



Protocol API
CANopen Slave

V2.4.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC070901API06EN | Revision 6 | English | 2010-06 | Released | Public

Revision History

Rev	Date	Name	Revisions
0	04.09.07	ES	Created first draft
1	07.11.07	RG/ES/HH	First final version
2	04.03.08	RG/HH	Review Technical Data Firmware/stack version 2.0.4
3	02.06.08	RG/HH	Firmware/stack version V2.0.6 Reference to netX Dual-Port Memory Interface Manual Revision 5. Chapter <i>Configuration Parameters</i> restructured
4	26.11.08	RG/ES	Firmware/ stack version V2.2.2 Warmstart -> Set Configuration Registration/unregistration packet marked as obsolete. Changed some error numbers to global error numbers Slave state information Bus parameter request Reference to netX Dual-Port Memory Interface Manual Revision 7. Added section on task structure.
5	01.10.09	ES/RG	Firmware/stack version V2.2.4 PDO indication packets Extended section on task structure with descriptions of single tasks. Added information of suitability of packets for LFW or LOM approach.
6	10.06.10	RG	Firmware/stack version V2.4.x Section Technical Data: New: Support of DMA for PCI targets and support of slot number for CIFS 50-CO

Table of Contents

1	Introduction	8
1.1	Abstract	8
1.2	System Requirements	8
1.3	Intended Audience	8
1.4	Specifications	9
1.4.1	Technical Data	9
1.5	Terms, Abbreviations and Definitions	10
1.6	References	10
1.7	Legal Notes	11
1.7.1	Copyright	11
1.7.2	Important Notes	11
1.7.3	Exclusion of Liability	12
1.7.4	Export	12
2	Fundamentals	13
2.1	General Access Mechanisms on netX Systems	13
2.2	Accessing the Protocol Stack by Programming the AP Task's Queue	14
2.2.1	Getting the Receiver Task Handle of the Process Queue	14
2.2.2	Meaning of Source- and Destination-related Parameters	14
2.3	Accessing the Protocol Stack via the Dual Port Memory Interface	15
2.3.1	Communication via Mailboxes	15
2.3.2	Using Source and Destination Variables correctly	16
2.3.3	Obtaining useful Information about the Communication Channel	20
2.4	Client/Server Mechanism	22
2.4.1	Application as Client	22
2.4.2	Application as Server	23
3	Dual-Port-Memory	24
3.1	Cyclic Data (Input/Output Data)	24
3.1.1	Input Data Image	25
3.1.2	Process Data Output	25
3.2	Acyclic Data (Mailboxes)	26
3.2.1	General Structure of Messages or Packets for Non-Cyclic Data Exchange	27
3.2.2	Status & Error Codes	30
3.2.3	Differences between System and Channel Mailboxes	30
3.2.4	Send Mailbox	30
3.2.5	Receive Mailbox	30
3.2.6	Channel Mailboxes (Details of Send and Receive Mailboxes)	31
3.3	Status	32
3.3.1	Common Status	32
3.3.2	Extended Status	38
3.4	Control Block	42
4	Configuration Parameters	43
4.1	Overview about Essential Functionality	43
4.2	Configuration Procedures	43
4.2.1	Using a Packet (CANOPEN_APS_WARMSTART_REQ/CNF)	43
4.2.2	Behavior when receiving a Set Configuration / Warmstart Command	45
4.3	Task Structure of the CANopen Slave Stack	46
4.4	Handling of Process Data	48
4.4.1	General	48
4.4.2	Mapping of Input and Output Image to Send and Receive Objects	48
5	The Application Interface	49
5.1	The CANopen-APS-Task	49
5.1.1	CANOPEN_APS_GET_STATE_REQ/CNF – Get State of AP-Task	50
5.1.2	CANOPEN_APS_WARMSTART_REQ/CNF – Set Warmstart Parameters	52
5.1.3	CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration	58
5.2	The CANopen Slave-Task	63
5.2.1	CANOPEN_SLAVE_REGISTER_REQ/CNF – Register Application	64
5.2.2	CANOPEN_SLAVE_EXCHANGE_DATA_REQ/CNF – Exchange Data	68

5.2.3	CANOPEN_SLAVE_STARTSTOP_REQ/CNF – Start/Stop CANopen Network	72
5.2.4	CANOPEN_SLAVE_INITIALIZE_REQ/CNF – Initialization of CANopen Slave	76
5.2.5	CANOPEN_SLAVE_SET_BUSPARAM_REQ/CNF – Set Bus Parameters	79
5.2.6	CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ/CNF – Get Buffer Handle	85
5.2.7	CANOPEN_SLAVE_STATE_CHANGE_IND/RES – Change of State Indication	89
5.2.8	CANOPEN_SLAVE_SEND_EMCY_REQ/CNF – Send Emergency Message	93
5.2.9	CANOPEN_SLAVE_SET_NMT_STATE_REQ/CNF – Set NMT State	97
5.2.10	CANOPEN_SLAVE_SET_WATCHDOG_FAIL_REQ/CNF – Set Watchdog Fail	100
5.2.11	CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ/CNF – Setup PDO Indication	102
5.2.12	CANOPEN_SLAVE_RECEIVE_PDO_IND/RES – Receive PDO	105
6	Status/Error Codes Overview	108
6.1	Codes of the CANopen-APS-Task	108
6.1.1	Error Messages	108
6.2	Codes of the CANopen Slave-Task	109
6.2.1	Error Messages	109
7	Contact	112

List of Figures

Figure 1: The 3 different Ways to access a Protocol Stack running on a netX System.....	13
Figure 2: Use of <code>ulDest</code> in Channel and System Mailbox	16
Figure 3: Using <code>ulSrc</code> and <code>ulSrcId</code>	18
Figure 4: Transition Chart Application as Client.....	22
Figure 5: Transition Chart Application as Server.....	23
Figure 6: Internal Structure of CANopen Slave Firmware	46

List of Tables

Table 1: Terms, Abbreviations and Definitions.....	10
Table 2: References.....	10
Table 3: ASCII Queue Name	14
Table 4: Meaning of Source- and Destination-related Parameters	14
Table 5: Destination Queue Handle.....	17
Table 6: Using ulSrc and ulSrcId.....	19
Table 7: Input Data Image	25
Table 8: Output Data Image.....	25
Table 9: General Structure of Messages or Packets for Non-Cyclic Data Exchange.....	27
Table 10: Channel Mailboxes.....	31
Table 11: Common Status Structure Definition.....	33
Table 12: Communication State of Change	34
Table 13: Meaning of Communication Change of State Flags.....	35
Table 14: Extended Status Block (General Structure)	38
Table 15: Additional Info Block.....	39
Table 16: Additional Info Flags.....	40
Table 17: Communication Control Block.....	42
Table 18: Overview about essential Functionality (Cyclic and acyclic Data Transfer and Alarm Handling)	43
Table 19: Meaning and allowed Values for Warmstart-Parameters.....	44
Table 20: Mapping of Input Data.....	48
Table 21: Mapping of Output Data.....	48
Table 22: APM-Task Process Queue.....	49
Table 23: CANOPEN_APS_PCK_GET_STATE_REQ_T – Get State of AP-Task Request	50
Table 24: CANOPEN_APS_PCK_GET_STATE_CNF_T – Get State of AP-Task Confirmation	51
Table 25: CANOPEN_APS_PCK_WARMSTART_REQ – Set Warmstart Parameter Request.....	54
Table 26: CANOPEN_APS_PCK_WARMSTART_CNF_T – Set Warmstart Parameter Confirmation.....	57
Table 27: CANOPEN_APS_PCK_SET_CONFIGURATION_REQ – Set Warmstart Parameter Request.....	61
Table 28: CANOPEN_APS_PCK_SET_CONFIGURATION_CNF_T – Set Warmstart Parameter Confirmation.....	62
Table 29 CANopen Slave-Task Process Queue.....	63
Table 30: CANOPEN_SLAVE_PACKET_APP_REGISTER_REQ_T – Register Application Request	65
Table 31: CANOPEN_SLAVE_REGISTER_REQ – Packet Status/Error	65
Table 32: CANOPEN_SLAVE_PACKET_APP_REGISTER_CNF_T – Register Application Confirmation	66
Table 33: CANOPEN_SLAVE_REGISTER_CNF – Packet Status/Error	67
Table 34: CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_T – Exchange Data Request	69
Table 35: CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_T –Exchange Data Confirmation	71
Table 36: CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_T – Start/Stop Communication Request.....	73
Table 37: CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_T – Start/Stop Communication Confirmation.....	75
Table 38: CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_T – Initialization of CANopen Slave Request	77
Table 39: CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_T – Initialization of CANopen Slave Confirmation	78
Table 40: CANOPEN_SLAVE_CFG_BUS_PARAM_T - Bus Parameter Configuration	80
Table 41: Codes and Corresponding Baud Rates of CANopen Network.....	80
Table 42: CANOPEN_SLAVE_CFG_ADD_PARAM_T - Additional Configuration.....	81
Table 43: CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T – Set Bus Parameter Request	83
Table 44: CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_T –Set Bus Parameter Confirmation.....	84
Table 45: CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_REQ_T – Get Buffer Handle Request.....	86
Table 46: CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF – Get Buffer Handle Confirmation	88
Table 47: CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_T – Change of State Indication	90
Table 48: CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_T – Change of State Response	92
Table 49: CANOPEN_SLAVE_PACKET_SEND_EMICY_REQ_T – Send Emergency Message Request.....	94
Table 50: Error Register.....	95
Table 51: CANOPEN_SLAVE_PACKET_SEND_EMICY_CNF_T – Send Emergency Message Confirmation.....	96
Table 52: NMT States	97
Table 53: CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_REQ_T – Set NMT State Request	98
Table 54: CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_CNF_T – Set NMT State Confirmation	99
Table 55: CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_T – Set Watchdog Fail Request	100
Table 56: CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_T – Set Watchdog Fail Confirmation	101
Table 57: CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_T – Setup PDO Indication Request... ..	103
Table 58: CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_T – Setup PDO Indication Confirmation	104
Table 59: CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_T – Receive PDO Indication.....	106
Table 60: CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_T – Receive PDO Response.....	107
Table 61: Error Messages of the AP-Task	108

Table 62: Error Messages of the CANopen slave-Task..... 111

1 Introduction

1.1 Abstract

This manual describes the application interface of the CANopen Slave stack, with the aim to support and lead you during the integration process of the given stack into your own application.

Base of the development of the stack itself is the Hilscher's Task Layer Reference Programming Model. It is a description of how to program a Task in general, which is defined as a combination of appropriate functions belonging to the same type of protocol layer. It furthermore defines of how different Tasks have to communicate with each other in order to exchange their layer information in between. The reference model is commonly used by all programmers at Hilscher and shall be used by you as well when writing your Application Task on top of the stack.

1.2 System Requirements

The software package has the following system requirements to its environment:

- netX-Chip as CPU hardware platform
- Operating system for task scheduling required

1.3 Intended Audience

This manual is suitable for software developers with the following background:

- Knowledge of the programming language C
- Knowledge of the use of the real-time operating system rcX
- Knowledge of the Hilscher Task Layer Reference Model
- Knowledge of the CiA Work Draft 301 specification

1.4 Specifications

The data below applies to CANopen Slave firmware and stack version 2.4.x. The firmware/stack has been designed in order to meet the CiA Work Draft 301 V4.02 specification.

1.4.1 Technical Data

Maximum number of cyclic input data	512 bytes
Maximum number of cyclic output data	512 bytes
Maximum number of receive PDOs	64
Maximum number of transmit PDOs	64
Exchange of process data	via PDO transfer (synchronized, remotely requested, event driven (change of date))
Acyclic communication	SDO up- and download Emergency message (producer)
Functions	Node guarding / life guarding, heartbeat PDO mapping NMT Slave SYNC protocol (consumer)
Baud rates	10 kBits/s to 1 MBits/s
Data transport layer	CAN Frames
CAN Frame type	11 Bit

Firmware/stack available for netX

netX 50	yes
netX 100, netX 500	yes

PCI

DMA Support for PCI targets	yes
-----------------------------	-----

Slot Number

Slot number supported for	CIFX 50-CO
---------------------------	------------

Configuration

Configuration by packet to transfer warmstart parameters

Diagnostic

Firmware supports common and extended diagnostic in the dual-port-memory for loadable firmware

1.5 Terms, Abbreviations and Definitions

Term	Description
AP	Application on top of the Stack
Boot up	Initial sequence of node during start-up
EMCY	Emergency
Guarding	Supervision of node
NMT	Network Management
PDO	Process Data Object (process data channel)
PDO-Mapping	Configurable process data per PDO
SDO	Service Data Object (representing an acyclic data channel)
SYNC	Synchronization cycle of the CANopen slave

Table 1: Terms, Abbreviations and Definitions

All variables, parameters and data used in this manual have basically the LSB/MSB (“Intel”) data representation. This corresponds to the convention of the Microsoft C Compiler.

1.6 References

This document is based on the following specifications:

1	EN 50325/4 Specification
2	CiA Work Draft 301
3	netX Dual-Port Memory Interface Manual

Table 2: References

1.7 Legal Notes

1.7.1 Copyright

© 2008-2010 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

The images, photographs and texts in the accompanying material (user manual, accompanying texts, documentation, etc.) are protected by German and international copyright law as well as international trade and protection provisions. You are not authorized to duplicate these in whole or in part using technical or mechanical methods (printing, photocopying or other methods), to manipulate or transfer using electronic systems without prior written consent. You are not permitted to make changes to copyright notices, markings, trademarks or ownership declarations. The included diagrams do not take the patent situation into account. The company names and product descriptions included in this document may be trademarks or brands of the respective owners and may be trademarked or patented. Any form of further use requires the explicit consent of the respective rights owner.

1.7.2 Important Notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.7.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.7.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different countries, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Fundamentals

2.1 General Access Mechanisms on netX Systems

This chapter explains the possible ways to access a Protocol Stack running on a netX system :

1. By accessing the Dual Port Memory Interface directly or via a driver.
2. By accessing the Dual Port Memory Interface via a shared memory.
3. By interfacing with the Stack Task of the Protocol Stack.

