIC_RANGE_RPDO can be checked whether the data package really contains data of a RPDO (or of another communication object). If this is true, the data of the original CAN frame is stored in bytes 5 to 12 of this data package.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

If you decide to receive PDO data asynchronously, the best way would be to work with the raw data which you have already present in the message handler at receive time. Below we are describing how to get the information about the process variables which belong to the received PDO data. But this can only be done in a relative time-consuming manner, because all this information needs to be read from the object dictionary which resides on the device.

For Janz CAN devices a variable mapping is supported, which means, that another node which has master capabilities is able to change the PDO mapping of our node during runtime! So in theory it makes no sense to hold the configuration settings on the host, because it might be changed on the CAN module after the initialization without host notification. This is the reason why the application can only get valid information about the mapped process variables from the object dictionary.

For getting information about the PDO mapping function 'readODUnsigned32' is used in a while loop to read the corresponding mapping parameter of the RPDO (here RPDO #1). So the mapped entries of the object dictionary can be found out. These entries contain the indices and sub-indices in which the process variables are stored which possibly are changed through the data contained in the received CAN frame.

After starting the program waits for incoming data of the first RPDO. When another network node has sent a message with identifier 181_{hex} (385_{dez}) this data is issued in text format on the screen.



For sample code please see directory :

code / chapter6 / conventional / pdo_recv / pdo_recv.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.8 Receiving data from a asynchronous, requested PDO

The property 'transmission type' in the PDO's communication parameters determines whether the PDO is event-based or request-controlled. Data of requested PDOs is not automatically sent over the CAN bus when its contents has changed. Sending of data stored in PDOs of this type have to be triggered with a special CAN frame from the node which want to have this data. After that a synchronous or asynchronous CAN frame will be generated for this PDO from the provider of data.

The application programmer must pay attention that writing data to corresponding object dictionary entries is always possible, but this does not automatically generate a corresponding CAN frame on the bus. This will be done not before another node has sent a request for it.

A request for data stored in a RPDO can be sent with function 'triggerRPDO' or 'triggerRPDOMappedTo'. Function 'triggerRPDO' gets a pointer of a data buffer and the number of the RPDO which shall be received. It will generate a firmware request which causes the firmware to send a corresponding CAN frame on the bus (RTR message, RTR = remote transmission request). When using function 'triggerRPDOMappedTo' the index and sub-index value of the wanted process value needs to be used as function parameters. The function will automatically trigger the corresponding receive PDO (RPDO). If there is a node in the network providing data of this PDO a CAN frame containing the wanted data is sent after the request was received. A corresponding indication is forwarded to the message handler like for data of an asynchronous RPDO. For this the message handler path already described in the chapter above can be used.

```
// triggering PDO data by using of mapped index and sub-index values
// Attention:
// Data may belong to more than one mapped object, it depends on
// the mapping in the DCF file!!
while ((myVars.nodeState == OPERATIONAL) && (trials < 10))
{
        sleep (1);
        printf ("\tTriggering data for index 0x2000, sub-index 0x01 ... ");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        if (triggerRPDOsMappedTo (&myVars, 0x2000, 0x01)) {
                printf ("done.\n");
        } else {
                printf ("failed.\n");
        }
        // increase counter
        trials++;
}
```



For sample code please see directory :

code / chapter6 / conventional / pdo_request / pdo_request.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.9 Initialization of a master node

The initialization for a master node application is not much more difficult than for a slave node. If we start with the 'init_slave' example, we only need to add a function call 'InitCalMaster' to make our application a CANopen master application. We did this in function 'canopen_iniit'.

```
        // Set endianess of the host machine
        printf ("\tSetting endianess of host machine ... ");
        SetHostEndianess(&myVars.msgBufferTx, COP_DPM_LITTLE_ENDIAN);
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot create list for cyclic sending\n");
                printf ("Progam aborted!\n\n");
                return (-3);
        }
        printf ("done\n");

        // initialize VMOD-ICAN3 module as CANopen slave
        initCanOpenAlt(&myVars);

        // create data to install CAL master
        printf ("\tInstalling CAL master ... ");
        InstallCalMaster(&myVars.msgBufferTx);
        // send message
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot install CAL master\n");
                printf ("Progam aborted!\n\n");
                return (-3);
        }
        printf ("done\n");

        // return success
        return 0;
}
```

Because we are a CANopen master now, we have to switch the bus into the 'OPERATIONAL' state in our own application and need not longer to wait until another node has done it for us! So we add also a call of function 'NmStartRemoteNodeReq' after function canopen_init has returned successfully and after we had switched our device 'BusOn'. Nevertheless we left the code for waiting for the state change in our sample application to show you, that state maintenance still works.

```
        // PRE-OPERATIONAL state reached
        // ============================

        // Normally we now have reached the state PRE-OPERATIONAL. It's time to
        // implement communication, which is typically done in this mode (SDO
        // transfer for example)

        printf ("\n\n\tReady for sending and receiving data !\n\n");
        printf ("IMPORTANT:\n");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
printf ("\tBecause we are master now, the state change will be made in\n");
printf ("\tour own application! So, we do not need to wait until the state\n");
printf ("\tchange has been done from another node!!\n\n");

// Starting CAN bus
printf ("\tStarting the CAN bus ... ");
NmStartRemoteNodeReq (&myVars.msgBufferTx, 0);
// send message
if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Cannot install CAL master\n");
        printf ("Progam aborted!\n\n");
        return (-3);
}
printf ("done\n");

// Waiting on OPERATIONAL state
// ============================

// Waiting on OPERATIONAL state switch is done with a while loop, which
// is getting the state cyclically. The device state is maintained from
// the CANopen library automatically!
//
// This loop should continue automatically after a NmStartRemoteNodeReq,
// call because as master we changed the node states ourself!

while ((myVars.nodeState != OPERATIONAL) && (timeout < 30)) {
        // wait a further second and try again
        sleep(1);
        // increase timeout counter
        timeout++;
}

// check whether timeout occurred
if (timeout < 30) {
        printf ("\n\tState is OPERATIONAL!\n");
} else {
        printf ("Application timed out!!\n");
        printf ("ERROR : No NMT state change notified before timeout\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}

// It's time to work with CANopen communication objects now (PDOs)
// In this example we only wait a short time for signaling, that
// the program is active now.
/* time to receive data */
printf ("\nRECEIVING DATA:\n");
```

Calling function'NmStartRemoteNodeReq' causes the firmware to send a CAN frame which causes other nodes to change the bus status to 'OPERATIONAL'. A node number is needed as function parameter. If this number is set to 0, the status of all nodes in the network will be changed. Otherwise only the status of the node with the entered number will be changed. Accessing single nodes in the network cannot be done without a remote node database. This is a small data storage area saving information about the nodes in the network. We'll come to this in another chapter below in this document

After that the initialization of a master node has finished. The main part of master functionality (for example NMT service functions) is available in the application now. But there are still some functions for what additional initialization work is needed. We exclude these functions from the master initialization because not all application will need them. To guarantee a lean and fast design, the application writer can decide himself, whether he needs these functions or not.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter6 / conventional / init_master / init_master.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.10 Emergency CAN frames

Emergency CAN frames are signalling exception situations on the CAN bus. For example, an emergency CAN frame is generated when errors occur on a node connected to the CAN bus.

6.10.1 Sending emergency CAN frames

Sending emergency CAN frames is done by using function 'triggerEMCY'. The parameters of this function are the error code, a 32 bit value with additional error information, an error register value and a pointer to a 5 byte long buffer with manufacturer specific additional information about the error situation on the node. Valid error register values are defined in macros in the header file 'copdefs.h'. Values defined in the CAN specification have to be used for the error code. The values in the buffer can be freely set. In the following example an emergency CAN frame with error code 0x1000 and an error register value 'GENERIC_ERROR' is generated. All bytes of the additional error information and the manufacturer specific buffer are set to 0.

```
// check whether timeout occurred
if (timeout < 30) {
        printf ("\n\tState is OPERATIONAL!\n");
} else {
        printf ("Application timed out!!\n");
        printf ("ERROR : No NMT state change notified before timeout\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}

// Sending a EMCY message on the CAN bus
printf ("\tSending an EMCY message ... ");
myVars.myManSpErrorField.mSpErrData[0] = 0;
myVars.myManSpErrorField.mSpErrData[1] = 0;
myVars.myManSpErrorField.mSpErrData[2] = 0;
myVars.myManSpErrorField.mSpErrData[3] = 0;
myVars.myManSpErrorField.mSpErrData[4] = 0;
if (!triggerEMCY(&myVars, 0x1000, 0x0000, GENERIC_ERROR, &myVars.myManSpErrorField))
{
        printf ("failed\n");
} else {
        printf ("done\n");
}

printf ("\n\n");
printf ("PROGRAM SHUTDOWN: \n");
```

6.10.2 Receiving emergency CAN frames

What a surprise, received emergency CAN frames end up in the message handler! But they will not be forwarded to the host application for default. Forwarding needs to be switched on by creating a so called emergency client object on the device. For this function ‘createEMCYClient’ is used. In our example we added this function call to our ‘canopen_init’ function. You need a corresponding call of function ‘createEMCYClient for each node from which we would like to have emergency CAN frames in our message handler. In our example we do this for nodes 1 ... 127.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



Please note, that receiving emergency CAN frames is only possible if master capabilities are available.

```
        // Starting CAN bus
        printf ("\tStarting the CAN bus ... ");
        NmStartRemoteNodeReq (&myVars.msgBufferTx, 0);
        // send message
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot install CAL master\n");
                printf ("Progam aborted!\n\n");
                return (-3);
        }
        printf ("done\n");

        // We want to be well informed about emergency
        // messages of all nodes on the network!
        printf ("\n\tCreating EMCY clients for all nodes in network ... ");
        for (nodeInNetwork = 1; nodeInNetwork <= 128; nodeInNetwork++)
        {
                createEMCYClient(
                                &myVars,
                                (BYTE_t)nodeInNetwork,
                                COBID_EMCY_DEFAULT_FROM_NODE_ID(nodeInNetwork));
        }
        printf ("done\n");

        // return success
        return 0;
}
```

Data packages containing emergency CAN frame data have the specifier M_CMS_DATA. We can get the object dictionary entry, which causes the actual indication, out of data bytes 3 and 4. We can also detect with this value whether the received firmware indication is an emergency message or not.

```
// Message which handles received SDOs/(asynchronous/acyclic)PDOs
case M_CMS_DATA:
        id = word_from_bytes(d[2],d[3]);

        // CANopen manager, emergency message handling of peer nodes
        if ( (id > EMCY_CALLBACK_BASE) && (id < (256 +  EMCY_CALLBACK_BASE)) ) {
                printf("\tEmergency from node %d arrived\n", (id - EMCY_CALLBACK_BASE));
                // evaluate error code
                printf("\t\tEmergency Error Code: 0x%x\n", d[4] + (d[5] << 8) );
                printf("\t\tError Register: 0x%x\n", d[6]);
                printf("\t\tManufacturer Specific Error Field: 0x%x, 0x%x, 0x%x,0x%x, 0x%x\n",
                           d[7], d[8], d[9], d[10], d[11]);

                break;
        }

        switch (subspec)
        {
        // In case of asynchronous Receive PDOs,
        // we get an indication. Nevertheless the
```

The macro EMCY_CALLBACK_BASE defines the index in which emergency message for node 1 are stored in the object dictionary. If the value in bytes 3 and 4 contains an index between the 256 possible index numbers for EMCY messages, the added path in the message handler will be executed.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter6 / conventional / emcy / emcy.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.11 Synchronous PDOs / CAN frames for synchronization (SYNC)

Synchronous PDOs are only sent over the CAN bus when a synchronisation CAN frame was received before. There is not so much to pay attention for application programmers in this case because PDO handling is mainly done by the firmware. Writing and reading object dictionary entries are done with functions which are already explained in chapters above. Also sending SYNC CAN frames is mainly handled by the firmware with the best possible time accuracy. Sending SYNC frames must be started with function 'startSync'. We will do this in function 'canopen_init'. After that the firmware is responsible for sending SYNC frames with a time distance between them which was defined in the DCF file.



Please note, that sending SYNC CAN frames is only possible if master capabilities are available.

```
        // Starting CAN bus
        printf ("\tStarting the CAN bus ... ");
        NmStartRemoteNodeReq (&myVars.msgBufferTx, 0);
        // send message
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot install CAL master\n");
                printf ("Progam aborted!\n\n");
                return (-3);
        }
        printf ("done\n");

        // Start sending of SYNC messages
        printf ("\tStart sending SYNC messsages ... ");
        if (!startSync(&myVars))
        {
                printf ("failed\n");
        } else {
                printf ("done\n");
        }

        // make SYNC message produced by ourself visible in message handler
        printf ("\tMake SYNC messages visible in message handler ... ");
        COPSetSyncToHostSending(&myVars.msgBufferTx, 1);
        // send message
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot set SyncToHostSending\n");
                printf ("Progam aborted!\n\n");
                return (-3);
        }
        printf ("done\n");

        // return success
        return 0;
}
```

Function 'COPSetSyncToHostSending' can be used, to specify whether the SYNC event shall also be signalled to our own message handler. If 1 was set function parameter, the firmware will send an indication to the message handler each time a SYNC frame was sent. Otherwise (default) the SYNC frames are sent without an indication in the message handler.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CAN identifier and time distance (communication cycle period) with which the SYNC frames will be sent on the bus are defined in the device configuration file (entries 1004_{hex}, 1005_{hex} and 1006_{hex}).

```
DefaultValue=0x00010001
PDOMapping=0

[1005]
SubNumber=0
ParameterName=COB-ID SYNC message
ObjectType=0x07
DataType=0x0007
AccessType=rw
DefaultValue=0xc0000080
PDOMapping=0

[1006]
SubNumber=0
ParameterName=communication cycle period
ObjectType=0x07
DataType=0x0007
AccessType=rw
ParameterValue=1000000
DefaultValue=0
PDOMapping=0

[1007]
SubNumber=0
ParameterName=synchronous window length
ObjectType=0x07
DataType=0x0007
AccessType=rw
DefaultValue=0x00
PDOMapping=0

[1008]
SubNumber=0
```

The entry for the CAN identifier (0x1004) also defines whether the application is a SYNC consumer (bit 30 not set) or a SYNC producer (bit 30 set). You will find more detailed descriptions about the contents of this entry in the CANopen specification.

Index 1007_{hex} specifies a time value in between a SYNC consumer node has to send the response after SYNC event occurrence. But it is not defined what to do when this time has elapsed and the response could not be sent or what to do when the time has elapsed and a response from another node was received. Each application programmer has to decide this himself in his own application.

Furthermore we added a call of function 'getchar' in our example to avoid that our sample program is stopped before SYNC frames could be received or sent. Pressing the RETURN button will end the execution of our sample program.

After starting on the one hand it should be possible to receive the SYNC frames generated by our sample program with another node (for example a CANlook program together with a further Janz CAN module). On the other hand each SYNC frame should generate a text issue 'Unhandled CMS data indication' on the screen where the sample program prints out its text messages. Everybody knows the reason why this happens. Of course, we haven't a corresponding path in our message handler for this message type. Following you will find the code segment to implement this functionality.

```
// Message which handles received SDOs/(asynchronous/acyclic)PDOs
case M_CMS_DATA:
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
id = word_from_bytes(d[2],d[3]);

// CANopen slave/manager, SYNC message handling
if (id == 0x1005)
{
        printf ("\tSYNC-message arrived\n");
        break;
}

switch (subspec)
{
```

After adding these lines of code the sample program should issue the message 'SYNC message arrived' each time a SYNC message was generated or received.



Receiving SYNC frames is independent from whether we have a master or a slave implementation. But sending SYNC frames is only available on master nodes.



For sample code please see directory :

code / chapter6 / conventional / sync / sync.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.12 Service data objects (SDO)

Service data objects (SDOs) are for directly accessing object dictionary entries of another node in the network. SDO transfer is possible when the network is in PRE-OPERATIONAL mode. So SDOs normally should be used to do configuration work before the network members will start their daily work. SDO transfer is also available in the OPERATIONAL mode if some additional configuration work is done in our application. This is suitable for transferring a large amount of data via one single CAN identifier for example.

For each value which should be read from a remote object dictionary a request has to be sent to the node using a pre-defined identifier. After that the node sends back a response containing the wanted data. The pre-defined identifier has larger numbers like used for PDOs. So CAN frames used for SDO transfer have a lower priority and generate more bus load because of the needed request message for reading. In most cases using SDOs will probably be slower than transferring data with PDOs.

To be able to access entries in an object dictionary of a special remote node, SDOs needs to be transferred only between two nodes. This means, that SDO transfer works like a peer-to-peer connection. Each SDO request is sent only to one special node in the CAN network which is addressed over its node id. This is done by using one special pre-defined CAN identifier for each node number for sending and receiving SDO transfer messages. Only the addressed node will answer to this if it exists. If it does not exist or cannot be reached, the request will not be answered. In this case a timer expires after configurable time duration and generates a timeout on the Janz CAN device.

A default SDO is available on a Janz CAN device which needs not to be configured in the DCF file. With this feature the programmer is able to access remote object dictionary entries with a minimum of effort.

For using the default SDO only a client object needs to be created during the CANopen initialization. This is done using function ‘createVariableSDOClient’. No configuration work is needed in the DCF file for this. Entries for index 1200_{hex} need not to be present in the DCF.

```
// create data to install CAL master
printf ("\tInstalling CAL master ... ");
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        InstallCalMaster(&myVars.msgBufferTx);
        // send message
        if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
                printf("ERROR : Cannot install CAL master\n");
                printf ("Progam aborted!\n\n");
                return (-3);
        }
        printf ("done\n");

        // Creating a SDO client with variable COB-IDs for
        // access to different (default-) SDOs of network nodes
        printf ("\tCreating a variable SDO client ...");
        nodeInNetwork = 1;
        createVariableSDOClient(
                &myVars,
                COBID_CLIENT_SDO_FROM_NODE_ID_TX(nodeInNetwork),
                COBID_CLIENT_SDO_FROM_NODE_ID_RX(nodeInNetwork));
        printf ("done\n");

        // return success
        return 0;
}
```

After that SDOs can be written and read with functions ‘readSDO’ and ‘writeSDO’. These functions automatically access the node either initially specified in function ‘createVariableSDOClient’ or the one set with function ‘adjustVariableClientSDOCobId’.

In our example we programmed a node scan mechanism with help of the default SDO. The corresponding client object is created in function ‘canopen_init’. The node scan is made in a loop in the main program.

```
        printf ("\n\nSCANNING for nodes while reading SDOs:\n");
        printf ("\n\t");

        // scanning among all posiibly nodes in a CANopen network
        for (nodeInNetwork=1; nodeInNetwork<128; nodeInNetwork++)
        {
                // chack whether node id is the id of our own node
                if (nodeInNetwork == myVars.nodeId)
                {
                        // no scanning needed, we hopefully know that we exist:
                        // COGITO, ERGO SUM !

                        // print out a message
                        printf("\n\n\tOf course, local node %d exists.",nodeInNetwork);
                        printf("\n\n\t");

                } else {
                        // set the correct COB-Id to be used in the SDO client object
                        // depending on which node number shall be accessed
                        adjustVariableSDOClientCobId(
                                &myVars,
                                COBID_CLIENT_SDO_FROM_NODE_ID_TX(nodeInNetwork),
                                COBID_CLIENT_SDO_FROM_NODE_ID_RX(nodeInNetwork));

                        // try to read index 0x1000 from the object dictionary of the
                        // remote node (index 0x1000 has no sub-indices, in this case the
                        // sub-index function argument has to set to 0 !)
                        if (readSDO(&myVars, 0x1000, 0))
                        {
                                // SDO response arrived, print out a message
                                printf ("\n\n\tSDO response arrived, ")
                                printf ("node %d exists.\n\tSDO-response-data:",nodeInNetwork);

                                // received data from the SDO transfer is
                                // stored in member 'sdoValidData' of the
                                // 'myVars' global variable. We print out this data here.
                                for (i = 0; i < myVars.sdoValidData; i++)
                                {
                                        printf(" 0x%02x", (BYTE_t)(myVars.sdoRxBuffer[i]) );
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                        }
                        printf("\n\n\t");
                } else {
                        // no peer node response, timeout occurred
                        printf("[%03i]", nodeInNetwork);

                        // after 10 nodes make a line feed
                        if ((nodeInNetwork % 10) == 0)
                        {
                                printf ("\n\t");
                        }
                }
        }
}
```

We implemented a loop among all possible node numbers in a CANopen network which tries reading a value from index 1000_{hex} which is mandatory defined in the CANopen specification. If reading was successful we assume that the node exists in the network.