The picture below visualizes these three ways:

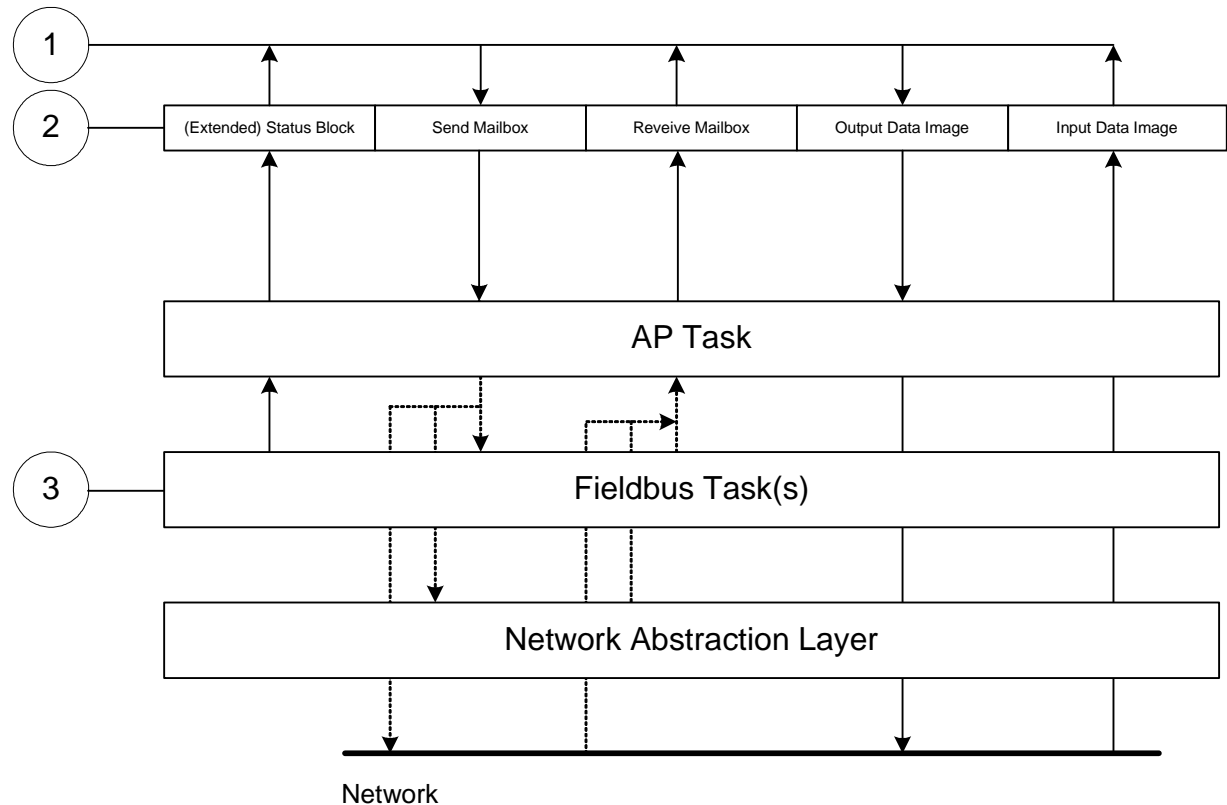


Figure 1: The 3 different Ways to access a Protocol Stack running on a netX System

This chapter explains how to program the stack (alternative 3) correctly while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the virtual DPM). Finally, chapter 5 titled “*The Application Interface*” describes the entire interface to the protocol stack in detail.

Depending on you choose the stack-oriented approach or the Dual Port Memory-based approach, you will need either the information given in this chapter or those of the next chapter to be able to work with the set of functions described in chapter 5. All of those functions use the four parameters `ulDest`, `ulSrc`, `ulDestId` and `ulSrcId`. This chapter and the next one inform about how to work with these important parameters.

2.2 Accessing the Protocol Stack by Programming the AP Task's Queue

In general, programming the AP task or the stack has to be performed according to the rules explained in the Hilscher Task Layer Reference Manual. There you can also find more information about the variables discussed in the following.

2.2.1 Getting the Receiver Task Handle of the Process Queue

To get the handle of the process queue of the CANopen Slave-Task the Macro `TLR_QUE_IDENTIFY()` needs to be used. It is described in detail within section 10.1.9.3 of the Hilscher Task Layer Reference Model Manual. This macro delivers a pointer to the handle of the intended queue to be accessed (which is returned within the third parameter, `phQue`), if you provide it with the name of the queue (and an instance of your own task). The correct ASCII-queue name for accessing the CANopen Slave-Task which you have to use as current value for the first parameter (`pszIdn`) is

ASCII queue name	Description
"QUE_CANOPENSLV"	Name of the CANopen slave-Task process queue
"QUE_CANOPENAPS"	Name of the APS-Task process queue

Table 3: ASCII Queue Name

The returned handle has to be used as value `ulDest` in all initiator packets the AP-Task intends to send to the CANopen slave. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDFIFO/LIFO()` for sending a packet to the respective task.

2.2.2 Meaning of Source- and Destination-related Parameters

The meaning of the source- and destination-related parameters is explained in the following table:

Variable	Meaning
<code>ulDest</code>	Application mailbox used for confirmation
<code>ulSrc</code>	Queue handle returned by <code>TLR_QUE_IDENTIFY()</code> as described above.
<code>ulSrcId</code>	Used for addressing at a lower level

Table 4: Meaning of Source- and Destination-related Parameters

For more information about programming the AP task's stack queue, please refer to the Hilscher Task Layer Reference Model Manual. Especially the following sections might be of interest in this context:

1. Section 7 "Queue-Packets"
2. Section 10.1.9 "Queuing Mechanism"

2.3 Accessing the Protocol Stack via the Dual Port Memory Interface

This chapter defines the application interface of the CANopen slave stack.

2.3.1 Communication via Mailboxes

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer to and from the netX.

- **Send Mailbox**
Packet transfer from host system to firmware
- **Receive Mailbox**
Packet transfer from firmware to host system

For more details about acyclic data transfer via mailboxes see section 3.2. The concept of using messages called packets in this context, is described in detail in section 3.2.1 “General Structure of Messages or Packets for Non-Cyclic Data Exchange” while the possible codes that may appear are listed in section 3.2.2. “Status & Error Codes”.

However, this section concentrates on correct addressing the mailboxes.

2.3.2 Using Source and Destination Variables correctly

2.3.2.1 How to use `ulDest` for Addressing `rcX` and the `netX` Protocol Stack by the System and Channel Mailbox

The preferred way to address the `netX` operating system `rcX` is through the system mailbox; the preferred way to address a protocol stack is through its channel mailbox. All mailboxes, however, have a mechanism to route packets to a communication channel or the system channel, respectively. Therefore, the destination identifier `ulDest` in a packet header has to be filled in according to the targeted receiver. See the following example.

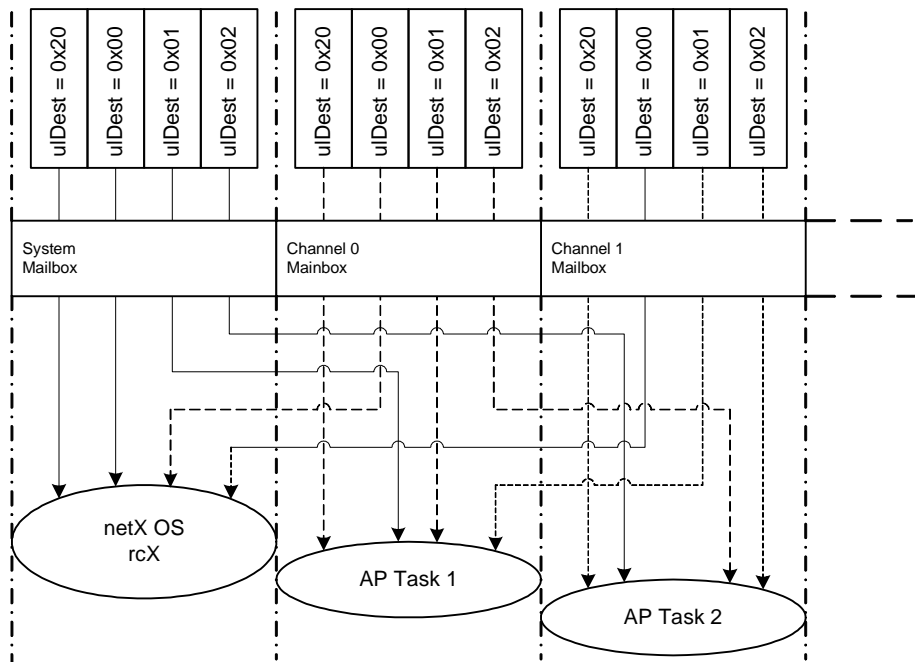


Figure 2: Use of `ulDest` in Channel and System Mailbox

For use in the destination queue handle, the tasks have been assigned to hexadecimal numerical values as described in the following table:

ulDest	Description
0x00000000	Packet is passed to the netX operating system rcX
0x00000001	Packet is passed to communication channel 0
0x00000002	Packet is passed to communication channel 1
0x00000003	Packet is passed to communication channel 2
0x00000004	Packet is passed to communication channel 3
0x00000020	Packet is passed to communication channel of the mailbox
else	Reserved, do not use

Table 5: Destination Queue Handle

The picture and the table above both show the use of the destination identifier *ulDest*.

A remark on the special channel identifier 0x00000020 (= *Channel Token*). The Channel Token is valid for any mailbox. That way the application uses the same identifier for all packets without actually knowing which mailbox or communication channel is applied. The packet stays 'local'. The system mailbox is a little bit different, because it is used to communicate to the netX operating system rcX. The rcX has its own range of valid commands codes and differs from a communication channel.

Unless there is a reply packet, the netX operating system returns it to the same mailbox the request packet went through. Consequently, the host application has to return its reply packet to the mailbox the request was received from.

2.3.2.2 How to use `ulSrc` and `ulSrcId`

Generally, a netX protocol stack can be addressed through its communication channel mailbox. The example below shows how a host application addresses a protocol stack running in the context of a netX chip. The application is identified by a number (#444 in this example). The application consists of three processes identified by the numbers #11, #22 and #33. These processes communicate through the channel mailbox with the AP task of the protocol stack. Have a look at the following image:

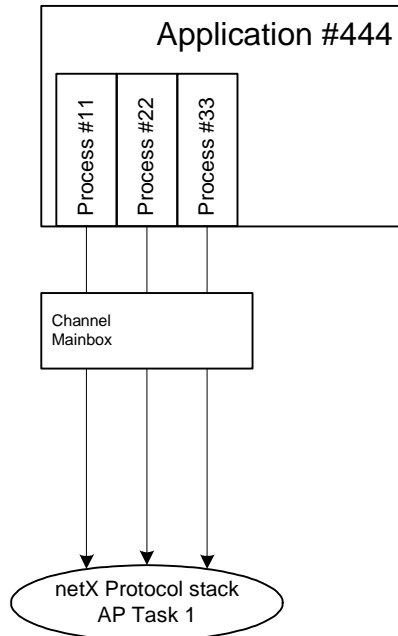


Figure 3: Using `ulSrc` and `ulSrcId`

Example:

This example applies to command messages initiated by a process in the context of the host application. If the process #22 sends a packet through the channel mailbox to the AP task, the packet header has to be filled in as follows:

Object	Variable Name	Numeric Value	Explanation
Destination Queue Handle	ulDest	= 32 (0x00000020)	This value needs always to be set to 0x00000020 (the channel token) when accessing the protocol stack via the local communication channel mailbox.
Source Queue Handle	ulSrc	= 444	Denotes the host application (#444).
Destination Identifier	ulDestId	= 0	In this example it is not necessary to use the destination identifier.
Source Identifier	ulSrcId	= 22	Denotes the process number of the process within the host application and needs therefore to be supplied by the programmer of the host application.

Table 6: Using ulSrc and ulSrcId

For packets through the channel mailbox, the application uses 32 (= 0x20, *Channel Token*) for the destination queue handler *ulDest*. The source queue handler *ulSrc* and the source identifier *ulSrcId* are used to identify the originator of a packet. The destination identifier *ulDestId* can be used to address certain resources in the protocol stack. It is not used in this example. The source queue handler *ulSrc* has to be filled in. Therefore its use is mandatory; the use of *ulSrcId* is optional.

The netX operating system passes the request packet to the protocol stack's AP task. The protocol stack then builds a reply to the packet and returns it to the mailbox. The application has to make sure that the packet finds its way back to the originator (process #22 in the example).

2.3.2.3 How to Route rcX Packets

To route an rcX packet the source identifier *ulSrcId* and the source queues handler *ulSrc* in the packet header hold the identification of the originating process. The router saves the original handle from *ulSrcId* and *ulSrc*. The router uses a handle of its own choices for *ulSrcId* and *ulSrc* before it sends the packet to the receiving process. That way the router can identify the corresponding reply packet and matches the handle from that packet with the one stored earlier. Now the router replaces its handles with the original handles and returns the packet to the originating process.

2.3.3 Obtaining useful Information about the Communication Channel

A communication channel represents a part of the Dual Port Memory and usually consists of the following elements:

- **Output Data Image**
is used to transfer cyclic process data to the network (normal or high-priority)
- **Input Data Image**
is used to transfer cyclic process data from the network (normal or high-priority)
- **Send Mailbox**
is used to transfer non-cyclic data to the netX
- **Receive Mailbox**
is used to transfer non-cyclic data from the netX
- **Control Block**
allows the host system to control certain channel functions
- **Common Status Block**
holds information common to all protocol stacks
- **Extended Status Block**
holds protocol specific network status information

This section describes a procedure how to obtain useful information for accessing the communication channel(s) of your netX device and to check if it is ready for correct operation.

Proceed as follows:

- 1) Start with reading the channel information block within the system channel (usually starting at address 0x0030).
- 2) Then you should check the hardware assembly options of your netX device. They are located within the system information block following offset 0x0010 and stored as data type UINT16. The following table explains the relationship between the offsets and the corresponding xC Ports of the netX device:

Offset	Port
0x0010	Hardware Assembly Options for xC Port[0]
0x0012	Hardware Assembly Options for xC Port[1]
0x0014	Hardware Assembly Options for xC Port[2]
0x0016	Hardware Assembly Options for xC Port[3]

Check each of the hardware assembly options whether its value has been set to `RCX_HW_ASSEMBLY_CAN = 0x0030`. If true, this denotes that this xCPort is suitable for running the CANopen protocol stack. Otherwise, this port is designed for another communication protocol. In most cases, xC Port[2] will be used for field-bus systems, while xC Port[0] and xC Port[1] are normally used for Ethernet communication.

- 3) You can find information about the corresponding communication channel (0...3) under the following addresses:

Offset	Communication Channel
0x0050	Communication Channel 0
0x0060	Communication Channel 1
0x0070	Communication Channel 2
0x0080	Communication Channel 3

In devices which support only one communication system which is usually the case (either a single field-bus system or a single standard for Industrial-Ethernet communication), always communication channel 0 will be used. In devices supporting more than one communication system you should also check the other communication channels.

- 4) There you can find such information as the ID (containing channel number and port number) of the communication channel, the size and the location of the handshake cells, the overall number of blocks within the communication channel and the size of the channel in bytes. Evaluate this information precisely in order to access the communication channel correctly.

The information is delivered as follows:

Size of Channel in Bytes

Address	Data Type	Description
0x0050	UINT8	Channel Type = COMMUNICATION (must have the fixed value define RCX_CHANNEL_TYPE_COMMUNICATION = 0x05)
0x0051	UINT8	ID (Channel Number, Port Number)
0x0052	UINT8	Size / Position Of Handshake Cells
0x0053	UINT8	Total Number Of Blocks Of This Channel
0x0054	UINT32	Size Of Channel In Bytes
0x0058	UINT8[8]	Reserved (set to zero)

These addresses correspond to communication channel 0, for communication channels 1, 2 and 3 you have to add an offset of 0x0010, 0x0020 or 0x0030 to the address values, respectively.

- 5) Finally, you can access the communication channel using the addresses you determined previously. For more information how to do this, please refer to the netX DPM Manual, especially section 3.2 "Communication Channel".

2.4 Client/Server Mechanism

2.4.1 Application as Client

The host application may send request packets to the netX firmware at any time (transition 1 ⇒ 2). Depending on the protocol stack running on the netX, parallel packets are not permitted (see protocol specific manual for details). The netX firmware sends a confirmation packet in return, signalling success or failure (transition 3 ⇒ 4) while processing the request.

The host application has to register with the netX firmware in order to receive indication packets (transition 5 ⇒ 6). Depending on the protocol stack, this is done either implicit (if application opens a TCP/UDP socket) or explicit (if application wants to receive unsolicited packets). Details on when and how to register for certain events is described in the protocol specific manual. Depending on the command code of the indication packet, a response packet to the netX firmware may or may not be required (transition 7 ⇒ 8).