Function 'readSDO' cannot be used for reading entries of our local node's object dictionary. For this other functions are provided (for example function 'readOD'. So we assume that our node exists without further checking!

Function 'writeSDO' can be used to write values to index items in an object dictionary of a remote node. If the index does not exists or is a read only entry, writing a value will end in an SDO abort which causes function 'writeSDO' to return with an error result. Function 'adjustVariableSDOClientCobId' is used to set the wanted node address.

```
// set the correct COB-Id to be used in the SDO client object
// depending on which node number shall be accessed
nodeInNetwork = 1;
adjustVariableSDOClientCobId(
        &myVars,
        COBID_CLIENT_SDO_FROM_NODE_ID_TX(nodeInNetwork),
        COBID_CLIENT_SDO_FROM_NODE_ID_RX(nodeInNetwork));

printf ("\n\tCopying value %d transmit buffer ... ", UI32value);

// copy the value which shall be written to the transmit buffer
memcpy (myVars.sdoTxBuffer, &UI32value, sizeof(Unsigned32));

printf ("done\n");
printf ("\tWriting the value to index 1000 of node %i ... ", nodeInNetwork);

// write the value via SDO to index 0x1000 and sub-index 0,
// and check the execution result
if (writeSDO(&myVars, 0x1000, 0, sizeof(Unsigned32)))
{
        printf("succeeded\n");
} else {
        printf("failed\n");
}
```

Accessing our own object dictionary from other nodes in the network is provided without any additional work. It is completely handled by the CANopen firmware and the CANopen library. It is important to know that access to the object dictionary of your own application via SDOs is possible only if your node is in PRE-OPERATIONAL mode. Of course, your application has to be running for having an object dictionary on the Janz CAN device at all.

To change this behaviour you have to re-configure the underlying CAL object in the firmware which can be made with function 'ConfigDomainObject'. This function uses the service 'CMSObjectConfReq' to change configuration settings of the underlying CAL object. Please see the ICAN3 Software Reference Manual to get more information about this.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



For sample code please see directory :

code / chapter6 / conventional / scannodes / scannodes.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13 Extended network management

The master part of the Network Management service (NMT service) on the Janz CAN device uses a data buffer called remote node database to store data objects for each remote CANopen node which has to be handled. We will also describe several CANopen firmware features in this chapter for which the remote node database is needed.

Each entry in this database has to be created by the host application. For this purpose the function 'CreateRemoteNode' is used. In this chapter we will continue with the 'scannodes' example which was present in chapter 'Service Data Objects (SDOs)' first. This program implements a CANopen master which is necessary for the following remote node database examples. Furthermore a loop was programmed which enumerates all CANopen nodes connected to the bus using the SDO transfer. We will add some code lines in this loop with which an entry in the remote node database will be created for each found node.

```
// scanning among all posiibly nodes in a CANopen network
for (nodeInNetwork=1; nodeInNetwork<128; nodeInNetwork++)
{
        // chack whether node id is the id of our own node
        if (nodeInNetwork == myVars.nodeId)
        {
                // no scanning needed, we hopefully know that we exist:
                // COGITO, ERGO SUM !

                // print out a message
                printf("\n\n\tOf course, local node %d exists.",nodeInNetwork);

                // creating remote node DB entry
                sprintf(lmtNodeName,"node%03d",nodeInNetwork);
                printf ("\n\tCreating remote node DB entry %s ... ", lmtNodeName);
                if (createRemoteNode(&myVars, lmtNodeName,   (WORD_t)nodeInNetwork,
                        (WORD_t)1000, (WORD_t)COBID_GUARD_DEFAULT_FROM_NODE_ID(nodeInNetwork)))
                {
                        printf ("done\n");
                } else {
                        printf ("failed\n");
                }

                printf("\n\n\t");

        } else {

                // set the correct COB-Id to be used in the SDO client object
                // depending on which node number shall be accessed
                adjustVariableSDOClientCobId(
                        &myVars,
                        COBID_CLIENT_SDO_FROM_NODE_ID_TX(nodeInNetwork),
                        COBID_CLIENT_SDO_FROM_NODE_ID_RX(nodeInNetwork));

                // try to read index 0x1000 from the object dictionary of the
                // remote node (index 0x1000 has no sub-indices, in this case the
                // sub-index function argument has to set to 0 !)
                if (readSDO(&myVars, 0x1000, 0))
                {
                        // SDO response arrived, print out a message
                        printf("\n\tSDO response arrived, node %d exists.\n");
                        printf ("\tSDO-response-data:",nodeInNetwork);

                        // received data from the SDO transfer is stored in member
                        // 'sdoValidData' of the
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                        // 'myVars' global variable. We print out this data here.
                        for (i = 0; i < myVars.sdoValidData; i++)
                        {
                                printf(" 0x%02x", (BYTE_t)(myVars.sdoRxBuffer[i]) );
                        }

                        // creating remote node DB entry
                        sprintf(lmtNodeName,"node%03d",nodeInNetwork);
                        printf ("\n\tCreating remote node DB entry %s ... ", lmtNodeName);
                        if (createRemoteNode(
                                &myVars,
                                lmtNodeName,
                                (WORD_t)nodeInNetwork,
                                (WORD_t)1000,
                                (WORD_t)COBID_GUARD_DEFAULT_FROM_NODE_ID(nodeInNetwork)))
                        {
                                printf ("done\n");
                        } else {
                                printf ("failed\n");
                        }

                        printf("\n\n\t");

                } else {
                        // no peer node response, timeout occurred
                        printf("[%03i]", nodeInNetwork);

                        // after 10 nodes make a line feed
                        if ((nodeInNetwork % 10) == 0)
                        {
                                printf ("\n\t");
                        }
                }
        }
}
```

After the remote node database was successfully created, we get the settings from this using function 'NmGetRemNodes' for performing a firmware request This request causes the firmware to transfer the data from the remote node database to the message handler. After that we print out this data using the corresponding 'format_xxxx' library function.

```
        // Messages from the NMT-Master
        case M_NMTM:
                switch (subspec)
                {
                        /* Data for printing NMT Remote Nodes database */
                        #define MAX_RNODES_LEN          4096
                        static BYTE_t rnodes_data[MAX_RNODES_LEN];
                        static int rnodes_len = 0;
                        static int rnodes_pos = 0;
                        static int l;

                case MS_NMTM_EVENT_ind:
                        showMessage("master> ", msg);
                        return;

                case MS_NMTM_GET_RNODES_con:
                        if( msg->len <4 )
                                return;

                        dlen = word_from_bytes(d[2],d[3]);
                        if (dlen & 0x8000)      /* start message? */
                        {
                            dlen &= 0x7fff;
                            l     = (int)dlen;
                            if (l > MAX_RNODES_LEN){
                                printf("sorry girl: to few mem allocated\n");
                                return;         /* error !!! ??? */
                            }
                            rnodes_len = l;
                            rnodes_pos = 0;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
            }
            l = MSGLEN - 4;
            l = dlen < l ? (int)dlen : l;
            if (rnodes_pos + l > rnodes_len)
            {
                rnodes_len = 0;
                return;                  /* error !!! ??? */
            }
            MY_MEMCPY((char *)&rnodes_data[rnodes_pos],
                                         (char *)&d[4], l);
            rnodes_pos += l;

            if (rnodes_pos >= rnodes_len) /* done? */
            {
                printf_nmtm(stderr, rnodes_len, &rnodes_data[0]);
                rnodes_len = 0;
            }
            return;

        default:
            return;
        }
        break;
```



Please note, that in version 4.0 of the CANopen specification some changes were made which are not compatible with older versions. To support this features but not to loose the compatibility to older versions of our library functions we decided to create a new function 'CreateRemoteNodeCOP40' for creating the remote node database entry. This function has to be used instead of function 'CreateRemoteNode' if the remote node, you want to work with, is a CANopen 4.0 node (important for heartbeat for example). The new function is used quite similar to the older one but has a little bit different function arguments! Please refer to the ICAN3 software reference manual for more detailed information about this function.



For sample code please see directory :

code / chapter6 / conventional / rnodedb / rnodedb.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13.1 Switching single nodes into OPERATINAL mode

For switching all nodes into the OPERATIONAL status, no remote node database is needed, because this is done with one CAN message which is received from all nodes in the network. So no further information is needed from the firmware to execute this command. This was already described in a chapter above.

A remote node database entry is needed for accessing single nodes this way. After creating such an entry with function 'CreateRemoteNode' or 'CreateRemoteNodeCOP40' for CANopen 4.0 nodes function 'NmStartRemoteNodeReq' can be used like described in chapter 'Initializing a master node', but with the node number as second function argument instead of the 0. This will generate a CAN message on the bus which will only switch the wanted remote node into the OPERATIONAL status.

A remote node database is needed, because the node status returned by the remote node can not be stored internally. That makes network maintenance impossible.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

6.13.2 Node guarding protocol features

Service and protocols of the node guarding are used to determine whether remote nodes are still alive or not longer responding. RTR frame are cyclically sent on the network from a NMT (Network management) master. For each remote CANopen node a separate identifier is used depending on its node id (700_{hex} + NodeId). The remote node has to send the CAN frame for response within a defined time. This time value is called 'node guarding time' and needs to be defined with index $100C_{hex}$ in the object dictionary of the remote node (not in our OD!).

To be able to check whether the master is alive the slave node can implement a mechanism which is called 'life guarding'. For this index value $100D_{hex}$ defines a life time factor. If you multiply the life time factor with the node guarding time, the result value defines a time in between the master node has to contact the slave with a RTR frame. If the RTR request from the master is missing longer than this time, the slave node treats this as an error situation (master maybe not longer alive or connection to master not longer present). We will discuss both, 'node guarding' and 'life guarding', in the coming chapters.

Writing a NMT master application is, of course, also possible with Janz CAN devices! For that we need CANopen master capabilities and a remote node database entry for each node which shall be guarded. After that we only need to start the node guarding, the guarding protocol is handled by the firmware.

Like already explained several times ago, it is not possible to explain completely in this document how node guarding works. Please take also a look to the CANopen specification and other literature for more detailed information about the node guarding protocol.