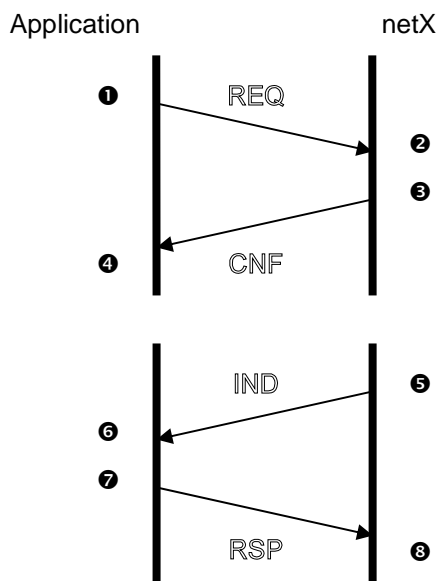


Figure 4: Transition Chart Application as Client

➊ ➋ The host application sends request packets to the netX firmware.

➌ ➍ The netX firmware sends a confirmation packet in return.

➎ ➏ The host application receives indication packets from the netX firmware.

➐ ➑ The host application sends response packet to the netX firmware (may not be required).

REQ	Request	CNF	Confirmation
IND	Indication	RSP	Response

2.4.2 Application as Server

The host application has to register with the netX firmware in order to receive indication packets. Depending on the protocol stack, this is done either implicit (if application opens a TCP/UDP socket) or explicit (if application wants to receive unsolicited packets). Details on when and how to register for certain events is described in the protocol specific manual.

When an appropriate event occurs and the host application is registered to receive such a notification, the netX firmware passes an indication packet through the mailbox (transition 1 ⇒ 2). The host application is expected to send a response packet back to the netX firmware (transition 3 ⇒ 4).

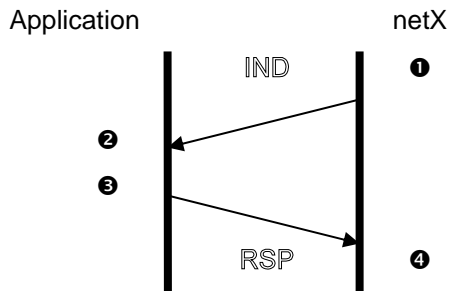


Figure 5: Transition Chart Application as Server

- ① ② The netX firmware passes an indication packet through the mailbox.
- ③ ④ The host application sends response packet to the netX firmware.

IND Indication RSP Response

3 Dual-Port-Memory

All data in the dual-port memory is structured in blocks. According to their functions, these blocks use different data transfer mechanisms. For example, data transfer through mailboxes uses a synchronized handshake mechanism between host system and netX firmware. The same is true for IO data images, when a buffered handshake mode is configured. Other blocks, like the status block, are read by the host application and use no synchronization mechanism.

Types of blocks in the dual-port memory are outlined below:

- **Mailbox**
transfer non-cyclic messages or packages with a header for routing information
- **Data Area**
holds the process image for cyclic IO data or user defined data structures
- **Control Block**
is used to signal application related state to the netX firmware
- **Status Block**
holds information regarding the current network state
- **Change of State**
collection of flags, that initiate execution of certain commands or signal a change of state

3.1 Cyclic Data (Input/Output Data)

The input block holds the process data image received **from** the network whereas the output block holds data sent **to** the network.

Process data transfer through the data blocks can be synchronized by using a handshake mechanism (configurable). If in uncontrolled mode, the protocol stack updates the process data in the input and output data image in the dual-port memory for each valid bus cycle. No handshake bits are evaluated and no buffers are used. The application can read or write process data at any given time without obeying the synchronization mechanism otherwise carried out via handshake location. This transfer mechanism is the simplest method of transferring process data between the protocol stack and the application. This mode can only guarantee data consistency over a byte.

For the controlled / buffered mode, the protocol stack updates the process data in the internal input buffer for each valid bus cycle. Each IO block uses handshake bits for access synchronization. Input and output data block handshake operates independently from each other. When the application toggles the input handshake bit, the protocol stack copies the data from the internal buffer into the input data image of the dual-port memory. Now the application can copy data from the dual-port memory and then give control back to the protocol stack by toggling the appropriate input handshake bit. When the application/driver toggles the output handshake bit, the protocol stack copies the data from the output data image of the dual-port memory into the internal buffer. From there the data is transferred to the network. The protocol stack toggles the handshake bits back, indicating to the application that the transfer is finished and a new data exchange cycle may start. This mode guarantees data consistency over both input and output area.

3.1.1 Input Data Image

The input data block is used by field bus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The input data image is used to transfer cyclic data **from** the network.

The default size of the input data image is 5760 byte. An output and input data block may or may not be available in the dual-port memory. They are always available in the default memory map (see the netX Dual-Port Memory Manual).

Input Data Image			
Offset	Type	Name	Description
0x2680	UINT8	abPd0Input [5760]	Input Data Image Cyclic Data From The Network

Table 7: Input Data Image

3.1.2 Process Data Output

The output data block is used by field bus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The output data Image is used to transfer cyclic data **to** the network.

The default size of the output data image is 5760 byte. An output data block may or may not be available in the dual-port memory. They are always available in the default memory map (see netX DPM Manual).

Output Data Image			
Offset	Type	Name	Description
0x1000	UINT8	abPd0Output [5760]	Output Data Image Cyclic Data To The Network

Table 8: Output Data Image

3.2 Acyclic Data (Mailboxes)

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer.

- **Send Mailbox**
Packet transfer from host system to firmware
- **Receive Mailbox**
Packet transfer from firmware to host system

The send and receive mailbox areas are used by field bus protocols providing a non-cyclic data exchange mechanism. Another use of the mailbox system is to allow access to the firmware running on the netX chip itself for diagnostic and identification purposes. The send mailbox is used to transfer cyclic data **to** the network or **to** the firmware. The receive mailbox is used to transfer cyclic data **from** the network or **from** the firmware.

A send/receive mailbox may or may not be available in the communication channel. It depends on the function of the firmware whether or not a mailbox is needed. The location of the system mailbox and the channel mailbox is described in the netX DPM Interface Manual.

Note: Each mailbox can hold one packet at a time. The netX firmware stores packets that are not retrieved by the host application in a packet queue. This queue has limited space and may fill up so new packets maybe lost. To avoid these deadlock situations, it is strongly recommended to empty the mailbox frequently, even if packets are not expected by the host application. Unexpected command packets should be returned to the sender with an *Unknown Command* in the status field; unexpected reply messages can be discarded.

3.2.1 General Structure of Messages or Packets for Non-Cyclic Data Exchange

The non-cyclic packets through the netX mailbox have the following structure:

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32		Destination Queue Handle
	ulSrc	UINT32		Source Queue Handle
	ulDestId	UINT32		Destination Queue Reference
	ulSrcId	UINT32		Source Queue Reference
	ulLen	UINT32		Packet Data Length (In Bytes)
	ulId	UINT32		Packet Identification As Unique Number
	ulSta	UINT32		Status / Error Code
	ulCmd	UINT32		Command / Response
	ulExt	UINT32		Reserved
	ulRout	UINT32		Routing Information
tData	Structure Information			
		User Data Specific To The Command

Table 9: General Structure of Messages or Packets for Non-Cyclic Data Exchange

Some of the fields are mandatory; some are conditional; others are optional. However, the size of a packet is always at least 10 double-words or 40 bytes. Depending on the command, a packet may or may not have a data field. If present, the content of the data field is specific to the command, respectively the reply.

Destination Queue Handler

The *ulDest* field identifies a task queue in the context of the netX firmware. The task queue represents the final receiver of the packet and is assigned to a protocol stack. The *ulDest* field has to be filled out in any case. Otherwise, the netX operating system cannot route the packet. This field is mandatory.

Source Queue Handler

The *ulSrc* field identifies the sender of the packet. In the context of the netX firmware (inter-task communication) this field holds the identifier of the sending task. Usually, a driver uses this field for its

own handle, but it can hold any handle of the sending process. Using this field is mandatory. The receiving task does not evaluate this field and passes it back unchanged to the originator of the packet.

Destination Identifier

The *ulDestId* field identifies the destination of an unsolicited packet from the netX firmware to the host system. It can hold any handle that helps to identify the receiver. Therefore, its use is mandatory for unsolicited packets. The receiver of unsolicited packets has to register for this.

Source Identifier

The *ulSrcId* field identifies the originator of a packet. This field is used by a host application, which passes a packet from an external process to an internal netX task. The *ulSrcId* field holds the handle of the external process. When netX operating system returns the packet, the application can identify the packet and returns it to the originating process. The receiving task on the netX does not evaluate this field and passes it back unchanged. For inter-task communication, this field is not used.

Length of Data Field

The *ulLen* field holds the size of the data field in bytes. It defines the total size of the packet's payload that follows the packet's header. The size of the header is not included in *ulLen*. So the total size of a packet is the size from *ulLen* plus the size of packet's header. Depending on the command, a data field may or may not be present in a packet. If no data field is included, the length field is set to zero.

Identifier

The *ulId* field is used to identify a specific packet among others of the same kind. That way the application or driver can match a specific reply or confirmation packet to a previous request packet. The receiving task does not change this field and passes it back to the originator of the packet. Its use is optional in most of the cases. But it is mandatory for sequenced packets. Example: Downloading big amounts of data that does not fit into a single packet. For a sequence of packets the identifier field is incremented by one for every new packet.

Status / Error Code

The *ulState* field is used in response or confirmation packets. It informs the originator of the packet about success or failure of the execution of the command. The field may be also used to hold status information in a request packet.

Command / Response

The *ulCmd* field holds the command code or the response code, respectively. The command/response is specific to the receiving task. If a task is not able to execute certain commands, it will return the packet with an error indication. A command is always even (the least significant bit is zero). In the response packet, the command code is incremented by one indicating a confirmation to the request packet.

Extension

The extension field *ulExt* is used for controlling packets that are sent in a sequenced manner. The extension field indicates the first, last or a packet of a sequence. If sequencing is not required, the extension field is not used and set to zero.

Routing Information

The *ulRout* field is used internally by the netX firmware only. It has no meaning to a driver type application and therefore set to zero.

User Data Field

This field contains data related to the command specified in *ulCmd* field. Depending on the command, a packet may or may not have a data field. The length of the data field is given in the *ulLen* field.

3.2.2 Status & Error Codes

The following status and error codes can be returned in `ulState`: List of codes see manual named *netX Dual-Port Memory Interface*.

3.2.3 Differences between System and Channel Mailboxes

The mailbox system on netX provides a non-cyclic data transfer channel for field bus and industrial Ethernet protocols. Another use of the mailbox is allowing access to the firmware running on the netX chip itself for diagnostic purposes. There is always a send and a receive mailbox. Send and receive mailboxes utilize handshake bits to synchronize these data or diagnostic packages through the mailbox. There is a pair of handshake bits for both the send and receive mailbox.

The netX operating system rcX only uses the system mailbox.

- The *system mailbox*, however, has a mechanism to route packets to a communication channel.
- A *channel mailbox* passes packets to its own protocol stack only.

3.2.4 Send Mailbox

The send mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **send** mailbox is used to transfer non-cyclic data **to** the network or **to** the protocol stack.

The size is 1596 bytes for the send mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of packages that can be accepted.

3.2.5 Receive Mailbox

The receive mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **receive** mailbox is used to transfer non-cyclic data **from** the network or **from** the protocol stack.

The size is 1596 bytes for the receive mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of waiting packages (for the receive mailbox).

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

Master Status			
Offset	Type	Name	Description
0x0200	UINT16	usPackagesAccepted	Packages Accepted Number of Packages that can be Accepted
0x0202	UINT16	usReserved	Reserved Set to 0
0x0204	UINT8	abSendMbx[1596]	Send Mailbox Non Cyclic Data To The Network or to the Protocol Stack
0x0840	UINT16	usWaitingPackages	Packages waiting Counter of packages that are waiting to be processed
0x0842	UINT16	usReserved	Reserved Set to 0
0x0844	UINT8	abRecvMbx[1596]	Receive Mailbox Non Cyclic Data from the network or from the protocol stack

Table 10: Channel Mailboxes

Channel Mailboxes Structure

```
typedef struct tagNETX_SEND_MAILBOX_BLOCK
{
  UINT16 usPackagesAccepted;
  UINT16 usReserved;
  UINT8 abSendMbx[1596];
} NETX_SEND_MAILBOX_BLOCK;
typedef struct tagNETX_RECV_MAILBOX_BLOCK
{
  UINT16 usWaitingPackages;
  UINT16 usReserved;
  UINT8 abRecvMbx[1596];
}NETX_RECV_MAILBOX_BLOCK;
```

3.3 Status

A status block is present within the communication channel. It contains information about network and task related issues. In some respects, status and control block are used together in order to exchange information between host application and netX firmware. The application reads a status block whereas the control block is written by the application. Both status and control block have registers that use the *Change of State* mechanism (see also section 2.2.1 of the *netX Dual-Port-Memory manual*).

3.3.1 Common Status

The Common Status Block contains information that is the same for all communication channels. The start offset of this block depends on the size and location of the preceding blocks. The status block is always present in the dual-port memory.

3.3.1.1 All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Structure Definition

Common Status			
Offset	Type	Name	Description
0x0010	UINT32	ulCommunicationCOS	<u>Communication Change of State</u> READY, RUN, RESET REQUIRED, NEW, CONFIG AVAILABLE, CONFIG LOCKED
0x0014	UINT32	ulCommunicationState	<u>Communication State</u> NOT CONFIGURED, STOP, IDLE, OPERATE
0x0018	UINT32	ulCommunicationError	<u>Communication Error</u> Unique Error Number According to Protocol Stack
0x001C	UINT16	usVersion	<u>Version</u> Version Number of this Diagnosis Structure
0x001E	UINT16	usWatchdogTime	<u>Watchdog Timeout</u> Configured Watchdog Time
0x0020	UINT16	usHandshakeMode	Handshake Mode Process Data Transfer Mode (see netX DPM Interface Manual)
0x0022	UINT16	usReserved	Reserved Set to 0
0x0024	UINT32	ulHostWatchdog	<u>Host Watchdog</u> Joint Supervision

			Mechanism Protocol Stack Writes, Host System Reads
0x0028	UINT32	ulErrorCount	<u>Error Count</u> Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	ulErrorLogInd	<u>Error Log Indicator</u> Total Number Of Entries In The Error Log Structure (not supported yet)
0x0030	UINT32	ulReserved[2]	<u>Reserved</u> Set to 0

Table 11: Common Status Structure Definition

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
    UINT32    ulCommunicationCOS;
    UINT32    ulCommunicationState;
    UINT32    ulCommunicationError;
    UINT16    usVersion;
    UINT16    usWatchdogTime;
    UINT16    ausReserved[2];
    UINT32    ulHostWatchdog;
    UINT32    ulErrorCount;
    UINT32    ulErrorLogInd;
    UINT32    ulReserved[2];
    union
    {
        {
            NETX_MASTER_STATUS_T    tMasterStatus;    /* for master implementation */
            UINT32                    aulReserved[6];    /* otherwise reserved */
        } unStackDepended;
    }
} NETX_COMMON_STATUS_BLOCK_T;
```

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
    UUINT32    ulCommunicationCOS;
    UUINT32    ulCommunicationState;
    UUINT32    ulCommunicationError;
    UUINT16    usVersion;
    UUINT16    usWatchdogTime;
    UUINT16    ausReserved[2];
    UUINT32    ulHostWatchdog;
    UUINT32    ulErrorCount;
    UUINT32    ulErrorLogInd;
    UUINT32    ulReserved[2];
    union
    {
        {
            NETX_MASTER_STATUS_T    tMasterStatus;    /* for master implementation */
            UUINT32                aulReserved[6];    /* otherwise reserved */
        }
    } unStackDepended;
} NETX_COMMON_STATUS_BLOCK_T;
```

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the *netX Change of State Command* flag in the netX communication flags register (see section 3.2.2.1 of the netX DPM Interface Manual). The application then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state (see section 3.2.2.2 of the netX DPM Interface Manual).

ulCommunicationCOS - netX writes, Host reads		
Bit	Short name	Name
D31..D7	unused, set to zero	
D6	Restart Required Enable	RCX_COMM_COS_RESTART_REQUIRED_ENABLE
D5	Restart Required	RCX_COMM_COS_RESTART_REQUIRED
D4	Configuration New	RCX_COMM_COS_CONFIG_NEW
D3	Configuration Locked	RCX_COMM_COS_CONFIG_LOCKED
D2	Bus On	RCX_COMM_COS_BUS_ON
D1	Running	RCX_COMM_COS_RUN
D0	Ready	RCX_COMM_COS_READY

Table 12: Communication State of Change

Communication Change of State Flags (netX System ⇒ Application)

Bit	Definition / Description
0	Ready (RCX_COMM_COS_READY) 0 - ... 1 - The <i>Ready</i> flag is set as soon as the protocol stack is started properly. Then the protocol stack is awaiting a configuration. As soon as the protocol stack is configured properly, the <i>Running</i> flag is set, too.
1	Running (RCX_COMM_COS_RUN) 0 - ... 1 -The <i>Running</i> flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the <i>Ready</i> flag and the <i>Running</i> flag are set.
2	Bus On (RCX_COMM_COS_BUS_ON) 0 - ... 1 -The <i>Bus On</i> flag is set to indicate to the host system whether or not the protocol stack has the permission to open network connections. If set, the protocol stack has the permission to communicate on the network; if cleared, the permission was denied and the protocol stack will not open network connections.
3	Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - ... 1 -The <i>Configuration Locked</i> flag is set, if the communication channel firmware has locked the configuration database against being overwritten. Re-initializing the channel is not allowed in this state. To unlock the database, the application has to clear the <i>Lock Configuration</i> flag in the control block (see section 3.2.4 of the netX DPM Interface Manual).
4	Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - ... 1 -The <i>Configuration New</i> flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated. This flag may be set together with the <i>Restart Required</i> flag.
5	Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - ... 1 -The <i>Restart Required</i> flag is set when the channel firmware requests to be restarted. This flag is used together with the <i>Restart Required Enable</i> flag below. Restarting the channel firmware may become necessary, if a new configuration was downloaded from the host application or if a configuration upload via the network took place.
6	Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - ... 1 - The <i>Restart Required Enable</i> flag is used together with the <i>Restart Required</i> flag above. If set, this flag enables the execution of the <i>Restart Required</i> command in the netX firmware (for details on the <i>Enable</i> mechanism see section 2.3.2 of the netX DPM Interface Manual)).
7 ... 31	Reserved, set to 0

Table 13: Meaning of Communication Change of State Flags

Other values are reserved.

Communication State (All Implementations)

The communication state field contains information regarding the current network status of the communication channel. Depending on the implementation, all or a subset of the definitions below is supported.

■ UNKNOWN	#define RCX_COMM_STATE_UNKNOWN	0x00000000
■ NOT_CONFIGURED	#define RCX_COMM_STATE_NOT_CONFIGURED	0x00000001
■ STOP	#define RCX_COMM_STATE_STOP	0x00000002
■ IDLE	#define RCX_COMM_STATE_IDLE	0x00000003
■ OPERATE	#define RCX_COMM_STATE_OPERATE	0x00000004

Communication Channel Error (All Implementations)

This field holds the current error code of the communication channel. If the cause of error is resolved, the communication error field is set to zero (= RCX_SYS_SUCCESS) again. Not all of the error codes are supported in every implementation. Protocol stacks may use a subset of the error codes below.

■ SUCCESS	#define RCX_SYS_SUCCESS	0x00000000
-----------	-------------------------	------------

Runtime Failures

■ WATCHDOG TIMEOUT	#define RCX_E_WATCHDOG_TIMEOUT	0xC000000C
--------------------	--------------------------------	------------

Initialization Failures

■ (General) INITIALIZATION FAULT	#define RCX_E_INIT_FAULT	0xC0000100
■ DATABASE ACCESS FAILED	#define RCX_E_DATABASE_ACCESS_FAILED	0xC0000101

Configuration Failures

■ NOT CONFIGURED	#define RCX_E_NOT_CONFIGURED	0xC0000119
■ (General) CONFIGURATION FAULT	#define RCX_E_CONFIGURATION_FAULT	0xC0000120
■ INCONSISTENT DATA SET	#define RCX_E_INCONSISTENT_DATA_SET	0xC0000121
■ DATA SET MISMATCH	#define RCX_E_DATA_SET_MISMATCH	0xC0000122
■ INSUFFICIENT LICENSE	#define RCX_E_INSUFFICIENT_LICENSE	0xC0000123
■ PARAMETER ERROR	#define RCX_E_PARAMETER_ERROR	0xC0000124
■ INVALID NETWORK ADDRESS	#define RCX_E_INVALID_NETWORK_ADDRESS	0xC0000125
■ NO SECURITY MEMORY	#define RCX_E_NO_SECURITY_MEMORY	0xC0000126

Network Failures

■ (General) NETWORK FAULT	#define RCX_COMM_NETWORK_FAULT	0xC0000140
■ CONNECTION CLOSED	#define RCX_COMM_CONNECTION_CLOSED	0xC0000141
■ CONNECTION TIMED OUT	#define RCX_COMM_CONNECTION_TIMEOUT	0xC0000142
■ LONELY NETWORK	#define RCX_COMM_LONELY_NETWORK	0xC0000143
■ DUPLICATE NODE	#define RCX_COMM_DUPLICATE_NODE	0xC0000144
■ CABLE DISCONNECT	#define RCX_COMM_CABLE_DISCONNECT	0xC0000145

Version (All Implementations)

The version field holds version of this structure. It starts with one; zero is not defined.

■ STRUCTURE VERSION	#define RCX_STATUS_BLOCK_VERSION	0x0001
---------------------	----------------------------------	--------

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see section 4.13 of the netX DPM Interface Manual.

Host Watchdog (All Implementations)

The protocol stack supervises the host system using the watchdog function. If the application fails to copy the value from the device watchdog location (section 3.2.5 of the netX DPM Interface Manual) to the host watchdog location (section 3.2.4 of the netX DPM Interface Manual), the protocol stack assumes that the host system has some sort of problem and shuts down all network connections. For details on the watchdog function, refer to section 4.13 of the netX DPM Interface Manual.

Error Count (All Implementations)

This field holds the total number of errors detected since power-up, respectively after reset. The protocol stack counts all sorts of errors in this field no matter if they were network related or caused internally.

Error Log Indicator (All Implementations)

The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero. The error log indicator is not supported yet.

3.3.1.2 Master Implementation

The master contains additional structures for the administration of all slaves which are connected to the master. These are not discussed here as they are not relevant for the slave.

3.3.1.3 Slave Implementation

The slave firmware uses only the common structure as outlined in section 3.2.5.1 of the Hilscher netX Dual-Port-Memory Manual.

3.3.2 Extended Status

The content of the channel specific extended status block is specific to the implementation. Depending on the protocol, a status area may or may not be present in the dual-port memory. It is always available in the default memory map (see section 3.2.1 of netX Dual-Port Memory Manual).

Note: Have in mind, that all offsets mentioned in this section are relative to the beginning of the common status block, as the start offset of this block depends on the size and location of the preceding blocks.

Extended Status Block			
Offset	Type	Name	Description
0x0050	UINT8[]	abExtendedStatus[432]	Extended Status Area Protocol Stack Specific Status Area

Table 14: Extended Status Block (General Structure)

Extended Status Block Structure

```
typedef struct CANOPEN_SLAVE_EXTENDED_STATE_Ttag
    CANOPEN_SLAVE_EXTENDED_STATE_T;

#define CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_INIT          0x00000001L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_ACTIVE       0x00000002L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_PASSIVE          0x00000004L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_BUS_OFF          0x00000008L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_RX_OVERFLOW      0x00000010L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_TX_OVERFLOW      0x00000020L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_WDG              0x00000100L
#define CANOPEN_SLAVE_EXT_STATE_CTRL                  0x00001000L
#define CANOPEN_SLAVE_EXT_STATE_NRDY                  0x00002000L
#define CANOPEN_SLAVE_EXT_STATE_TIMEOUT               0x00004000L

#define CANOPEN_SLAVE_EXT_STATE_UNKNOWN                0x00000000L
#define CANOPEN_SLAVE_EXT_STATE_OPERATIONAL           0x00000001L
#define CANOPEN_SLAVE_EXT_STATE_STOP                  0x00000002L
#define CANOPEN_SLAVE_EXT_STATE_PRE_OPERATIONAL        0x00000080L
#define CANOPEN_SLAVE_EXT_STATE_INITIALISING           0x000000FFL

#define CANOPEN_SLAVE_ADD_DETAIL_SIZE                  0x00000003L

struct CANOPEN_SLAVE_EXTENDED_STATE_Ttag
{
    TLR_UINT32 ulFlags;
    TLR_UINT32 ulNodeState;
    TLR_UINT32 ulBusOffEveCnt;
    TLR_UINT32 ulErrorPassiveEveCnt;
    TLR_UINT32 ulRxOverflowCnt;
    TLR_UINT32 ulTxOverflowCnt;
    TLR_UINT32 ulReserved;
    TLR_UINT32 ulTimeoutCnt;

    TLR_UINT32 aulReserved[4];

    TLR_UINT32 ulDiagInfoCount;
    TLR_UINT32 ulLastDiagInfo;
    TLR_UINT32 ulMaxRecvIdx;
    TLR_UINT32 ulMaxSendIdx;
    TLR_UINT32 aulAddDetail[CANOPEN_SLAVE_ADD_DETAIL_SIZE];
};
```

3.3.2.1 Extended Status Block

Additional Info Block for CANopen Slave			
Offset	Type	Name	Description
0x50	unsigned long	ulFlags	Bit field for Flags
0x54	unsigned long	ulNodeState	Current Node State
0x58	unsigned long	ulBusOffEveCnt	Counter for bus off events
0x5C	unsigned long	ulErrorPassiveEveCnt	Counter for passive event errors
0x60	unsigned long	ulRxOverflowCnt	Counter for receive overflows
0x64	unsigned long	ulTxOverflowCnt	Counter for transmit overflows
0x68	unsigned long	ulReserved	Reserved for further use
0x6C	unsigned long	ulTimeoutCnt	Number of timeouts
0x70	unsigned long[]	aulReserved[4]	Reserved for further use
0x80	unsigned long	ulDiagInfoCount	Number of diagnostic entries
0x84	unsigned long	ulLastDiagInfo	Last diagnostic entry
0x88	unsigned long	ulMaxRecvIdx	Maximum Object Index Value for Receive Data
0x8C	unsigned long	ulMaxSendIdx	Maximum Object Index Value for Send Data
0x90	unsigned long[]	aulAddDetail[3]	Additional detail for diagnostic entry

Table 15: Additional Info Block

ulFlags

This variable is organized as a bit field as described in the table below:

Bit	Name	Description
D31.. D15	Reserved	Reserved for further use
D14	CANOPEN_SLAVE_EXT_STATE_TIMEOUT	The DEVICE has detected an overstepped timeout supervision time of at least one CAN message to be sent. The transmission of this message was aborted. The data is lost. Its an indication that no other CAN device was connected or responsive at this time to acknowledge the sent message requests. The bit will be set when the first timeout was detected and will not be deleted any more.
D13	CANOPEN_SLAVE_EXT_STATE_NRDY	Indicates if the HOST program has set its state to operative or not. If the bit is set the HOST program is not ready to communicate.
D12	CANOPEN_SLAVE_EXT_STATE_CTRL	Parameterization error or severe run time error
D11.. D9	Reserved	Reserved for further use
D8	CANOPEN_SLAVE_EXT_STATE_FLAG_WDG	Watchdog error detected
D7.. D6	Reserved	Reserved for further use
D5	CANOPEN_SLAVE_EXT_STATE_FLAG_TX_OVERFLOW	Transmit overflow detected
D4	CANOPEN_SLAVE_EXT_STATE_FLAG_RX_OVERFLOW	Receive overflow detected
D3	CANOPEN_SLAVE_EXT_STATE_FLAG_BUS_OFF	CAN is in Bus-off state
D2	CANOPEN_SLAVE_EXT_STATE_FLAG_PASSIVE	CAN is in error passive state
D1	CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_ACTIVE	CAN is activated
D0	CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_INIT	CAN is initialized

Table 16: Additional Info Flags

ulNodeState

Internal node state of node:

0: Unknown state

1: Operational state

2: Stop

128: Pre-operational state

255: Initializing

ulBusOffEveCnt

Number of Bus-off events

ulErrorPassiveEveCnt

Number of error passive events

ulRxOverflowCnt

Number of receive overrun events

ulTxOverflowCnt

Number of transmit overrun events

ulReserved

Reserved for further use

ulTimeoutCnt

Each CAN message is supervised by the card to be sent during 20ms by the CAN chip. If not possible, because the chip for example gets no acknowledging partner on the bus, the counter is incremented by one.

aulReserved[]

Reserved for further use

ulDiagInfoCount

Number of diagnostic entries

ulLastDiagInfo

Last diagnostic entry

ulMaxRecvIdx

Number of highest PDO mapped receive object index

ulMaxSendIdx

Number of highest PDO mapped send object index

aulAddDetail[]

Additional detail for diagnostic entry

3.4 Control Block

A control block is always present in both system and communication channel. In some respects, control and status block are used together in order to exchange information between host application and netX firmware. The control block is written by the application, whereas the application reads a status block. Both control and status block have registers that use the *Change of State* mechanism (see section 0).

The following gives an example of the use of control and status block. The host application wishes to lock the configuration settings of a communication channel to protect them against changes. The application sets the *Lock Configuration* flag in the control block to the communication channel firmware. As a result, the channel firmware sets the *Configuration Locked* flag in the status block (see below), indicating that the current configuration settings cannot be deleted, altered, overwritten or otherwise changed.

The control block of a dual-port memory features a watchdog function to allow the operating system running on the netX supervise the host application and vice versa. The control area is always present in the dual-port memory.

Control Block			
Offset	Type	Name	Description
0x0008	UINT32	ulApplicationCOS	Application Change Of State State Of The Application Program INITIALIZATION, LOCK CONFIGURATION
0x000C	UINT32	ulDeviceWatchdog	Device Watchdog Host System Writes, Protocol Stack Reads

Table 17: Communication Control Block

Communication Control Block Structure

```
typedef struct NETX_CONTROL_BLOCK_Ttag
{
  UINT32 ulApplicationCOS;
  UINT32 ulDeviceWatchdog;
} NETX_CONTROL_BLOCK_T;
```

For more information concerning the control block please refer to the netX DPM Interface Manual.

4 Configuration Parameters

4.1 Overview about Essential Functionality

You can find the most commonly used functionality of the CANopen slave API within the following sections of this document:

Topic	Section Number	Section Name
Process (PDO) data transfer (Input/Output)	5.2.1	CANOPEN_SLAVE_EXCHANGE_DATA_REQ/CNF – Exchange Data
Emergency Handling	5.2.8	CANOPEN_SLAVE_SEND_EMCY_REQ/CNF – Send Emergency Message

Table 18: Overview about essential Functionality (Cyclic and acyclic Data Transfer and Alarm Handling).