6.13.2.1 Node guarding for an NMT slave

Implementing node guarding in a NMT slave application is as easy as it can be, because a simple CANopen slave application is automatically able to work as NMT slave and therefore to respond to node guarding requests (RTR frames) if the corresponding DCF entries $100C_{hex}$ are defined. If we start with our application used for chapter ‘Initializing a slave application’, we only need to add the following entries to the corresponding DCF file. After that the node will respond to node guarding requests.

```
DefaultValue=1.0
PDOMapping=0

[100C]
SubNumber=0
ParameterName=guard time
ObjectType=0x07
DataType=0x0006
AccessType=rw
DefaultValue=0
ParameterValue=1000
PDOMapping=0

[100D]
SubNumber=0
ParameterName=life time factor
ObjectType=0x07
DataType=0x0005
AccessType=rw
DefaultValue=0
ParameterValue=3
PDOMapping=0
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
[1014]
SubNumber=0
ParameterName=COB-ID EMCY message
```

We defined a node guarding time of 1000 milliseconds in our example which means that the master should send an RTR each second. We have defined a life time factor of 2 in index $100D_{hex}$ of the DCF which means that the node will signal an error situation after 1000ms * 2 = 2000ms = 2 seconds.

If also index $100D_{hex}$ is defined, than the node will expect node guarding RTR frame in a defined time. If such a RTR frame is missing, an error indication will be generated on the node. At the moment a missing node guarding requests is the only possibility that remote node errors can occur. Therefore the corresponding indication in the message handler contains no additional data. Of course, we need to write an own execution path in our message handler for handling node guarding error situations. In fact, this is the same path also described in chapter 'CANopen error handling'.

The firmware will send a 'M_NMTS' indication containing value 'NE_REMOTE_OCCURRED' in data byte 2 each time a node guarding error is detected. Furthermore the same indication will be sent each time when a node guarding request will be received again after an error situation. But this indication will have value 'NE_REMOTE_RESOLVED' in data byte 2 for signalling that the error is not longer present. So, your application will receive indications each time node guarding errors have occurred and each time when they have gone. Please decide yourself, how you to handle both situations in your application.



For sample code please see directory :

code / chapter6 / conventional / nodeguarding / ng_slave.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



For sample code please see directory :

code / chapter6 / conventional / nodeguarding / ng_slave.dcf

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13.2.2 Node guarding for an NMT master

Node guarding for a NMT master application is not much more complicated than using node guarding in a NMT slave application. You will need CANopen master capabilities and a remote node database for storing status information from the remote nodes.

Please note, that function 'createRemoteNode' defines an own node guarding time for each node found in the network (during network scan loop). You can either use the same value for all nodes like shown in the example (1000 milliseconds, marked in red colour below) or you can define a different value for each node depending on their node ID for example. After that the node guarding needs to be started for each single node separately with function 'NmStartGuardRemoteCOPNodeReq'. This must be done before the network is switched into OPERATIONAL state with command 'NmStartRemoteNodeReq'. Otherwise this function has no effect and the node guarding will not be started.

```
// try to read index 0x1000 from the object dictionary of the
// remote node (index 0x1000 has no sub-indices, in this case the
// sub-index function argument has to set to 0 !)
if (readSDO(&myVars, 0x1000, 0))
{
       // SDO response arrived, print out a message
       printf("\n\tSDO response arrived, node %d exists.\n\tSDO-response -data:",
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        nodeInNetwork);

// received data from the SDO transfer is stored in member 'sdoValidData' of the
// 'myVars' global variable. We print out this data here.
for (i = 0; i < myVars.sdoValidData; i++)
{
        printf(" 0x%02x", (BYTE_t)(myVars.sdoRxBuffer[i]) );
}

// creating remote node DB entry
sprintf(lmtNodeName,"node%03d",nodeInNetwork);
printf ("\n\tCreating remote node DB entry %s ... ", lmtNodeName);
if (createRemoteNode(&myVars, lmtNodeName,  (WORD_t)nodeInNetwork,
        (WORD_t)1000, (WORD_t)COBID_GUARD_DEFAULT_FROM_NODE_ID(nodeInNetwork)))
{
        printf ("done\n");
} else {
        printf ("failed\n");
}

// Start node guarding for all nodes
// adding remote node to node guarding
printf ("\n\tStarting node guarding for all nodes ... ");
NmStartGuardRemoteCOPNodeReq (&msg, nodeInNetwork);
// send message
if (can_send(myVars.fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot start node guarding\n");
        printf ("Progam aborted!\n\n");
        return (-4);
}
printf ("done\n");
```

So we have to setup a NMT master in the firmware. After that we can simply start the node guarding using function 'NmStartGuardRemoteCOPNodeReq' and the firmware will do the work without the need of interaction. We will only receive firmware indication messages if error situations occur. This requires an own path in our message handler, of course! We do not print this source code here because it is too long and you probably know how it looks like. You will find it in the corresponding code example!

If you use a CAN analyser tool (CANlook, for example), you will see that the firmware is sending RTR messages autonomously. Either a remote node is responding before the defined node guarding time has expired or a M_NMTM indication will be sent to our message handler. The ‘subspec’ value will be 'MS_NMTM_EVENT_ind' in this case.



For sample code please see directory :

code / chapter6 / conventional / nodeguarding / ng_master.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



For sample code please see directory :

code / chapter6 / conventional / nodeguarding / ng_master.dcf

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13.3 Heartbeat protocol features

The heartbeat protocol defines an error control service which enables a heartbeat consumer to audit heartbeat producer without the need of remote frames. A heartbeat producer cyclically transmits a

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

heartbeat CAN frame. More than one heartbeat consumer can receive these messages in between a specified time window.

A Janz CAN device can work as a heartbeat producer as well as a heartbeat consumer. Programming a heartbeat consumer is a little bit more complicated than writing a heartbeat producer. Also both can be mixed so that an application will send and receive heartbeat CAN frames at the same time.

There exist two possibilities for configuring a heartbeat producer as well as for configuring a heartbeat consumer. You can either make the configuration settings in the DCF indices or you can code the settings hardly into your application. We will describe both possibilities, but we have only one example application for the producer and one for the consumer. Both application will check the object dictionary if the corresponding entries are present. If they are, the program will use these settings. If not, the program will use hardly coded default settings. So, we have both options handled in one example application.

A remote node database is needed only for heartbeat consumer applications. Heartbeat producer simply are sending the heartbeat CAN frames and do not make any checks. Therefore they do not need to store data about remote nodes!

6.13.3.1 Heartbeat producer features configured in the DCF

For implementing a heartbeat producer configured with the corresponding DCF entry only a small work has to be done! On the one hand the DCF entry needs to be added, of course! On the other hand heartbeat needs to be enabled.

For configuring heartbeat producer capabilities in a DCF file index 1017_{hex} is used. This index defines the heartbeat producer time which is the time distance between two sent heartbeat CAN frames. In our example we set this value to 1000 milliseconds.

```
[1017]
SubNumber=0
ParameterName=Producer heartbeat time
ObjectType=0x07
DataType=0x0006
AccessType=rw
DefaultValue=1000
PDOMapping=0
```

After that we have to call function 'enableHeartbeat' which will do the initialization work for us. This function takes the DCF file settings regarding heartbeat and transfers them to the firmware by generating the corresponding firmware requests. After calling 'enableHeartbeatCopSlv' the node starts sending heartbeat messages on the bus. This is independent from whether the node is in OPERATIONAL or in PRE-OPERATIONAL status!

```
printf ("\nHEARTBEAT:\n");
// checking whether OD entry 1017 exists
printf ("\tChecking for value of OD entry [1017] ... ");
readODUnsigned16 (&myVars, (WORD_t) 0x1017, (BYTE_t) 0, &u16Value) ;
if (u16Value != 0)
{
        // entry exists, nothing to do
        printf ("value is %i ms\n", u16Value);

        // enabling heartbeat
        printf ("\tEnabling heartbeat ... ");
        if (enableHeartbeatCopSlv (&myVars) == TRUE)
        {
                printf ("successful\n");
        } else {
                printf ("failed\n");
        }

} else {
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
            // entry does not exist, heartbeat will be installed without
            // DCF entries

            // Configuring timer for sending heartbeat messages
            printf ("value is %i ms\n", u16Value);
            printf ("Installing heartbeat producer, heartbeat producer time is %i ms... ",
                        hbProdTime);
            NsConfigureHeartbeat (&myVars.msgBufferTx, hbProdTime);
            if (can_send (myVars.fd, &myVars.msgBufferTx) <= 0)
            {
                  printf ("failed\n");
            } else {
                  printf ("successful\n");
            }

            // setting 'heartbeat enabled' flag for later usage
            myVars.heartbeatEnabled = 1;
      }
```

To be able to test both configuration options we provide two DCFs with the heartbeat producer example. The hb_prod.dcf file contains the needed entry. The file default.dcf contains a DCF without heartbeat settings. For this chapter you should use file hb_prod.dcf



For sample code please see directory :

code / chapter6 / conventional / heartbeat_producer / hb_prod.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



For sample code please see directory :

code / chapter6 / conventional / heartbeat_producer / default.dcf

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13.3.2 Heartbeat producer features configured in the application

Implementing a heartbeat producer without settings in the DCF is not more difficult than with! Using a DCF entry for configuration is only much more flexible, because you do not need to change your program code if you want to change the heartbeat producer time. On the other hand no one can make changes (and errors!) without re-compiling the application. Sometimes this can be much more important than the flexibility.

For setting the heartbeat producer time only a call of function 'NsConfigureHeartbeat' is needed. This function will prepare a firmware request with the heartbeat producer time given in the function argument. After transferring this request to the firmware the firmware will start sending heartbeat messages.

```
      printf ("\nHEARTBEAT:\n");
      // checking whether OD entry 1017 exists
      printf ("\tChecking for value of OD entry [1017] ... ");
      readODUnsigned16 (&myVars, (WORD_t) 0x1017, (BYTE_t) 0, &u16Value) ;
      if (u16Value != 0)
      {
            // entry exists, nothing to do
            printf ("value is %i ms\n", u16Value);

            // enabling heartbeat
            printf ("\tEnabling heartbeat ... ");
            if (enableHeartbeatCopSlv (&myVars) == TRUE)
            {
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                printf ("successful\n");
        } else {
                printf ("failed\n");
        }

} else {
        // entry does not exist, heartbeat will be installed without
        // DCF entries

        // Configuring timer for sending heartbeat messages
        printf ("value is %i ms\n", u16Value);
        printf ("Installing heartbeat producer, heartbeat producer time is %i ms ... ",
                        hbProdTime);
        NsConfigureHeartbeat (&myVars.msgBufferTx, hbProdTime);
        if (can_send (myVars.fd, &myVars.msgBufferTx) <= 0)
        {
                printf ("failed\n");
        } else {
                printf ("successful\n");
        }

        // setting 'heartbeat enabled' flag for later usage
        myVars.heartbeatEnabled = 1;
}
```

You can also use both mechanisms in parallel. You can start with a configuration stored in the DCF file and change this setting in the program by using function 'NsConfigureHeartbeat' for example.

To be able to test both configuration options we provide two DCFs with the heartbeat consumer example. The hb_prod.dcf file contains the needed entry. The file default.dcf contains a DCF without heartbeat settings. For this chapter you should use file default.dcf



For sample code please see directory :

code / chapter6 / conventional / heartbeat_producer / hb_prod.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



For sample code please see directory :

code / chapter6 / conventional / heartbeat_producer / hb_prod.dcf

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13.3.3 Heartbeat consumer features configured in a DCF

Implementing a heartbeat consumer configured via DCF entries requires the corresponding entries in our example DCF 'hb_cons.dcf' first. The sub-indices under 1016_{hex} define the heartbeat consumer time values for different remote nodes. Each sub-index defines the consumer heartbeat time for one remote node. Node number and time value are coded in a 32-bit value (time value in lower 16 bits, node number in higher 16 bits, whereby the highest byte is reserved). Please see the CANopen specification or documentation for getting more detailed information about the format. The table below lists the three nodes configured in our example DCF file with their heartbeat consumer time values and the corresponding 'DefaultValue' entries.

Node ID	Heartbeat consumer time	DefaultValue
4	1500 ms	0x000405DC
19	2500 ms	0x001309C4
20	800 ms	0x00140320

Table 18: Heartbeat consumer time settings in file hb_cons.dcf

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Below the entries which configure the heartbeat consumer in our DCF are listed:

```
[1016]
SubNumber=4
ParameterName=Consumer heartbeat time

[1016sub0]
ParameterName=number of entries
ObjectType=0x09
DataType=0x0007
AccessType=rw
DefaultValue=3
PDOMapping=0

[1016sub1]
ParameterName=1st Comsumer heartbeat time
ObjectType=0x09
DataType=0x0007
AccessType=rw
DefaultValue=0x000405DC
PDOMapping=0

[1016sub2]
ParameterName=2nd Comsumer heartbeat time
ObjectType=0x09
DataType=0x0007
AccessType=rw
DefaultValue=0x001309C4
PDOMapping=0

[1016sub3]
ParameterName=3rd Comsumer heartbeat time
ObjectType=0x09
DataType=0x0007
AccessType=rw
DefaultValue=0x00140320
PDOMapping=0
```

After having heartbeat configured in the DCF file we only need a call of function 'enableHeartbeat'CopSlv in our example program. This function will get all heartbeat consumer time values from the object dictionary and generate the needed firmware requests for configuring a heartbeat consumer.

```
printf ("\n\n");

// enabling heartbeat
printf ("\tEnabling heartbeat ... ");
if (enableHeartbeatCopSlv (&myVars) == TRUE)
{
        printf ("successful\n");
} else {
        printf ("failed\n");
}

// Starting CAN bus
printf ("\tStarting the CAN bus ... ");
NmStartRemoteNodeReq (&myVars.msgBufferTx, 0);
// send message
if (can_send(myVars.fd, &myVars.msgBufferTx) <= 0)
{
        printf ("failed\n");
        printf("ERROR : Cannot install CAL master\n");
        printf ("Progam aborted!\n\n");
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        return (-2);
}
printf ("done\n");
```



Please note, that a remote node database entry is needed for each node which shall be guarded with the heartbeat protocol. Furthermore the entries need to be created using function 'createRemoteNodeCOP40' because heartbeat is a CANopen 4.0 feature.

We use a variant of the example program already used in chapter regarding SDOs as basis for the current example. This program already implements a scan for CANopen nodes connected to the network. In addition we take a look at the stored object dictionary entries, after a node was found. If there is an entry present for the found node, we use these settings. Otherwise the node is configured for heartbeat from within the application using a default heartbeat consumer time value. We'll come to this in more detail later on in this chapter.

The heartbeat mechanism is activated automatically after configuration when the first heartbeat message was received from a node.

To be able to test both configuration options we provide two DCFs with the heartbeat consumer example. The hb_cons.dcf file contains the needed entries. The file default.dcf contains a DCF without heartbeat settings. For this example you should use file hb_cons.dcf



For sample code please see directory :

code / chapter6 / conventional / heartbeat_consumer / hb_cons.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



For sample code please see directory :

code / chapter6 / conventional / heartbeat_consumer / hb_cons.dcf

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13.3.4 Heartbeat consumer features configured in an application

Like already mentioned in chapters before the configuration of a heartbeat consumer can also be done from within an application. Using DCF entries for configuration is much more flexible, because you do not need to change your program code if you want to change the heartbeat consumer time of a node. On the other hand no one can make changes (and errors!) without re-compiling the application. Sometimes this can be much more important than the flexibility.

For the internal heartbeat consumer configuration the heartbeat consumer time of each node needs to be set insight the application. For this function 'NmSetHeartbeatConsumerTime' can be used. Furthermore we have to switch each remote node database entry from node guarding (default) to heartbeat. For this function 'NmSwitchGuardingOrHeartbeat' is used with 1 as function argument.

```
// if no entry was found, configure this entry manually with an default
// heartbeat consumer time
if (hbIndexFound != 1)
{
        // no entry found configuring heartbeat for this node,
        // we have to configure heartbeat manually
        printf ("\tNode found, for which no settings could be found in OD!\n");
        printf ("\t  => Configuring manually\n");

        // switch node settings internally from node guarding (default) to
        // heartbeat
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
            printf ("\tSwitching from Node Guarding to Heartbeat for node %i ... ",
                    nodeInNetwork);
            NmSwitchGuardingOrHeartbeat (&myVars.msgBufferTx, nodeInNetwork, 1);
            if (can_send (myVars.fd, &myVars.msgBufferTx) <= 0)
            {
                    printf ("failed\n");
            } else {
                    printf ("successful\n");
            }

            // set heartbeat consumer time
            printf ("\tSetting consumer heartbeat time for node %i to 500 ms ... ",
                    nodeInNetwork);
            NmSetHeartbeatConsumerTime (&myVars.msgBufferTx, nodeInNetwork, 500);
            if (can_send (myVars.fd, &myVars.msgBufferTx) <= 0)
            {
                    printf ("failed\n\n\t");
            } else {
                    printf ("successful\n\n\t");
            }
    } else {
            // no entry found configuring heartbeat for this node,
            // we have to configure heartbeat manually
            printf ("\tDCF settings found for heartbeat configuration\n");
            printf ("\t  => Calling 'enableHeartbeat' below\n\n\t");
    }
```



Please note, that a remote node database entry is needed for each node which shall be guarded with the heartbeat protocol. Furthermore the entries need to be created using function 'createRemoteNodeCOP40' because heartbeat is a CANopen 4.0 functionality.

We use a variant of the example program already used in chapter regarding SDOs which scans for CANopen nodes connected to the network. After a node was found, we take a look at the stored object dictionary entries. If there is an entry present for the found node, we use these settings. Otherwise the node is configured for heartbeat from within the application using a default heartbeat consumer time value.

The heartbeat mechanism is activated automatically after configuration when the first heartbeat message was received from a node.

To be able to test both configuration options we provide two DCFs with the heartbeat consumer example. The hb_cons.dcf file contains the needed entries. The file default.dcf contains a DCF without heartbeat settings. For this example you should use file default.dcf



For sample code please see directory :

code / chapter6 / conventional / heartbeat_consumer / hb_cons.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!



For sample code please see directory :

code / chapter6 / conventional / heartbeat_consumer / default.dcf

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.13.4 Bootup protocol features

Sending a bootup CAN frame with Janz CAN devices has to be initiated from within your application using the function 'NsCanopenBootupProtocol'. Neither a master implementation nor a remote

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

database entry is needed for that. Nevertheless we take the 'rnodedb' example as basis for the sample program in this chapter, because we will need the remote node database for receiving bootup indications which is also described below.

```
// check whether timeout occurred
if (timeout < 30) {
        printf ("\n\tState is OPERATIONAL!\n");
} else {
        printf ("Application timed out!!\n");
        printf ("ERROR : No NMT state change notified before timeout\n");
        printf ("Progam aborted!\n\n");
        return (-6);
}

// initiate sending of bootup message
printf ("\tGenerating BOOTUP message for own node id ... ");
NsCanopenBootupProtocol(&msg, myVars.nodeId);
// send message
if (can_send(myVars.fd, &msg) <= 0)
{
        printf ("failed\n");
        printf ("ERROR : Cannot change bus status\n");
        printf ("Progam aborted!\n\n");
        return (-2);
}
printf ("done\n");

// It's time to work with CANopen communication objects now (PDOs)
// In this example we only wait a short time for signaling, that
// the program is active now.
// time to receive data
printf ("\nRECEIVING DATA:\n");
```

That's all we need to do. After that a corresponding CAN frame for the bootup indication is generated on the CAN bus during the initialization phase of our example program.

For receiving bootup CAN frames sent by other nodes a remote node database is needed. Each node from which we would like to receive bootup indications needs an entry in our remote node database.



Please note that the CANopen 4.0 version 'CreateRemoteNodeCOP40' has to be used instead of 'CreateRemoteNode' because the bootup protocol is a CANopen 4.0 feature. Furthermore we need CANopen master capabilities when we would like to receive bootup indications from other nodes.

The scanning mechanism already implemented in several examples before is also used in this example to get the information about what nodes are present in the network. Furthermore we add a call of function 'NmSwitchGuardingOrHeartbeat' to switch the corresponding remote node database object setting from node guarding (default) to heartbeat. After that each received bootup message which comes from one of these nodes will be forwarded to the message handler of our application. We do not print the corresponding example code here because it's too long. Please take a look at the corresponding source module delivered with this tutorial in this case!