4.2 Configuration Procedures

The following ways are available to configure the CANopen Slave:

- By warmstart packet
- By netX configuration and diagnostic utility

4.2.1 Using a Packet (CANOPEN_APS_WARMSTART_REQ/CNF)

The following table contains relevant information about the warmstart parameters for the CANopen Slave firmware such as an explanation of the meaning of the parameter and ranges of allowed values:

Parameter	Meaning	Range of Value / Value
Bus Startup	<p>The start of the device can be performed either application controlled or automatically:</p> <ul style="list-style-type: none"> ■ Automatic: <p>Network connections are opened automatically without taking care of the state of the host application.</p> ■ Application controlled: <p>The channel firmware is forced to wait for the host application to set the Application Ready flag in the communication change of state register (see section 3.2.5.1 of the <i>netX DPM Interface Manual</i>).</p> <p>For more information concerning this topic see section 4.4.1 “Controlled or Automatic Start” of the <i>netX DPM Interface Manual</i>.</p>	<p>Application controlled (1), Automatic (0) Default: Automatic (0)</p>
I/O Status (not yet supported)	<p>This parameter is represented by bits 1 and 2 of the system flags.</p> <p>Using this parameter you can set the status of the input or the output data. For each input and output date the following status information (in byte) is stored in the dual-</p>	

	<p>port memory.</p> <p>The bits have the following meaning:</p> <p>Bit 1 (I/O Status Enable):</p> <p>0 = Status disabled</p> <p>1 = Status enabled (not yet supported)</p> <p>Bit 2 (I/O Status 8/32Bit):</p> <p>0 = 1 Byte mode (not yet supported)</p> <p>1 = 4 Byte mode (not yet supported)</p>	
Watchdog Time [ms]	<p>Watchdog time within which the device watchdog must be retriggered from the application program while the application program monitoring is activated. When the watchdog time value is equal to 0 respectively the application program monitoring is deactivated.</p>	<p>[0, 20 ... 65535] ms, 0 = Off</p>
Node ID	<p>Node ID of CANopen slave</p>	<p>Allowed values: 1 .. 127</p>
Baudrate	<p>Baud rate of CANopen connection. See below!</p>	<p>Allowed values: all baud rates offered in <i>Table 41: Codes and Corresponding Baud Rates of CANopen Network</i>.</p>
Flags	<p>CANopen Flags, see explanation below!</p>	<p>BIT 0: 29-BIT IDENTIFIER DISABLED/ENABLED</p> <p>Not supported yet</p> <p>BIT 1 - 31: Reserved for further use, set to zero</p>

Table 19: Meaning and allowed Values for Warmstart-Parameters.

4.2.2 Behavior when receiving a Set Configuration / Warmstart Command

The following rules apply for the behaviour of the CANopen Slave protocol stack when receiving a set configuration command:

- The configuration packets name is `CANOPEN_APS_SET_CONFIGURATION_REQ` for the request and `CANOPEN_APS_SET_CONFIGURATION_CNF` for the confirmation.
- The configuration data are checked for consistency and integrity.
- In case of failure no data are accepted.
- In case of success the configuration parameters are stored internally (within the RAM).
- The parameterized data will be activated only after a channel init has been performed.
- No automatic registration of the application at the stack happens.
- The confirmation packet `CANOPEN_APS_SET_CONFIGURATION_CNF` only transfers simple status information, but does not repeat the whole parameter set.

For all versions up to firmware version V2.1.2.0, only the warmstart command (the predecessor of the set configuration command) was present showing up the following deviations from the behavior described above:

1. Contrary to the situation when receiving a set configuration command, on every received warmstart packet an automatic channel-init is performed.
2. The entire parameter set is completely delivered within the `CANOPEN_APS_SET_CONFIGURATION_CNF` or `CANOPEN_APS_WARMSTART_CNF` packet.

4.3 Task Structure of the CANopen Slave Stack

The illustration below displays the internal structure of the tasks which together represent the CANopen Slave Stack:

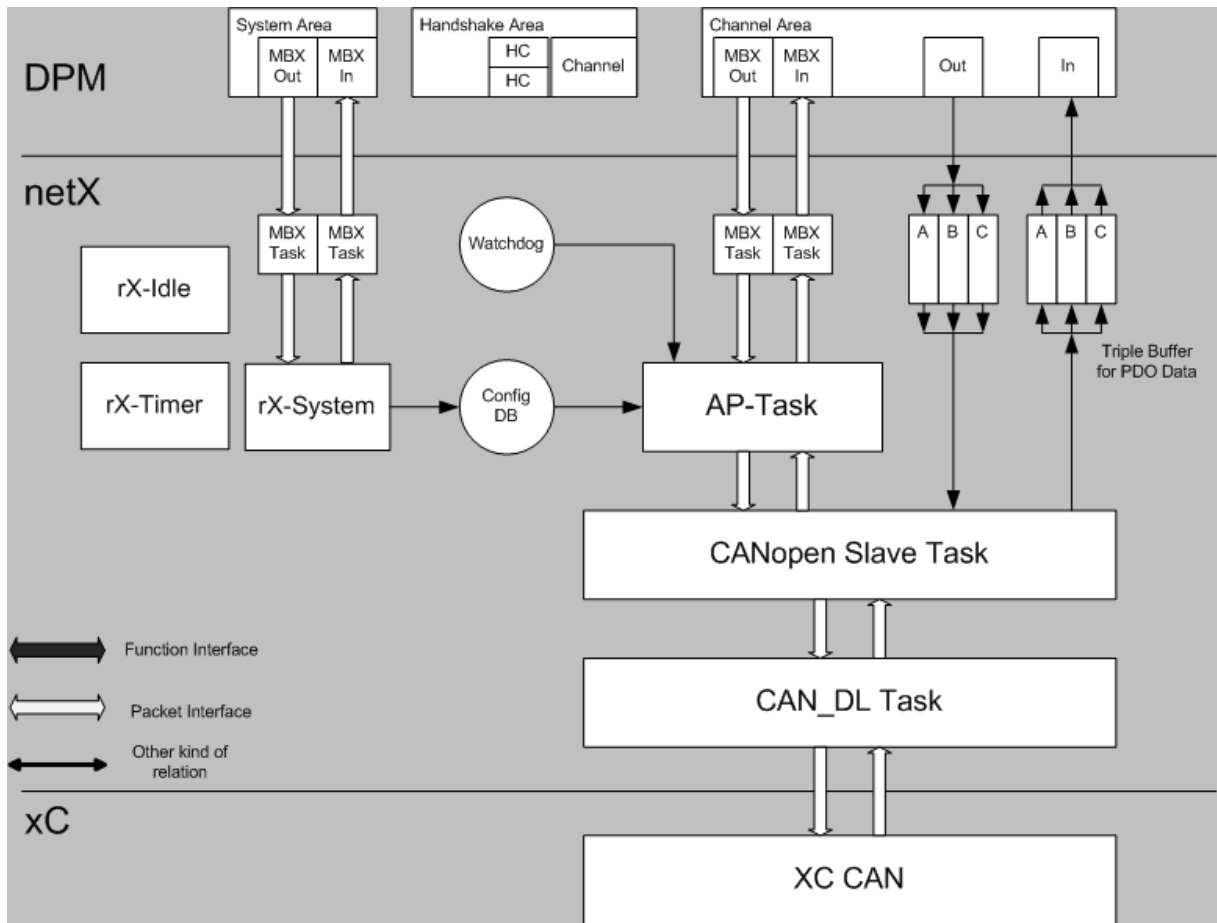


Figure 6: Internal Structure of CANopen Slave Firmware

The dual-port memory is used for exchange of information, data and packets. Configuration and IO data will be transferred using this way.

The user application only accesses the task located in the highest layer namely the AP task which constitutes the application interface of the CANopen Slave stack.

The triple buffer mechanism provides a consistent synchronous access procedure from both sides (DPM and AP task). The triple buffer technique ensures that the access will always affect the last written cell.

In detail, the various tasks have the following functionality and responsibilities:

AP-Task

The AP-Task provides the interface to the user application and the control of the stack. It also completely handles the DualPort Memory interface of the communication channel. In detail, it is responsible for the following:

- - Handling the communication channels DPM-interface
- - Configuration of the protocol stack
- - IO Process data exchange
- - Channel mailboxes
- - Watchdog supervision
- - Handling of applications packets
- - Send/Receive packets

CANopen Slave Task

The CANopen Slave Task is the CANopen Slave stack implementation. It is responsible for the protocol handling, the communication to/from CAN_DL layer and it is the counterpart of the AP-Task.

CAN_DL Task

The CAN_DL Task handles the interface of the XC CAN and is responsible for configuration, events and sending and receiving of CAN-Frames.

4.4 Handling of Process Data

4.4.1 General

The CANopen slave implementation provides 4 objects for send data and 4 objects of receive data; each object has up to 128 bytes of process data and is transferred via PDO according to the active network configuration. For accessing these objects, the CANopen slave-Task provides 4 triple buffers for both directions. Each triple buffer contains the process data for one object.

The data of these buffers are exchanged between the AP-Task and the CANopen slave-Task via triple buffer exchange. The AP-Task transfers data from the receive triple buffers to the DPM input image and from the DPM output image to the send triple buffers.

These objects can also be accessed by the user application via the packet interface as described in section CANOPEN_SLAVE_REGISTER_REQ/CNF – Register Application of this document.

4.4.2 Mapping of Input and Output Image to Send and Receive Objects

The data of the send and receive objects are mapped linear to the input and output image of the DPM as shown in the following table:

DPM input image byte offset	Receive object index	Receive object sub-index
0	2200h	01h
1	2200h	02h
..
127	2200h	80h
128	2201h	01h
..
510	2203h	7Fh
511	2203h	80h

Table 20: Mapping of Input Data

DPM output image byte offset	Send object index	Send object sub-index
0	2000h	01h
1	2000h	02h
..
127	2000h	80h
128	2001h	01h
..
510	2003h	7Fh
511	2003h	80h

Table 21: Mapping of Output Data

5 The Application Interface

This chapter defines the application user interface of the CANopen slave stack.

The application itself has to be developed as a Task according to the Hilscher's Task Layer Reference Model. The Application-Task is named AP-Task in the following sections and chapters.

The AP-Task's process queue is keeping track of all its incoming packets and has to be addressed from the user application. It provides the communication channel for the underlying CANopen slave stack. Once, the CANopen slave stack communication is established, events received by the stack are mapped to packets that are sent to the AP-Task's process queue. On one hand every packet has to be evaluated in the AP-Task's context and corresponding actions be executed. On the other hand, Initiator-Services that are be requested by the AP-Task itself are sent via predefined queue macros to the underlying CANopen slave stack queues via packets as well. The AP-Task will not route all commands to the CANopen slave task. Some commands are used for internal communication between the AP- and CANopen slave-Task only. Other requests are not possible in specific states.

5.1 The CANopen-APS-Task

To get the handle of the process queue of the CANopen-APS-task the Macro `TLR_QUE_IDENTIFY()` needs to be used.

ASCII queue name	Description
"QUE_CANOPENAPS"	Name of the APS-Task process queue

Table 22: APM-Task Process Queue

The returned handle has to be used as value `ulDest` in all request packets to be sent to the AP-Task. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDFILE_PACKET_FIFO/LIFO()` for sending a packet to the AP-Task.

5.1.1 CANOPEN_APS_GET_STATE_REQ/CNF – Get State of AP-Task

This request can be used by the user application to get status information from the AP-Task.

Packet Structure Reference

```
typedef struct CANOPEN_APS_PCK_GET_STATE_REQ_Ttag
    CANOPEN_APS_PCK_GET_STATE_REQ_T;

struct CANOPEN_APS_PCK_GET_STATE_REQ_Ttag /* Get state request */
{
    TLR_PACKET_HEADER_T tHead; /** packet header */
};
```

Packet Description

Structure Information CANOPEN_APS_PCK_GET_STATE_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN APS	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.1 Codes of the CANopen-APS-Task
	ulCmd	UINT32	0x00002E02	CANOPEN_APS_GET_STATE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 23: CANOPEN_APS_PCK_GET_STATE_REQ_T – Get State of AP-Task Request

Packet Structure Reference

```
typedef struct CANOPEN_APS_GET_STATE_CNF_DATA_Ttag
    CANOPEN_APS_GET_STATE_CNF_DATA_T;

struct CANOPEN_APM_GET_STATE_CNF_DATA_Ttag /* Get state confirmation data */
{
    TLR_UINT32 ulHighestMappedSendBufferNum;
    TLR_UINT32 ulHighestMappedRecvBufferNum;
}

typedef struct CANOPEN_APS_PCK_GET_STATE_CNF_Ttag
    CANOPEN_APS_PCK_GET_STATE_CNF_T;

struct CANOPEN_APM_PCK_GET_STATE_CNF_Ttag /* Get state confirmation */
{
    TLR_PACKET_HEADER_T tHead; /** packet header */
    CANOPEN_APM_GET_STATE_CNF_DATA_T tData; /** packet data */
};
```

Packet Description

Structure Information CANOPEN_APS_PCK_GET_STATE_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSL VOID	Source End Point Identifier, untouched
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.1 Codes of the CANopen-APS-Task
	ulCmd	UINT32	0x00002E03	CANOPEN_APS_GET_STATE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	Structure CANOPEN_APS_GET_STATE_CNF_DATA_T			
	ulHighest MappedSend BufferNum	UINT32	0 .. 4	Number of the highest PDO mapped send triple buffer.
	ulHighest MappedRecv BufferNum	UINT32	0 .. 4	Number of the highest PDO mapped receive triple buffer.

Table 24: CANOPEN_APS_PCK_GET_STATE_CNF_T – Get State of AP-Task Confirmation

5.1.2 CANOPEN_APS_WARMSTART_REQ/CNF – Set Warmstart Parameters

This service can be used by the user application in order to configure the AP-task with warmstart parameters. After this request is received, the AP-task will configure the CANOpen slave task with the given parameters from this request and, if configured, starts the communication with the CANOpen network. This request will be denied if the configuration lock flag is set (for more information on this topic see section **Fehler! Verweisquelle konnte nicht gefunden werden. “Fehler! Verweisquelle konnte nicht gefunden werden.”**).

Note: The packet described in this section is obsolete and will not longer be supported after September, 1, 2009. Do not use this packet for all new developments! It is replaced by the packet CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration described in the next section and has to be used for new developments.