Of course, an additional path in the message handler is needed which handles the incoming indications for received bootup CA frames from other nodes. Please look at the code lines below to see which MACRO name can be used for the 'subspec' value in this case. Data byte 2 contains the node number from which the bootup indication was received.

```
// Messages from the NMT-Master
case M_NMTM:
        switch (subspec) {
                /* Data for printing NMT Remote Nodes database */
                #define MAX_RNODES_LEN          4096
                static BYTE_t rnodes_data[MAX_RNODES_LEN];
                static int rnodes_len = 0;
                static int rnodes_pos = 0;
                static int l;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        case M_NMTM_BOOTUP_ind:
                printf ("\t\tNMTM - BOOTUP indication received from node %i\n", d[2]);
                break;

        case MS_NMTM_EVENT_ind:
                showMessage("master> ", msg);
                return;
```

In this example we only added a 'printf' call for printing out a text message and the number of the remote node. After that for each incoming bootup indication a text message should be issued on the screen. Of course, you need to fill this code with life in your own application.



For sample code please see directory :

code / chapter6 / conventional / bootup / bootup.c

If the code directory was not delivered together with this document, take a look at our home page http://www.janz.de/as. The tutorial is available for download including all examples and sample code!

6.14 CANopen error handling

For each detected error on the CAN bus (error frames) the CAN controller increases an internal error counter by 8. If the error situation is obviously resolved, the error counter is decreased again by 1. The error counter can reach two error levels, the warning level which is at 96 per default and the passive level which is at 128. The firmware will send an event indication message to the application if one of the levels is reached or left.

Each application which has a message handler implemented is already prepared to handle error messages which come from the firmware. The only thing to do is adding a further path to the message handler's 'switch' command which will handle the error messages in a special way. The code lines below will show you a very simple way to get information about errors in the firmware.

In this step we will only show how an additional path for error handling is added to the message handler. Some of the error cases have additional information in their message about what is the reason for this message. You will get more detailed information about this reading the corresponding chapter in the ICAN3 Software Reference Manual.

```
// error indication message
case MS_NMTS_EVENT_ind:
        printf ("\tNode error event indication: ");
        // data byte 2 contains the error type
        switch (d[2])
        {
        case NE_LOCAL_OCCURRED:
                // error occurred on local node
                printf ("local error occurred\n");

                // data byte 3 contains the error code
                switch(d[3])
                {
                case EVT_BERR:
                        // CAN bus error, occurs if quota was installed for transferring
                        // each occurred error frame
                        // CAN bus error interrupt, occurs if warning or error level
                        // are reached
                        printf ("\t\tCAN bus error\n");
                        if( msg->len > 2)
                        {
                                // data byte 4 contains the controller
                                // type on the VMOD-ICAN module
                                switch(d[4])
                                {
                                case EVT_ERRT_82C200:
                                        printf("\t\tcontrol register value is 0x%2.2x\n", d[5]);
                                        printf("\t\tstatus register value is 0x%2.2x\n", d[6]);
                                        break;
```


© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                        case EVT_ERRT_SJA1000:
                                printf("\t\terror code value is 0x%2.2x\n", d[5]);
                                ShowSJA1000ErrorRegisterMeanings (d[5]);
                                printf("\t\tstatus register value is 0x%2.2x\n", d[6]);
                                ShowSJA1000StatusRegisterMeanings (d[6]);
                                printf("\t\tRX counter value is %d\n", d[7]);
                                printf("\t\tTX counter value is %d\n", d[8]);
                                break;
                        default:
                                printf ("\t\tCAN bus error interrupt\n");
                        }
                }
                break;

        case EVT_ERROR:
                // CAN bus error interrupt, occurs if warning or error level
                // are reached
                printf ("\t\tCAN error interrupt\n");
                if( msg->len > 2)
                {
                        // data byte 4 contains the controller type on
                        // the VMOD-ICAN module
                        switch(d[4])
                        {
                        case EVT_ERRT_82C200:
                                printf("\t\tcontrol register value is 0x%2.2x\n", d[5]);
                                printf("\t\tstatus register value is 0x%2.2x\n", d[6]);
                                break;
                        case EVT_ERRT_SJA1000:
                                printf("\t\tmode register value is 0x%2.2x\n", d[5]);
                                ShowSJA1000ModeRegisterMeanings (d[5]);
                                printf("\t\tstatus register value is 0x%2.2x\n", d[6]);
                                ShowSJA1000StatusRegisterMeanings (d[6]);
                                printf("\t\tRX counter value is %d\n", d[7]);
                                printf("\t\tTX counter value is %d\n", d[8]);
                                break;
                        default:
                                printf ("\t\tCAN bus error interrupt\n");
                        }
                }
                break;
        case EVT_OVERRUN:
                // overrun error,
                printf ("\t\tCAN bus overrun interrupt\n");
                break;
        case EVT_LOST_INTERRUPTS:
                // lost interrupts,
                printf ("\t\tICANOS lost interrupts\n");
                break;
        case EVT_QUEUE_FULL:
                // send queue overflow, occurrs when the application is transferring
                // send requests to one send queue and these requests cannot be send
                // as fast as they are inserted into the queue.
                printf ("\t\tsend queue overflow\n");
                break;
        default:
                printf("errc=%d",d[3]);
        }
        break;

case NE_LOCAL_RESOLVED:
        // error resolved on local node
        printf ("local error resolved\n");
        break;
case NE_REMOTE_OCCURRED:
        // error occurred on remote node (only node guarding)
        printf ("remote error occurred\n");
        break;
case NE_REMOTE_RESOLVED:
        // error resolved on remote node (only node guarding)
        printf ("remote error resolved\n");
        break;
default:
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
            // unknown error code, may not occur
            printf ("\t\t* unkown error type 0x%X\n", d[2]);
        }
        break;
```

An error indication message has the specifier 'MS_NMTS_EVENT_ind' in data byte 0 and 1. Data byte 2 contains a numeric value specifying whether the error comes from the local or from a remote node and whether the error has occurred or has resolved. If the error has occurred on the local node (our own CAN adapter device) a numeric value in data byte 3 specifies the reason. The following table gives some more information about the possible error cases.

Remote node errors only occur if the node guarding capabilities of the NMT slave service on the firmware is used. So, if a remote node error has occurred, it means, that a node guarding request (RTR message from NMT master node) was not received in the time defined as life time (node guarding time * life time factor). Maybe the NMT master node has a problem or the connection to this node has been cut in this case. For this error type also a 'resolved' indication exists. It is sent when the next node guarding request was received.

MACRO	Error reason	Description
EVT_ERROR	Error interrupt	An 'error interrupt' is the error-counter-passed-warning-level interrupt that the CAN controller generates on noisy networks. The 82C200 that is used on the VMOD-ICAN2, will also indicate such an event if the error counter falls below the warning level at a later time. When the module is equipped with an SJA1000 CAN controller, then an error interrupt will also be signalled when the controller enters the error-passive state (error counter reaches 128). This can be distinguished from the other error condition by looking at the error counter values that is appended in the error message (please see also chapter 4: CAN bus error handling). When the problem goes away, then you will receive a second error interrupt message.
EVT_OVERRUN	Overrun interrupt	The 'overrun interrupt' will be generated by the CAN controller when messages were not read fast enough by the Janz CAN adapter's local CPU. The firmware is designed that this should not occur at all (even at 1Mbit/s baud-rate).
EVT_LOST_INTERRUPTS	Interrupts lost	Lost interrupts are generated from the module's CAN bus interrupt service routine when an interrupt (whatever type) was served, but there were currently no resources in the message queue to that the interrupt handler will post such CAN bus interrupts. This could occur if the rate of incoming messages will be higher than the processing speed of the processor, and that for a longer time (so that the decoupling message queue gets exhausted). Any type of CAN bus interrupts could get lost (data or error), but the interrupt 'lost message' will not get lost.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

EVT_QUEUE_FULL	Send queue full	The 'send-queue-full' condition signals that the priority send queue, that decouples the send requests by the firmware from the CAN controller, has reached a maximum limit. This could occur if you send a lot of low prioritised messages, while the bus is full with high prioritised traffic. Another cause for this is, if you have a slow bit-rated network, and you try to send a huge amount of data without any confirmation.
EVT_BERR	Bus error	A 'bus-error' will be indicated each time that a fault condition is detected by the MAC layer of the CAN controller. Note that only the SJA1000 is capable of generating such messages. Note that this error could occur very frequently (i.e. if you loose connection to the bus by a cable fault). For this reason it is disabled at start-up of the module, and must be explicitly enabled (see chapter 4: CAN bus error handling').

Table 19: CANopen error cases

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7 Appendices

7.1 MODULbus

The MODULbus concept is a system independent flexible modular input/output solution for industrial applications. It consists of a carrier board for the specific host computer bus system (VMEbus, CompactPCI, PC-ISA, PC-PCI ...) and a range of more than 50 different I/O modules with MODULbus interface to the carrier board.

Using the MODULbus I/O-concept, the system integrator is able to develop computer systems with flexible configurations for application in industrial environments, small and low-cost solutions for e. g. test and measurement purpose can also easily be designed.

The Janz CAN MODULbus module (VMOD-ICAN3) is one of the available MODULbus modules:

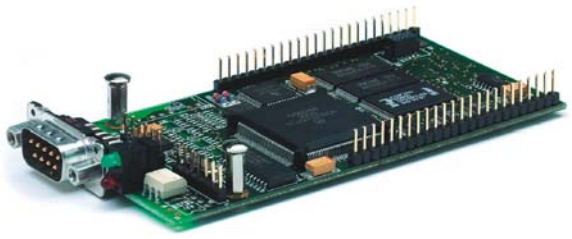


Figure 6: The Janz VMOD-ICAN3 MODULbus module

7.2 Philips SJA1000 filter modes

7.2.1 Single Filter mode

In this filter configuration one long filter (4-bytes) could be defined. The bit correspondences between the filter bytes and the message bytes depend on the currently received frame format:

7.2.1.1 Standard frame:

If a standard frame format message is received, the complete identifier including the RTR bit and the first two data bytes are used for acceptance filtering. Messages may also be accepted if there are no data bytes existing due to a set RTR bit or if there is none or only one data byte because of the corresponding data length code. For a successful reception of a message, all single bit comparisons have to signal acceptance.



Note, that the 4 least significant bits of the most significant identifier byte of the acceptance mask (AMR1) and the acceptance code (ACR1) are not used (see figure below). In order to be compatible with future products these bits should be programmed to be 'don't care' by setting the corresponding bits in the acceptance mask to logic 1.

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

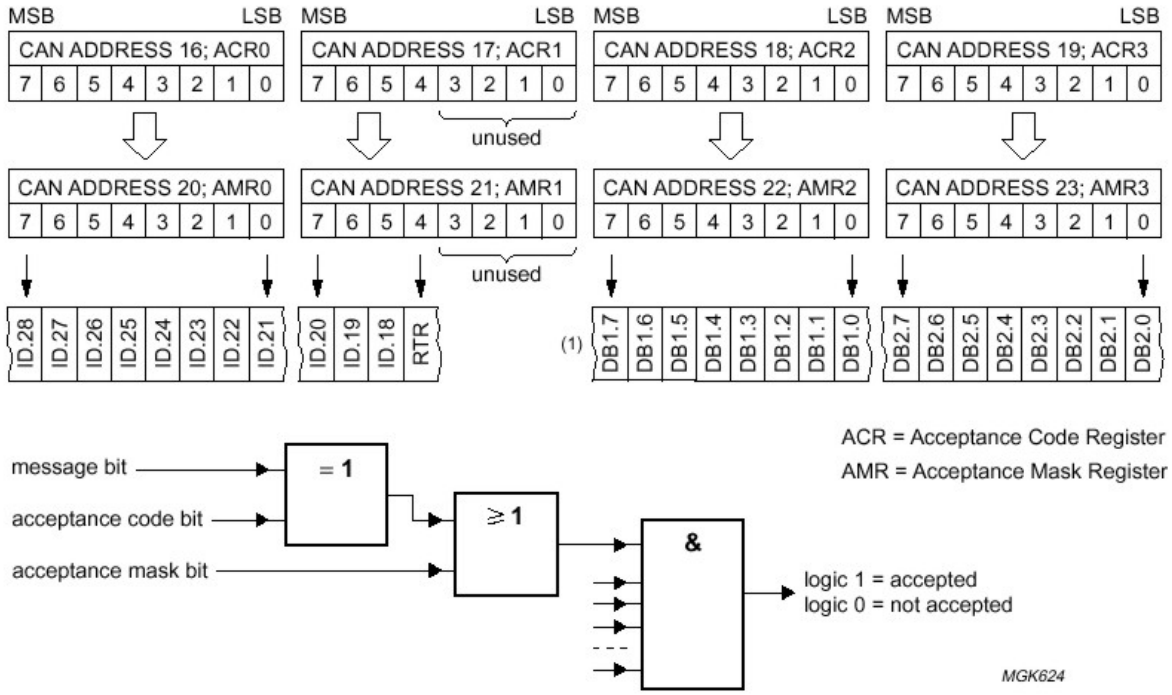


Figure 7: Single filter configuration, receiving standard CAN messages

7.2.1.2 Extended frame:

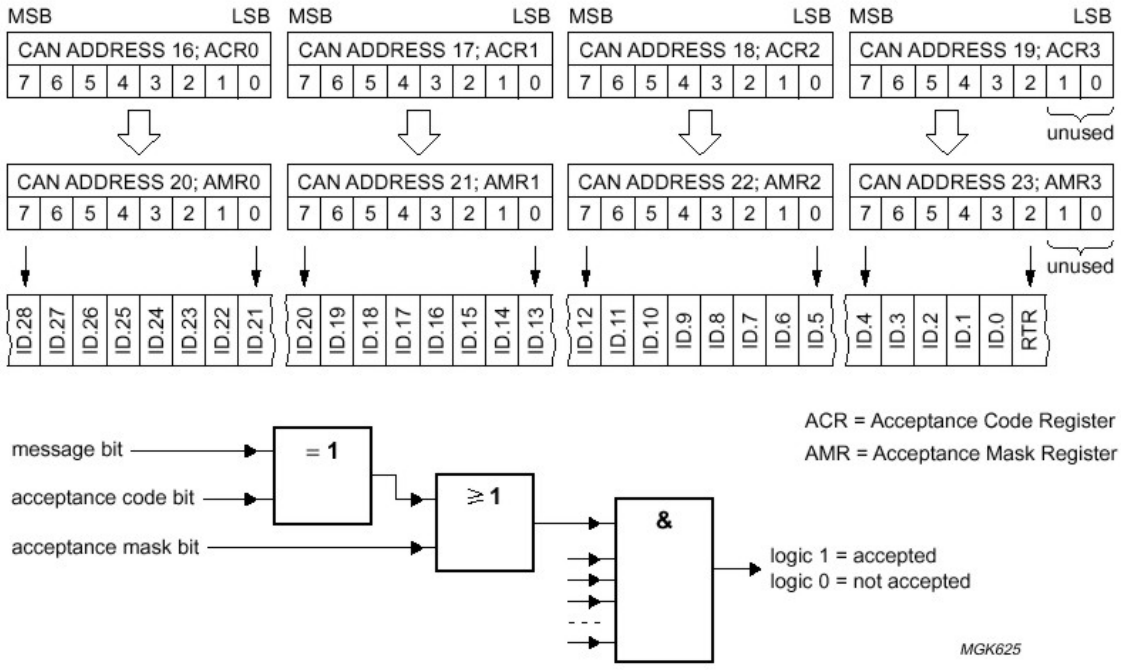


Figure 8: Single filter configuration, receiving extended CAN messages

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

If an extended frame format message is received, the complete identifier including the RTR bit is used for acceptance filtering.

For a successful reception of a message, all single bit comparisons have to signal acceptance. It should be noted that the 2 least significant bits of the most significant identifier byte of the acceptance mask (AMR3) and the acceptance code (ACR3) are not used (see figure 2 below). In order to be compatible with future products these bits should be programmed to be 'don't' care' by setting the corresponding bits in the acceptance mask to logic 1.

This chapter was copied out of the SJA1000 data sheet from Philips Semiconductor. For more detailed information please see this manual.

7.2.2 Dual filter mode

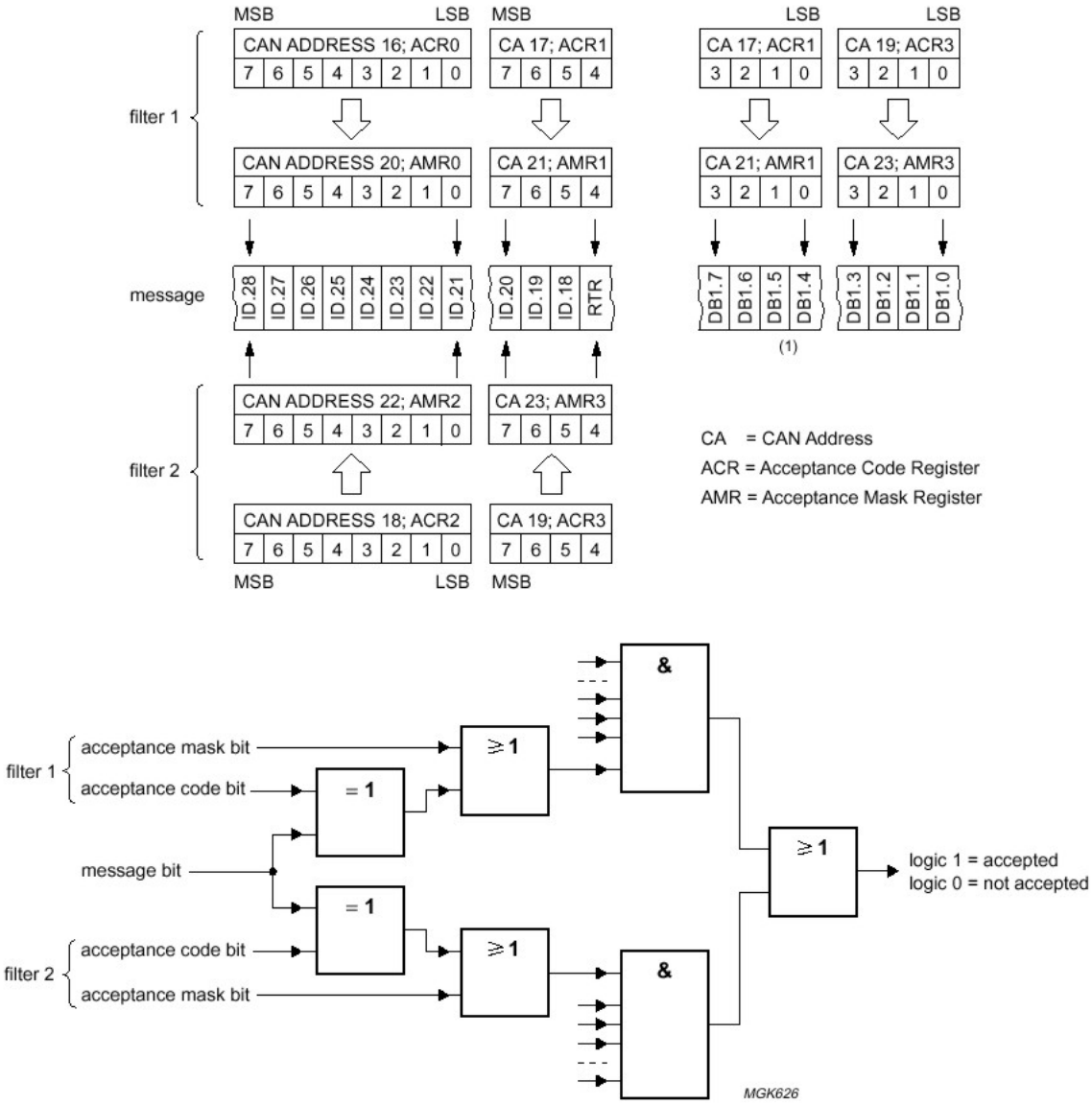


Figure 9: Dual filter configuration, receiving standard CAN messages

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

In this filter configuration two short filters can be defined. A received message is compared with both filters to decide, whether this message should be copied into the receive buffer or not. If at least one of the filters signals an acceptance, the received message becomes valid. The bit correspondence between the filter bytes and the message bytes depends on the currently received frame format.

7.2.2.1 Standard frame

If a standard frame message is received, the two defined filters are looking different! The first filter compares the complete standard identifier including the RTR bit and the first data byte of a message. The second filter just compares the complete standard identifier including the RTR bit.