Packet Structure Reference

```
typedef struct CANOPEN_APS_WARMSTART_REQ_DATA_Ttag
    CANOPEN_APS_WARMSTART_REQ_DATA_T;

#define CANOPEN_APS_SYS_FLAG_COM_CONTROLLED_RELEASE    0x00000001L
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_ENABLED        0x00000002L
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_32_BIT         0x00000004L

#define CANOPEN_APS_WD_OFF                             0x00000000L
#define CANOPEN_APS_WD_MIN_TIMEOUT                    0x00000014L
#define CANOPEN_APS_WD_MAX_TIMEOUT                    0x0000FFFFL

#define CANOPEN_APS_CFG_FLAG_29_BIT                    0x00000001L

#define CANOPEN_SLAVE_MIN_SLAVE_NODE_ID    1
#define CANOPEN_SLAVE_MAX_SLAVE_NODE_ID    127

#define CANOPEN_SLAVE_CFG_BAUD_1000    0x00000000L    /* 1MBaud */
#define CANOPEN_SLAVE_CFG_BAUD_800    0x00000001L    /* 800kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_500    0x00000002L    /* 500kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_250    0x00000003L    /* 250kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_125    0x00000004L    /* 125kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_100    0x00000005L    /* 100kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_50     0x00000006L    /* 50kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_20     0x00000007L    /* 20kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_10     0x00000008L    /* 10kBaud */

struct CANOPEN_APS_WARMSTART_REQ_DATA_Ttag
{
    TLR_UINT32    ulSystemFlags;    /* System flags */
    TLR_UINT32    ulWdgTime;        /* Watchdog time */
    TLR_UINT32    ulNodeId;         /* Node ID */
    TLR_UINT32    ulBaudrate;       /* Baudrate */
    TLR_UINT32    ulCanopenFlags;   /* CANopen flags */
};

typedef struct CANOPEN_APS_PCK_WARMSTART_REQ_Ttag
    CANOPEN_APS_PCK_WARMSTART_REQ_T;

struct CANOPEN_APS_PCK_WARMSTART_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;    /** packet header */
    CANOPEN_APS_WARMSTART_REQ_DATA_T    tData;    /** packet data */
};
```

Packet Description

Structure Information CANOPEN_APS_PCK_WARMSTART_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPE NAPS	Destination Queue-Handle of CANopen slave-task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-task Process Queue
	ulDestId	UINT32	ulCANOPESL VT0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	20	Packet Data Length (In Bytes)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
	ulSta	UINT32		<i>See section 6.1 Codes of the CANopen-APS-Task</i>
	ulCmd	UINT32	0x00002E00	CANOPEN_APS_WARMSTART_REQ - Command
	ulExt	UINT32		Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32		Routing, do not touch

tData	Structure CANOPEN_APS_WARMSTART_REQ_DATA_T			
	ulSystemFlags	UINT32	0 1	<p>System Flags</p> <p>BIT 0: AUTOSTART / APPLICATION CONTROLLED communication with a controller after a device start is allowed without BUS_ON flag, but the communication will be interrupted if the BUS_ON flag changes state to 0</p> <p>communication with controller is allowed only with the BUS_ON flag.</p> <p>BIT 1: I/O STATUS DISABLED/ ENABLED Not supported yet</p> <p>BIT 2: IO STATUS 32 BIT Not supported yet</p> <p>BIT 3 - 31: Reserved for further use, set to zero</p>
	ulWdgTime	UINT32	0 20 .. 65535	<p>Watchdog supervision</p> <p>Watchdog supervision deactivated Watchdog time in milliseconds</p>
	ulNodeId	UINT32	1 .. 127	Node ID of CANopen slave
	ulBaudrate	UINT32		See Table 41: Codes and Corresponding Baud Rates of CANopen Network
	ulCanopenFlags	UINT32		<p>CANopen Flags</p> <p>BIT 0: 29-BIT IDENTIFIER DISABLED/ ENABLED Not supported yet</p> <p>BIT 1 - 31: Reserved for further use, set to zero</p>

Table 25: CANOPEN_APS_PCK_WARMSTART_REQ – Set Warmstart Parameter Request

Packet Structure Reference

```

typedef struct CANOPEN_APS_WARMSTART_CNF_DATA_Ttag
    CANOPEN_APS_WARMSTART_CNF_DATA_T;

#define CANOPEN_APS_SYS_FLAG_COM_CONTROLLED_RELEASE    0x00000001L
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_ENABLED        0x00000002L
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_32_BIT        0x00000004L

#define CANOPEN_APS_WD_OFF                            0x00000000L
#define CANOPEN_APS_WD_MIN_TIMEOUT                    0x00000014L
#define CANOPEN_APS_WD_MAX_TIMEOUT                    0x0000FFFFL

#define CANOPEN_APS_CFG_FLAG_29_BIT                    0x00000001L

#define CANOPEN_SLAVE_MIN_SLAVE_NODE_ID    1
#define CANOPEN_SLAVE_MAX_SLAVE_NODE_ID    127

#define CANOPEN_SLAVE_CFG_BAUD_1000    0x00000000L    /* 1MBaud */
#define CANOPEN_SLAVE_CFG_BAUD_800     0x00000001L    /* 800kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_500     0x00000002L    /* 500kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_250     0x00000003L    /* 250kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_125     0x00000004L    /* 125kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_100     0x00000005L    /* 100kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_50      0x00000006L    /* 50kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_20      0x00000007L    /* 20kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_10      0x00000008L    /* 10kBaud */

struct CANOPEN_APS_WARMSTART_CNF_DATA_Ttag
{
    TLR_UINT32    ulSystemFlags;    /* System flags */
    TLR_UINT32    ulWdgTime;        /* Watchdog time */
    TLR_UINT32    ulNodeId;        /* Node ID */
    TLR_UINT32    ulBaudrate;      /* Baudrate */
    TLR_UINT32    ulCanopenFlags;  /* CANopen flags */
};

typedef struct CANOPEN_APS_PCK_WARMSTART_CNF_Ttag
    CANOPEN_APS_PCK_WARMSTART_CNF_T;

struct CANOPEN_APS_PCK_WARMSTART_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;    /** packet header */
    CANOPEN_APS_WARMSTART_CNF_DATA_T    tData;    /** packet data */
};

```

Packet Description

Structure Information CANOPEN_APS_PCK_WARMSTART_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPESLVT0Id	Source End Point Identifier, untouched
	ulLen	UINT32	20	Packet Data Length (In Bytes)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as Unique Number
	ulSta	UINT32		<i>See section 6.1 Codes of the CANopen-APS-Task</i>
	ulCmd	UINT32	0x00002E01	CANOPEN_APS_WARMSTART_CNF - Command
	ulExt	UINT32		Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32		Routing, do not touch

tData	Structure CANOPEN_APS_WARMSTART_CNF_DATA_T			
	ulSystemFlags	UINT32	0 1	System Flags BIT 0: AUTOSTART / APPLICATION CONTROLLED communication with a controller after a device start is allowed without BUS_ON flag, but the communication will be interrupted if the BUS_ON flag changes state to 0 Communication with controller is allowed only with the BUS_ON flag. BIT 1: I/O STATUS DISABLED/ ENABLED Not supported yet BIT 2: IO STATUS 32 BIT Not supported yet BIT 3 - 31: Reserved for further use, set to zero
	ulWdgTime	UINT32	0 20 .. 65535	Watchdog supervision Watchdog supervision deactivated Watchdog time in milliseconds
	ulNodeId	UINT32	1 .. 127	Node ID of CANopen slave
	ulBaudrate	UINT32		See <i>Table 41: Codes and Corresponding Baud Rates of CANopen Network</i>
	ulCanopenFlags	UINT32		CANopen Flags BIT 0: 29-BIT IDENTIFIER DISABLED/ ENABLED Not supported yet BIT 1 - 31: Reserved for further use, set to zero

Table 26: CANOPEN_APS_PCK_WARMSTART_CNF_T – Set Warmstart Parameter Confirmation

5.1.3 CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration

This service can be used by the user application in order to configure the AP-task with warmstart parameters. After this request is received, the AP-task will configure the CANopen slave task with the given parameters from this request and, if configured, starts the communication with the CANopen network.

The following applies:

- Configuration parameters will be stored internally.
- In case of any error no data will be stored at all.
- A channel init is required to activate the parameterized data.
- This packet does not perform any registration at the stack automatically. Registering must be performed with a separate packet such as the registration packet described in the netX Dual-Port-Memory Manual (RCX_REGISTER_APP_REQ, code 0x2F10).
- This request will be denied if the configuration lock flag is set

(for more information on this topic see section **Fehler! Verweisquelle konnte nicht gefunden werden. "Fehler! Verweisquelle konnte nicht gefunden werden."**).

Packet Structure Reference

```

typedef struct CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag
    CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_T;

#define CANOPEN_APS_SYS_FLAG_COM_CONTROLLED_RELEASE    0x00000001L
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_ENABLED        0x00000002L
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_32_BIT        0x00000004L

#define CANOPEN_APS_WD_OFF                            0x00000000L
#define CANOPEN_APS_WD_MIN_TIMEOUT                   0x00000014L
#define CANOPEN_APS_WD_MAX_TIMEOUT                   0x0000FFFFL

#define CANOPEN_APS_CFG_FLAG_29_BIT                   0x00000001L

#define CANOPEN_APS_CFG_FLAG_VENDOR_ID                0x00000010L
#define CANOPEN_APS_CFG_FLAG_PRODUCT_CODE            0x00000020L
#define CANOPEN_APS_CFG_FLAG_SERIAL_NUMBER           0x00000040L
#define CANOPEN_APS_CFG_FLAG_REVISION_NUMBER         0x00000080L

#define CANOPEN_SLAVE_MIN_SLAVE_NODE_ID    1
#define CANOPEN_SLAVE_MAX_SLAVE_NODE_ID    127

#define CANOPEN_SLAVE_CFG_BAUD_1000    0x00000000L    /* 1MBaud */
#define CANOPEN_SLAVE_CFG_BAUD_800     0x00000001L    /* 800kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_500     0x00000002L    /* 500kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_250     0x00000003L    /* 250kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_125     0x00000004L    /* 125kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_100     0x00000005L    /* 100kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_50      0x00000006L    /* 50kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_20      0x00000007L    /* 20kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_10      0x00000008L    /* 10kBaud */

struct CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag
{
    TLR_UINT32  ulSystemFlags;    /* System flags */
    TLR_UINT32  ulWdgTime;       /* Watchdog time */
    TLR_UINT32  ulNodeId;        /* Node ID */
    TLR_UINT32  ulBaudrate;      /* Baudrate */
    TLR_UINT32  ulCanopenFlags;  /* CANopen flags */

    TLR_UINT32  ulVendorId;      /* Vendor ID */
    TLR_UINT32  ulProductCode;   /* Product code */
    TLR_UINT32  ulSerialNumber;  /* Serial number */
    TLR_UINT32  ulRevisionNumber; /* Revision number */

    TLR_UINT32  aulReserved[8];  /* Reserved */
};

typedef struct CANOPEN_APS_PCK_SET_CONFIGURATION_REQ_Ttag
    CANOPEN_APS_PCK_SET_CONFIGURATION_REQ_T;

struct CANOPEN_APS_PCK_SET_CONFIGURATION_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;    /** packet header */
    CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_T    tData;    /** packet data */
};

```

Packet Description

Structure Information CANOPEN_APS_PCK_SET_CONFIGURATION_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPE NAPS	Destination Queue-Handle of CANopen slave-task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-task Process Queue
	ulDestId	UINT32	ulCANOPESL VT0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	68	Packet Data Length (In Bytes)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
	ulSta	UINT32		<i>See section 6.1 Codes of the CANopen-APS-Task</i>
	ulCmd	UINT32	0x00002E04	CANOPEN_APS_SET_CONFIGURATION_REQ - Command
	ulExt	UINT32		Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32		Routing, do not touch

tData	Structure CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_T			
ulSystemFlags	UINT32	0 1	<p>System Flags</p> <p>BIT 0: AUTOSTART / APPLICATION CONTROLLED communication with a controller after a device start is allowed without BUS_ON flag, but the communication will be interrupted if the BUS_ON flag changes state to 0</p> <p>communication with controller is allowed only with the BUS_ON flag.</p> <p>BIT 1: I/O STATUS DISABLED/ ENABLED Not supported yet</p> <p>BIT 2: IO STATUS 32 BIT Not supported yet</p> <p>BIT 3 - 31: Reserved for further use, set to zero</p>	
ulWdgTime	UINT32	0 20 .. 65535	<p>Watchdog supervision</p> <p>Watchdog supervision deactivated</p> <p>Watchdog time in milliseconds</p>	
ulNodeId	UINT32	1 .. 127	Node ID of CANopen slave	
ulBaudrate	UINT32		See <i>Table 41: Codes and Corresponding Baud Rates of CANopen Network</i>	
ulCanopenFlags	UINT32		<p>CANopen Flags</p> <p>BIT 0: 29-BIT IDENTIFIER DISABLED/ ENABLED Not supported yet</p> <p>BIT 1- 3: Reserved for further use, set to zero</p> <p>BIT 4: Evaluate Vendor ID DISABLED/ENABLED</p> <p>BIT 5: Evaluate Product Code DISABLED/ENABLED</p> <p>BIT 6: Evaluate Serial Number DISABLED/ENABLED</p> <p>BIT 7: Evaluate Revision Number DISABLED/ENABLED</p> <p>BIT 8 - 31: Reserved for further use, set to zero</p>	
ulVendorId	UINT32		Vendor code if corresponding bit in parameter <code>ulCanopenFlags</code> is set	
ulProductCode	UINT32		Product code if corresponding bit in parameter <code>ulCanopenFlags</code> is set	
ulSerialNumber	UINT32		Serial number if corresponding bit in parameter <code>ulCanopenFlags</code> is set	
ulRevisionNumber	UINT32		Revision number if corresponding bit in parameter <code>ulCanopenFlags</code> is set	
aulReserved[8]	UINT32[]		Reserved for further use, set to zero	

Table 27: CANOPEN_APS_PCK_SET_CONFIGURATION_REQ – Set Warmstart Parameter Request

Packet Description

Structure Information CANOPEN_APS_PCK_SET_CONFIGURATION_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPESLVT0Id	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet Data Length (In Bytes)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as Unique Number
	ulSta	UINT32		See section 6.1 Codes of the CANopen-APS-Task
	ulCmd	UINT32	0x00002E05	CANOPEN_APS_SET_CONFIGURATION_CNF - Command
	ulExt	UINT32		Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32		Routing, do not touch

Table 28: CANOPEN_APS_PCK_SET_CONFIGURATION_CNF_T – Set Warmstart Parameter Confirmation

5.2 The CANopen Slave-Task

To get the handle of the process queue of the CANopen slave-task the Macro `TLR_QUE_IDENTIFY()` needs to be used.

ASCII queue name	Description
"QUE_CANOPENSLV"	Name of the CANopen slave-task process queue

Table 29 CANopen Slave-Task Process Queue

The returned handle has to be used as value `ulDest` in all request packets to be sent to the CANopen slave-Task. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDBUFFER_FIFO/LIFO()` for sending a packet to the CANopen slave-Task.

5.2.1 CANOPEN_SLAVE_REGISTER_REQ/CNF – Register Application

This packet is used in order to register to the CANopen slave task. After this request is performed successfully, indication packets are sent from the CANopen slave task to the source queue of this request packet.



Note: Use this packet only when working with linkable object modules. It is not designed for usage in the context of loadable firmware.



Note: This packet is used by the AP-task only and will not be routed from the user application to the CANopen slave-task.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_APP_REGISTER_REQ_DATA_Ttag
    CANOPEN_SLAVE_APP_REGISTER_REQ_DATA_T;

struct CANOPEN_SLAVE_APP_REGISTER_REQ_DATA_Ttag
{
    TLR_UINT32 ulReserved;
};

typedef struct CANOPEN_SLAVE_PACKET_APP_REGISTER_REQ_Ttag
    CANOPEN_SLAVE_PACKET_APP_REGISTER_REQ_T;

/** Structure of task command application register request*/
struct CANOPEN_SLAVE_PACKET_APP_REGISTER_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header.      */
    CANOPEN_SLAVE_APP_REGISTER_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_APP_REGISTER_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See Table 31: CANOPEN_SLAVE_REGISTER_REQ – Packet Status/Error
	ulCmd	UINT32	0x00002900	CANOPEN_SLAVE_REGISTER_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	structure CANOPEN_SLAVE_APP_REGISTER_REQ_DATA_T			
	ulReserved	UINT32		Reserved for further use, set to zero

Table 30: CANOPEN_SLAVE_PACKET_APP_REGISTER_REQ_T – Register Application Request

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok

Table 31: CANOPEN_SLAVE_REGISTER_REQ – Packet Status/Error

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_APP_REGISTER_CNF_DATA_Ttag
    CANOPEN_SLAVE_APP_REGISTER_CNF_DATA_T;

struct CANOPEN_SLAVE_APP_REGISTER_CNF_DATA_Ttag
{
    TLR_UINT32 ulReserved;
};

typedef struct CANOPEN_SLAVE_PACKET_APP_REGISTER_CNF_Ttag
    CANOPEN_SLAVE_PACKET_APP_REGISTER_CNF_T;

/** Structure of task command application register confirmation*/
struct CANOPEN_SLAVE_PACKET_APP_REGISTER_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header.          */
    CANOPEN_SLAVE_APP_REGISTER_CNF_DATA_T tData;  /** packet confirmation data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_APP_REGISTER_CNF				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLVOID	Source End Point Identifier, untouched
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See Table 33: CANOPEN_SLAVE_REGISTER_CNF – Packet Status/Error
	ulCmd	UINT32	0x00002901	CANOPEN_SLAVE_REGISTER_CNF - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change
tData	Structure CANOPEN_SLAVE_APP_REGISTER_CNF_DATA_T			
	ulReserved	UINT32		Reserved for further use, set to zero

Table 32: CANOPEN_SLAVE_PACKET_APP_REGISTER_CNF_T – Register Application Confirmation

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok
TLR_I_CANOPEN_SLAVE_INITIALIZE (0x40430026)	Slave is initializing.
TLR_E_CANOPEN_SLAVE_PACKET_LENGTH (0xC0430002)	Invalid length in packet.

Table 33: CANOPEN_SLAVE_REGISTER_CNF – Packet Status/Error

5.2.2 CANOPEN_SLAVE_EXCHANGE_DATA_REQ/CNF – Exchange Data

This command can be used to exchange send and receive object data with the CANopen network and can be used instead of exchanging data with the input and output image of the DPM. With each command, data of one receive object can be read and data of one send object can be written.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_Ttag
    CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_T;

#define CANOPEN_SLAVE_RECV_OBJECT_CNT          4

#define CANOPEN_SLAVE_MIN_RECV_IDX            0x2200
#define CANOPEN_SLAVE_MAX_RECV_IDX            0x2203

#define CANOPEN_SLAVE_MIN_RECV_SUB_IDX         1
#define CANOPEN_SLAVE_MAX_RECV_SUB_IDX         128

#define CANOPEN_SLAVE_SEND_OBJECT_CNT          4

#define CANOPEN_SLAVE_MIN_SEND_IDX             0x2000
#define CANOPEN_SLAVE_MAX_SEND_IDX             0x2003

#define CANOPEN_SLAVE_MIN_SEND_SUB_IDX         1
#define CANOPEN_SLAVE_MAX_SEND_SUB_IDX         128

struct CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_Ttag
{
    TLR_UINT32 ulRecvIndex;      /* Object index for recv data */
    TLR_UINT32 ulRecvSubIndex;   /* Object sub-index for recv data */
    TLR_UINT32 ulRecvDataCnt;    /* Recv data count */

    TLR_UINT32 ulSendIndex;      /* Object index for send Data */
    TLR_UINT32 ulSendSubIndex;   /* Object sub-index for send Data */
    TLR_UINT32 ulSendDataCnt;    /* Send data count */

    TLR_UINT8  abSendData[CANOPEN_SLAVE_MAX_SEND_SUB_IDX];
};

typedef struct CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_Ttag
    CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_T;

struct CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead; /** packet header. */
    CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	24 .. 152	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x0000290A	CANOPEN_SLAVE_EXCHANGE_DATA_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_T			
	ulRecvIndex	UINT32	2200h..2203h	Receive object index
	ulRecvSub Index	UINT32	1.. 128	Receive object sub-index
	ulRecvData Cnt	UINT32	0 ..128	Number data byte to be read
	ulSendIndex	UINT32	2000h..2003h	Send object index
	ulSendSub Index	UINT32	1.. 128	Send object sub-index
	ulSendData Cnt	UINT32	0 ..128	Number data byte to be sent
	abSendData [128]	UINT8[]		Send data

Table 34: CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_T – Exchange Data Request

Packet Structure Reference

```

typedef struct CANOPEN_SLAVE_EXCHANGE_DATA_CNF_DATA_Ttag
    CANOPEN_SLAVE_EXCHANGE_DATA_CNF_DATA_T;

#define CANOPEN_SLAVE_RECV_OBJECT_CNT          4

#define CANOPEN_SLAVE_MIN_RECV_IDX            0x2200
#define CANOPEN_SLAVE_MAX_RECV_IDX            0x2203

#define CANOPEN_SLAVE_MIN_RECV_SUB_IDX         1
#define CANOPEN_SLAVE_MAX_RECV_SUB_IDX         128

#define CANOPEN_SLAVE_SEND_OBJECT_CNT          4

#define CANOPEN_SLAVE_MIN_SEND_IDX             0x2000
#define CANOPEN_SLAVE_MAX_SEND_IDX             0x2003

#define CANOPEN_SLAVE_MIN_SEND_SUB_IDX         1
#define CANOPEN_SLAVE_MAX_SEND_SUB_IDX         128

struct CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_Ttag
{
    TLR_UINT32 ulRecvIndex;      /* Object index for recv data      */
    TLR_UINT32 ulRecvSubIndex;  /* Object sub-index for recv data  */
    TLR_UINT32 ulRecvDataCnt;   /* Recv data count                 */

    TLR_UINT32 ulSendIndex;     /* Object index for send Data      */
    TLR_UINT32 ulSendSubIndex;  /* Object sub-index for send Data  */
    TLR_UINT32 ulSendDataCnt;   /* Send data count                 */

    TLR_UINT8  abSendData[CANOPEN_SLAVE_MAX_SEND_SUB_IDX];
};

typedef struct CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_Ttag
    CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_T;

struct CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header.                */
    CANOPEN_SLAVE_EXCHANGE_DATA_CNF_DATA_T tData; /** packet confirmation data.      */
};

```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
	ulLen	UINT32	24 .. 152	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x0000290B	CANOPEN_SLAVE_EXCHANGE_DATA_CNF- Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change
tData	structure CANOPEN_SLAVE_EXCHANGE_DATA_CNF_DATA_T			
	ulRecvIndex	UINT32	2200h..2203h	Receive object index
	ulRecvSubIndex	UINT32	1.. 128	Receive object sub-index
	ulRecvDataCnt	UINT32	0 ..128	Number data byte to be read
	ulSendIndex	UINT32	2000h..2003h	Send object index
	ulSendSubIndex	UINT32	1.. 128	Send object sub-index
	ulSendDataCnt	UINT32	0 ..128	Number data byte to be sent
	abRecvData [128]	UINT8[]		Receive data

Table 35: CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_T –Exchange Data Confirmation

5.2.3 CANOPEN_SLAVE_STARTSTOP_REQ/CNF – Start/Stop CANopen Network

This packet starts or stops the communication with the CANopen network, depending on the value of the `ulMode` parameter.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_STARTSTOP_REQ_DATA_Ttag
    CANOPEN_SLAVE_STARTSTOP_REQ_DATA_T;

#define CANOPEN_SLAVE_STOP_CANOPEN    0x00000000L
#define CANOPEN_SLAVE_START_CANOPEN   0x00000001L

struct CANOPEN_SLAVE_STARTSTOP_REQ_DATA_Ttag
{
    TLR_UINT32 ulMode;
};

typedef struct CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_Ttag
    CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_T;

/** Structure of task command start/stop CANopen request */
struct CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header. */
    CANOPEN_SLAVE_STARTSTOP_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002902	CANOPEN_SLAVE_STARTSTOP_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_STARTSTOP_REQ_DATA_T			
	ulMode	UINT32	0 1	Depending on this assignment, communication is either started or stopped: Stop CANopen Start CANopen

Table 36: CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_T – Start/Stop Communication Request

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_STARTSTOP_CNF_DATA_Ttag
    CANOPEN_SLAVE_STARTSTOP_CNF_DATA_T;

#define CANOPEN_SLAVE_STOP_CANOPEN      0x00000000L
#define CANOPEN_SLAVE_START_CANOPEN     0x00000001L

struct CANOPEN_SLAVE_STARTSTOP_CNF_DATA_Ttag
{
    TLR_UINT32 ulMode;
};

typedef struct CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_Ttag
    CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_T;

/** Structure of task command start/stop communication request */
struct CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header.          */
    CANOPEN_SLAVE_STARTSTOP_CNF_DATA_T tData;  /** packet confirmation data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002903	CANOPEN_SLAVE_STARTSTOP_CNF - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change
tData	structure CANOPEN_SLAVE_STARTSTOP_CNF_DATA_T			
	ulMode	UINT32	0 1	Depending on this assignment, communication is either started or stopped: Stop CANopen Start CANopen

Table 37: CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_T – Start/Stop Communication Confirmation

5.2.4 CANOPEN_SLAVE_INITIALIZE_REQ/CNF – Initialization of CANopen Slave

This command is used in order to reset the CANopen slave.



Note: Use this packet preferably when working with linkable object modules. In the context of loadable firmware we recommend to use 'config reload' instead.



Note: This command does not delete configuration databases. If the CANopen slave is configured by configuration database, this configuration is reloaded again after the initialize command is completed.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_Ttag
    CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_T;

/** Structure of task command initialize CANopen slave request */
struct CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header.      */
    CANOPEN_SLAVE_INITIALIZE_REQ_DATA_T tData; /** packet request data. */
};

struct CANOPEN_SLAVE_INITIALIZE_REQ_DATA_Ttag
{
    TLR_UINT32 ulReserved;
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL VOID	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002904	CANOPEN_SLAVE_SLAVE_REQ – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_INITIALIZE_REQ_DATA_T			
	ulReserved	UINT32		Reserved for further use, set to zero

Table 38: CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_T – Initialization of CANopen Slave Request

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_Ttag
    CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_T;

/** Structure of task command initialize CANopen slave request */
struct CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header.          */
    CANOPEN_SLAVE_INITIALIZE_CNF_DATA_T tData; /** packet confirmation data. */
};

struct CANOPEN_SLAVE_INITIALIZE_CNF_DATA_Ttag
{
    TLR_UINT32 ulReserved;
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLVOID	Source End Point Identifier, untouched
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002905	CANOPEN_SLAVE_INITIALIZE_CNF- Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change
tData	Structure CANOPEN_SLAVE_INITIALIZE_CNF_DATA_T			
	ulReserved	UINT32		Reserved for further use, set to zero

Table 39: CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_T – Initialization of CANopen Slave Confirmation

5.2.5 CANOPEN_SLAVE_SET_BUSPARAM_REQ/CNF – Set Bus Parameters

This packet can be applied for setting the bus parameters for the CANopen slave. This request will be denied if the configuration lock flag is set.



Note: Use this packet preferably when working with linkable object modules. In the context of loadable firmware we recommend to use 'set configuration' or 'warmstart' instead.

Bus parameter Structure Reference

```
typedef struct CANOPEN_SLAVE_CFG_BUS_PARAM_Ttag
    CANOPEN_SLAVE_CFG_BUS_PARAM_T;

#define CANOPEN_SLAVE_MIN_SLAVE_NODE_ID 1
#define CANOPEN_SLAVE_MAX_SLAVE_NODE_ID 127

#define CANOPEN_SLAVE_CFG_BAUD_1000    0x00000000L /* 1MBaud */
#define CANOPEN_SLAVE_CFG_BAUD_800     0x00000001L /* 800kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_500     0x00000002L /* 500kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_250     0x00000003L /* 250kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_125     0x00000004L /* 125kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_100     0x00000005L /* 100kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_50      0x00000006L /* 50kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_20      0x00000007L /* 20kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_10      0x00000008L /* 10kBaud */

struct CANOPEN_SLAVE_CFG_BUS_PARAM_Ttag
{
    TLR_UINT32    ulSlaveNodeId; /* Node ID of CANopen slave */
    TLR_UINT32    ulBaudrate; /* Baudrate */
    TLR_BOOLEAN32 f29BitSelector; /* Enable 29-Bit identifier */
};

typedef struct CANOPEN_SLAVE_CFG_ADD_PARAM_Ttag
    CANOPEN_SLAVE_CFG_ADD_PARAM_T;

#define CANOPEN_SLAVE_CFG_VENDOR_ID    0x00000001L
#define CANOPEN_SLAVE_CFG_PRODUCT_CODE 0x00000002L
#define CANOPEN_SLAVE_CFG_SERIAL_NUMBER 0x00000004L

struct CANOPEN_SLAVE_CFG_ADD_PARAM_Ttag
{
    TLR_UINT32 ulFlags;
    TLR_UINT32 ulVendorId;
    TLR_UINT32 ulProductCode;
    TLR_UINT32 ulSerialNumber;
    TLR_UINT32 ulRevisionNumber;
    TLR_UINT32 aulReserved[12];
};
```

Variable	Type	Range	Explanation
ulSlaveNodeid	UINT32	1...127	Node ID of CANopen slave
ulBaudrate	UINT32		See Table 41: Codes and Corresponding Baud Rates of CANopen Network
f29BitSelector	BOOLEAN32		Enable 29 bit identifier Not supported yet, set to FALSE

Table 40: CANOPEN_SLAVE_CFG_BUS_PARAM_T - Bus Parameter Configuration

The variable `ulSlaveNodeid` indicating the node ID of the CANopen slave is required for the addressing of the device at the bus and has to be unique in the network. Therefore it is not allowed to use this number two times in the same network. Allowed values range from 1 to 127.

The baud rate of the CANopen network can be set using the `ulBaudRate` variable. The settings listed in the following table are applicable:

Value	Corresponding Baud Rate of CANopen Network
0	1 MBaud
1	800 KBaud
2	500 KBaud
3	250 KBaud
4	125 KBaud
5	100 KBaud
6	50 KBaud
7	20 KBaud
8	10 KBaud

Table 41: Codes and Corresponding Baud Rates of CANopen Network

The flag `f29BitSelector` decides whether 11-bit or 29-bit-addresses are used for the COB-IDs in the CANopen network. If the value is 1, then 29-bit-addresses are enabled otherwise 11-bit-addresses will be used. 29-bit mode is not supported yet.

Variable	Type	Range	Explanation
ulFlags		0 1 0 1 0 1 0 1	Additional configuration flags: Bit 0: CANOPEN_SLAVE_CFG_VENDOR_ID Parameter ulVendorId will not be evaluated Parameter ulVendorId will be evaluated Bit 1: CANOPEN_SLAVE_CFG_PRODUCT_CODE Parameter ulProductCode will not be evaluated Parameter ulProductCode will be evaluated Bit 2: CANOPEN_SLAVE_CFG_SERIAL_NUMBER Parameter ulSerialNumber will not be evaluated Parameter ulSerialNumber will be evaluated Bit 3: CANOPEN_SLAVE_CFG_SERIAL_NUMBER Parameter ulRevisionNumber will not be evaluated Parameter ulRevisionNumber will be evaluated Bit 4 .. 31: Reserved for further use, set to zero
ulVendorId			Vendor ID, used if corresponding bit in ulFlags parameter is set, otherwise vendor ID is set to zero
ulProductCode			Product code, used if corresponding bit in ulFlags parameter is set, otherwise product code is set to zero
ulSerialNumber			Serial number, used if corresponding bit in ulFlags parameter is set, otherwise serial number is set to zero
ulRevisionNumber			Revision number, used if corresponding bit in ulFlags parameter is set, otherwise revision number is set to default
aulReserved[12]	UINT32[]		Reserved for further use, set to zero

Table 42: CANOPEN_SLAVE_CFG_ADD_PARAM_T - Additional Configuration

The ulFlags parameter is used for setting the following parameters individually valid or invalid.

The ulVendorId parameter provides the value for the Vendor ID entry in the object dictionary of the CANopen slave (Object 1018h sub-index 1) if bit 0 in parameter ulFlags is set. Otherwise, the vendor ID is set to zero.

The ulProductCode parameter provides the value for the product code entry in the object dictionary of the CANopen slave (Object 1018h sub-index 2) if bit 1 in parameter ulFlags is set. Otherwise, the product code is set to zero.

The ulSerialNumber parameter provides the value for the serial number entry in the object dictionary of the CANopen slave (Object 1018h sub-index 4) if bit 2 in parameter ulFlags is set. Otherwise, the serial number is set to zero.

The ulRevisionNumber parameter provides the value for the revision number entry in the object dictionary of the CANopen slave (Object 1018h sub-index 3) if bit 3 in parameter ulFlags is set. Otherwise, the revision number is set to default.

Packet Structure Reference

```

typedef struct CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_Ttag
    CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T;

struct CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_Ttag
{
    CANOPEN_SLAVE_CFG_BUS_PARAM_T tCfgBusParam;
    CANOPEN_SLAVE_CFG_ADD_PARAM_T tCfgAddParam;
};

typedef struct CANOPEN_SLAVE_PACKET_SET_BUSPARAM_REQ_Ttag
    CANOPEN_SLAVE_PACKET_SET_BUSPARAM_REQ_T;

struct CANOPEN_SLAVE_PACKET_SET_BUSPARAM_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead; /** packet header.          */
    CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T tData; /** packet request data. */
};

```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SET_BUSPARAM_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... 2 ³² -1	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	80	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002906	CANOPEN_SLAVE_SET_BUSPARAM_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information

tData	Structure CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T	
	CANOPEN_SLAVE_CFG_BUS_PARAM_T	See Table 40: CANOPEN_SLAVE_CFG_BUS_PARAM_T - Bus Parameter
	CANOPEN_SLAVE_CFG_ADD_PARAM_T	See Table 42: CANOPEN_SLAVE_CFG_ADD_PARAM_T - Additional Configuration

Table 43: CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T – Set Bus Parameter Request

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_SET_BUSPARAM_CNF_DATA_Ttag
    CANOPEN_SLAVE_SET_BUSPARAM_CNF_DATA_T;

struct CANOPEN_SLAVE_SET_BUSPARAM_CNF_DATA_Ttag
{
    CANOPEN_SLAVE_CFG_BUS_PARAM_T tCfgBusParam;
    CANOPEN_SLAVE_CFG_ADD_PARAM_T tCfgAddParam;
};

typedef struct CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_Ttag
    CANOPEN_SLAVE_PACKET_SET_BUSPARAM_REQ_T;

struct CANOPEN_SLAVE_PACKET_SET_BUSPARAM_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead; /** packet header. */
    CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLVOID	Source End Point Identifier, untouched
	ulLen	UINT32	80	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002907	CANOPEN_SLAVE_SET_BUSPARAM_CNF - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change
tData	Structure CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T			
	CANOPEN_SLAVE_CFG_BUS_PARAM_T	See Table 40: CANOPEN_SLAVE_CFG_BUS_PARAM_T - Bus Parameter		
	CANOPEN_SLAVE_CFG_ADD_PARAM_T	See Table 42: CANOPEN_SLAVE_CFG_ADD_PARAM_T - Additional Configuration		

Table 44: CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_T –Set Bus Parameter Confirmation

5.2.6 CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ/CNF – Get Buffer Handle

Using this packet you can get a handle to the send and receive triple buffers.



Note: Use this packet only when working with linkable object modules. It is not designed for usage in the context of loadable firmware.



Note: This packet is used by the AP-task only and will not be routed from the user application to the CANopen slave-task.

The received handles can be used with the macros `TLR_GETEXCHGED_TRIBUFF()` and `TLR_EXCHANGE_TRIBUFF()` in order to exchange data with the CANopen network.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ_DATA_Ttag
    CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ_DATA_T;

struct CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ_DATA_Ttag
{
    TLR_UINT32 ulReserved;
};

typedef struct CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_REQ_Ttag
    CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_REQ_T;

struct CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /** packet header. */
    CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x0000290C	CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_GET_BUFFER_HANDLE_REQ_DATA_T			
	ulReserved	UINT32		Reserved, set to zero

Table 45: CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_REQ_T – Get Buffer Handle Request

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF_DATA_Ttag
    CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF_DATA_T;

#define CANOPEN_SLAVE_RECV_OBJECT_CNT    4
#define CANOPEN_SLAVE_SEND_OBJECT_CNT    4

struct CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF_DATA_Ttag
{
    TLR_UINT32  aulRecvBuffer[CANOPEN_SLAVE_RECV_OBJECT_CNT];
    TLR_UINT32  aulSendBuffer[CANOPEN_SLAVE_SEND_OBJECT_CNT];
};

typedef struct CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_CNF_Ttag
    CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_CNF_T;

struct CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_CNF_Ttag
{
    TLR_PACKET_HEADER_T                tHead; /** packet header. */
    CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_GET_BUFFER_HANDLE_CNF				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
	ulLen	UINT32	32	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x0000290D	CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change
tData	Structure CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF_DATA_T			
	aulRecvBuffer [4]	UINT32[]		Receive Buffer
	aulSendBuffer [4]	UINT32[]		Send Buffer

Table 46: CANOPEN_SLAVE_GET_BUFFER_HANDLE_CNF – Get Buffer Handle Confirmation

5.2.7 CANOPEN_SLAVE_STATE_CHANGE_IND/RES – Change of State Indication

This indication packet signifies a change of the state of the CANopen slave-task or the CANopen network. The indication delivers two important blocks containing status information about the CANopen slave, namely

- The slave state
- The extended slave state

These blocks delivering information about the change of state are described in detail below.



Note: Use this packet only when working with linkable object modules. It is not designed for usage in the context of loadable firmware.



Note: This indication is used by the AP-Task in order to set status information in the DPM and will not be routed to the user application.

In order to be able to receive this indication, the CANOPEN_SLAVE_REGISTER_REQ/CNF – Register Application request has to be executed by the AP-Task.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_Ttag
    CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_T;

struct CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_Ttag
{
    CANOPEN_SLAVE_SLAVE_STATE_T    tSlaveState;
    CANOPEN_SLAVE_EXTENDED_STATE_T tExtendedState;
};

typedef struct CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_Ttag
    CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_T;

struct CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;    /** packet header.          */
    CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_T tData; /** packet request data.    */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_T				
Type: Indication				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle of AP-Task Process Queue
	ulSrc	UINT32		Source Queue-Handle of CANopen slave-Task Process Queue
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	120	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002912	CANOPEN_SLAVE_STATE_CHANGE_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_T			
	tSlaveState	CANOPEN_SLAVE_SLAVE_STATE_T		Structure for slave state, see explanation below.
	tExtended State	CANOPEN_SLAVE_EXTENDED_STATE_T		Structure for extended slave state, see explanation below.

Table 47: CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_T – Change of State Indication

CANopen Slave State Structure Reference

```
typedef struct CANOPEN_SLAVE_SLAVE_STATE_Ttag
    CANOPEN_SLAVE_SLAVE_STATE_T;

#define CANOPEN_SLAVE_STATE_FLAG_RDY          0x00000001L
#define CANOPEN_SLAVE_STATE_FLAG_RUN         0x00000002L
#define CANOPEN_SLAVE_STATE_FLAG_COM        0x00000004L
#define CANOPEN_SLAVE_STATE_FLAG_BUS_ON     0x00000008L
#define CANOPEN_SLAVE_STATE_FLAG_COMM_ERROR 0x00000010L

struct CANOPEN_SLAVE_SLAVE_STATE_Ttag
{
    TLR_UINT32          ulCanState;
    CANOPEN_SLAVE_IO_STATUS_T tIoStatus;
    TLR_UINT32          ulFlags;
    TLR_UINT32          ulErrorCount;
    TLR_UINT32          ulCommError;
    TLR_UINT32          ulRunLedState;
    TLR_UINT32          ulErrLedState;
    TLR_UINT32          ulRecvDataCnt;
    TLR_UINT32          ulSendDataCnt;

    TLR_UINT32          ulReserved;
};
```

CANopen Slave IO Status Structure Reference

```
typedef struct CANOPEN_SLAVE_IO_STATUS_Ttag
    CANOPEN_SLAVE_IO_STATUS_T;

struct CANOPEN_SLAVE_IO_STATUS_Ttag
{
    TLR_UINT32 ulHighestMappedSendBufferNum;
    TLR_UINT32 ulHighestMappedRecvBufferNum;
};
```

Extended Slave State Structure Reference

```
typedef struct CANOPEN_SLAVE_EXTENDED_STATE_Ttag
    CANOPEN_SLAVE_EXTENDED_STATE_T;

#define CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_INIT          0x00000001L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_ACTIVE       0x00000002L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_PASSIVE          0x00000004L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_BUS_OFF          0x00000008L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_RX_OVERFLOW      0x00000010L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_TX_OVERFLOW      0x00000020L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_WDG              0x00000100L
#define CANOPEN_SLAVE_EXT_STATE_CTRL                  0x00001000L
#define CANOPEN_SLAVE_EXT_STATE_NRDY                  0x00002000L
#define CANOPEN_SLAVE_EXT_STATE_TIMEOUT               0x00004000L

#define CANOPEN_SLAVE_EXT_STATE_UNKNOWN                0x00000000L
#define CANOPEN_SLAVE_EXT_STATE_OPERATIONAL           0x00000001L
#define CANOPEN_SLAVE_EXT_STATE_STOP                  0x00000002L
#define CANOPEN_SLAVE_EXT_STATE_PRE_OPERATIONAL       0x00000008L
#define CANOPEN_SLAVE_EXT_STATE_INITIALISING          0x000000FFL

#define CANOPEN_SLAVE_ADD_DETAIL_SIZE                  0x00000003L

__PACKED_PRE struct CANOPEN_SLAVE_EXTENDED_STATE_Ttag
{
    TLR_UINT32 ulFlags;
    TLR_UINT32 ulNodeState;
    TLR_UINT32 ulBusOffEveCnt;
    TLR_UINT32 ulErrorPassiveEveCnt;
    TLR_UINT32 ulRxOverflowCnt;
    TLR_UINT32 ulTxOverflowCnt;
    TLR_UINT32 ulReserved;
    TLR_UINT32 ulTimeoutCnt;

    TLR_UINT32 aulReserved[4];

    TLR_UINT32 ulDiagInfoCount;
    TLR_UINT32 ulLastDiagInfo;
    TLR_UINT32 ulMaxRecvIdx;
    TLR_UINT32 ulMaxSendIdx;
    TLR_UINT32 aulAddDetail[CANOPEN_SLAVE_ADD_DETAIL_SIZE];
}__PACKED_POST;
```

The extended slave state is described in detail in [section 3.3.2 "Extended Status"](#).

Packet Structure Reference

```

typedef struct CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_Ttag
    CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_T;

struct CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_Ttag
{
    TLR_PACKET_HEADER_T tHead; /** packet header.          */
};

```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_T				
Type: Response				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSLVOID	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002913	CANOPEN_SLAVE_STATE_CHANGE_RES - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change

Table 48: CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_T – Change of State Response

5.2.8 CANOPEN_SLAVE_SEND_EMCY_REQ/CNF – Send Emergency Message

This packet sends an emergency telegram to the CANopen network.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_Ttag
    CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_T;

#define CANOPEN_SLAVE_EMCY_DATA_SIZE 5

#define CANOPEN_SLAVE_ERROR_REGISTER_ERROR_RESET          0x00
#define CANOPEN_SLAVE_ERROR_REGISTER_GENERIC_BIT          0x01
#define CANOPEN_SLAVE_ERROR_REGISTER_CURRENT_BIT          0x02
#define CANOPEN_SLAVE_ERROR_REGISTER_VOLTAGE_BIT          0x04
#define CANOPEN_SLAVE_ERROR_REGISTER_TEMPERATURE_BIT      0x08
#define CANOPEN_SLAVE_ERROR_REGISTER_COMM_ERROR_BIT       0x10
#define CANOPEN_SLAVE_ERROR_REGISTER_DEV_PROFILE_BIT      0x20
#define CANOPEN_SLAVE_ERROR_REGISTER_RESERVED_BIT         0x40
#define CANOPEN_SLAVE_ERROR_REGISTER_MANU_SPEC_BIT        0x80

struct CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_Ttag
{
    TLR_UINT16 usErrorCode;
    TLR_UINT8  abManErrorCode[CANOPEN_SLAVE_EMCY_DATA_SIZE];
    TLR_UINT8  bErrorRegister;
};

typedef struct CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_Ttag
    CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_T;

struct CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead; /** packet header.          */
    CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002918	CANOPEN_SLAVE_SEND_EMCY_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_T			
	usErrorCode	UINT16		Error Code
	abManError Code[5]	UINT8[]		Area for Error Code
	bErrorRegist er	UINT8		See Table 50: Error Register

Table 49: CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_T – Send Emergency Message Request

The bits of the error register have the following meaning:

Error	Code
CANOPEN_SLAVE_ERROR_REGISTER_GENERIC_BIT	0x01
CANOPEN_SLAVE_ERROR_REGISTER_CURRENT_BIT	0x02
CANOPEN_SLAVE_ERROR_REGISTER_VOLTAGE_BIT	0x04
CANOPEN_SLAVE_ERROR_REGISTER_TEMPERATURE_BIT	0x08
CANOPEN_SLAVE_ERROR_REGISTER_COMM_ERROR_BIT	0x10
CANOPEN_SLAVE_ERROR_REGISTER_DEV_PROFILE_BIT	0x20
CANOPEN_SLAVE_ERROR_REGISTER_RESERVED_BIT	0x40
CANOPEN_SLAVE_ERROR_REGISTER_MANU_SPEC_BIT	0x80

Table 50: Error Register

Packet Structure Reference

```

typedef struct CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_Ttag
    CANOPEN_SALVE_PACKET_SEND_EMCY_CNF_T;

struct CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /** packet header. */
};

```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLVOID	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x00002919	CANOPEN_SALVE_SEND_EMCY_CNF - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change

Table 51: CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_T – Send Emergency Message Confirmation

5.2.9 CANOPEN_SLAVE_SET_NMT_STATE_REQ/CNF – Set NMT State

This packet allows you to set the NMT state of the CANopen slave. Normally the state is set by the NMT Master, but sometimes it may be necessary to control the state manually from the host application.

Which state is set depends on the value of the variable `ulNmtState`, which may have the values described in the following table:

Value	Symbolic Name	Meaning
1	CANOPEN_SLAVE_SET_NMT_STATE_OPERATIONAL	Operational
2	CANOPEN_SLAVE_SET_NMT_STATE_STOP	Stop
128	CANOPEN_SLAVE_SET_NMT_STATE_PRE_OPERATIONAL	Pre-operational
129	CANOPEN_SLAVE_SET_NMT_STATE_RESET_NODE	Reset node
130	CANOPEN_SLAVE_SET_NMT_STATE_RESET_COMM	Reset communication

Table 52: NMT States

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_Ttag
    CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_T;

#define CANOPEN_SLAVE_SET_NMT_STATE_OPERATIONAL    0x00000001L
#define CANOPEN_SLAVE_SET_NMT_STATE_STOP          0x00000002L
#define CANOPEN_SLAVE_SET_NMT_STATE_PRE_OPERATIONAL 0x00000080L
#define CANOPEN_SLAVE_SET_NMT_STATE_RESET_NODE    0x00000081L
#define CANOPEN_SLAVE_SET_NMT_STATE_RESET_COMM    0x00000082L

struct CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_Ttag
{
    TLR_UINT32 ulNmtState;
};

typedef struct CANOPEN_SLAVE_PACKET_SET_NMT_STATE_REQ_Ttag
    CANOPEN_SLAVE_PACKET_SET_NMT_STATE_REQ_T;

struct CANOPEN_SLAVE_PACKET_SET_NMT_STATE_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead; /** packet header.          */
    CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_T tData; /** packet data.          */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x0000291A	CANOPEN_SLAVE_SET_NMT_STATE_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_NODE_NMT_COMMAND_REQ_DATA_T			
	ulNmtState	UINT32		See Table 52: NMT States

Table 53: CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_REQ_T – Set NMT State Request

Packet Structure Reference

```

typedef struct CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_Ttag
    CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_T;

#define CANOPEN_SLAVE_SET_NMT_STATE_