For a successful reception of a message, all single bit comparisons of at least one complete filter have to signal acceptance. In case of a set RTR bit or a data length code of logic 0 no data byte is existing. Nevertheless a message may pass filter 1, if the first part up to the RTR bit signals acceptance.

If no data byte filtering is required for filter 1, the four least significant bits of the most significant acceptance code register bytes (AMR1 and AMR3) have to be set to logic 1 (don't care). Then both filters are working identically using the standard identifier range including the RTR bit.



Please note, that AMR3 as well as ACR3 is divided into to parts in this case. The first part of each register, the most significant four bits, are used for settings regarding the first data byte in filter 1. The least significant four bits are used for settings regarding the identifier byte in filter 2!

7.2.2.2 Extended frame

If an extended frame message is received, the two defined filters are looking identically. Both filters are comparing the first two bytes of the extended identifier range only.

For a successful reception of a message, all single bit comparisons of at least one complete filter have to indicate acceptance.



This chapter was copied out of the SJA1000 data sheet from Philips Semiconductor. For more detailed information please see this manual.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

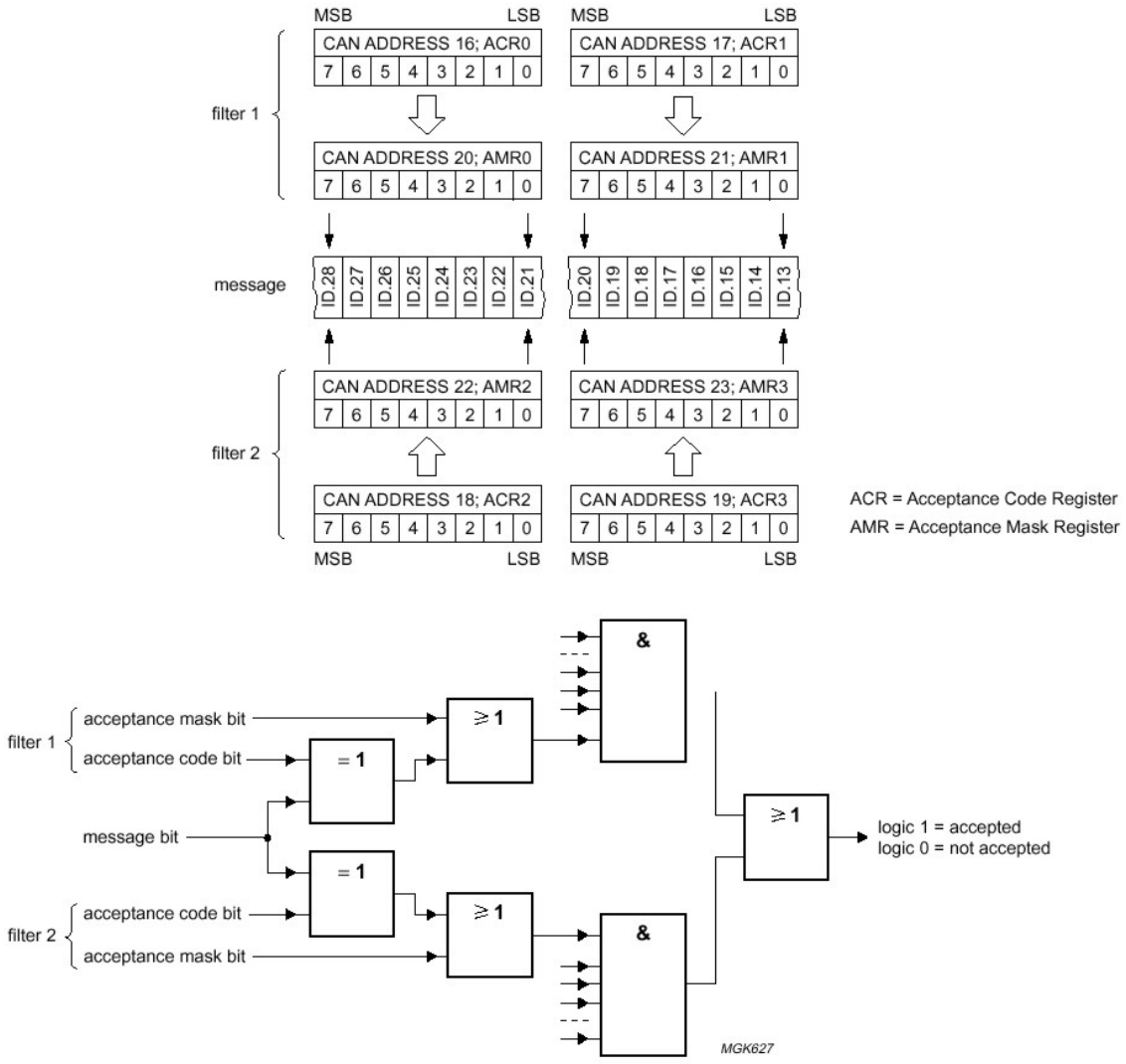


Figure 10: Dual filter configuration, receiving extended CAN messages

7.3 Firmware interfaces

The dual port memory (DPM) of Janz CAN devices provides several (independent) data transport mechanisms, as described below:

Standard host interface

This interface allows sending firmware requests to the module. The interpretation of these requests depends on whether you have ICANOS only or CAL/CANopen firmware installed. Messages that are send through this interface perform initialization, configuration and of course data transport. The standard host interface has two styles:

- Old style interface

  A very poor interface that has only limited functionality and is unreliable. For compatibility reasons, this is the only host interface that is active after a reset. It is used to configure the new style host interface.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

- New style interface

  This is the interface to use! Once enabled, it replaces the old style interface. Using function ‘can_device_setup’ for device initialization automatically enables this interface.

Fast host interface

The fast host interface provides a simple but direct way to access to CAN bus. There is very few software overhead, so fast data transport is possible.

Any configuration regarding the fast host interface has to be done through the standard host interface. Once configured, the fast host interface is used simultaneously to the standard host interface.

Memory mapped process picture (PP)

The process picture is used by the CAN/CANopen firmware only. Process data objects (PDOs) are stored memory mapped, and can be read by the host at any time and in any sequence. In contrast, the standard and the fast host interface serialize data transport through queues.

7.4 DPM queue sizes

The firmware installed on intelligent CAN devices queues firmware responses. We call these queues ‘DPM queues’ because they reside in the DPM memory of the device in fact!

The user is free to choose the size that each interface uses in the DPM. However some rules need to be obeyed:

- The DPM has a fixed size of 64KByte. It is organized in pages of 256bytes. Thus 256 pages are available.

- When you use CANopen process picture, the size of the process picture reduces the available memory. By default the process picture starts at address 0x8000 and extends to the end of the DPM. Thus available size for messages buffers is 128 pages (addresses 0x0000..0x8000).

- 9 Pages are used for general management purposes and further reduce the available message buffer size. E.g. with standard process picture 117 pages are left, without process pictures 247 pages are left.

- The remaining space can be freely partitioned for standard and fast host interface queues. You must check that the number of buffers in all of the queues does not exceed the available DPM space (as calculated above):

- Each buffer in any of the standard host interface queues consumes one DPM page. If you do not configure the three priority queues, the unconfigured two queues still consume one DPM page each. Each of the queues must not exceed 128 elements. Sometimes it is not possible to even reach this limit for a single queue, as the maximum 128 limit is independent from any total DPM size limits.

- Up to 16 buffers in a fast host interface queue fit to one DPM page. Remaining space in a page (if queue length is not a multiple of 16) is wasted. There is no special restriction for the length of fast host interface queues.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7.5 System requirements

This chapter will show you on which software, firmware and device driver versions this tutorial bases on. Normally the tutorial is also suitable for versions higher than the ones printed here.

7.5.1 Firmware

ICANOS firmware :

- Version 1.35 (ican3_1.35) or higher

CANopen firmware :

- Version 1.48 (v3cal_1.48) or higher

7.5.2 Device driver

Microsoft Windows 95/98

- not supported

Microsoft Windows NT :

- not longer in the scope of this document, please see Version 2.0!

Microsoft Windows 2000/XP:

- WDM Carrier device driver version 5.0 or higher
- WDM DPM device driver version 3.0 or higher (containing CANopen library version 2.2 or higher)
- Software development kit version 4.0 or higher

Linux :

- JHAL carrier device driver version 3.1 or higher
- DPM device driver version 3.2 or higher (containing CANopen library version 2.2 or higher)

Other operating systems :

- not supported

7.5.3 Janz library support

CANopen library :

- CANopen library version 2.2

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7.6 References

These references direct you to manuals and specifications that you might need to know when you attempt to program the Janz CAN devices. Most of the documents can be downloaded from the Internet.

[1] **MODULbus Specification Manual**, Fördergemeinschaft MODULbus e.V., Version 2.3, 1997.

[2] MODULbus+ Specification, Janz Computer AG, Version 1.0, 1999.

[2] ICAN3 Software Reference Manual, Janz Automationssysteme AG, Version 1.10, 2003.

7.7 Manual History

Version	Release Date	Name	Changes
V 1.0	24.03.2002	cm	• Initial version
V 2.0	15.02.2003	cm	• Completely revised, German version not longer supported
V 2.1	24.11.2003	cm	• Added a 'sleep' after BusOn command in fast host interface examples. • Added document history • System requirements updated
V 3.0	04.07.2005	cm	• Changed format from HTML to PDF, completely revised • Added hint for ICANOS daemon usage • Deleted chapter “2.5 Device driver installation” • Deleted description for conventional device initialization method • Conventional device initialization method not longer described. This method was declared as obsolete, customers should not use this method in new projects. Use the method instead which is known as convenient device initialization method from former versions of this document. This is the actual method now. • Changed title of chapter ‘ICANOS’ to ‘Standard host interface’ • Exchanged chapter 4 and chapter 5.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Version	Release Date	Name	Changes
V 3.1	20.10.2005	cm	• Added executables of all tutorial example programs for Windows operating systems
V 4.0	22.03.2006	cm	• Changed CANopen library from jic3cop.dll to jic3cop2.dll in all Windows examples • Added Visual Studion 2005 project files • Used environment variable JANZ_SDKINSTLL_DPM for include and library path settings in all Windows examples • Added tutorial to Janz DPM SDK for default

Table 20: Manual history

© **Janz Automationssysteme AG**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP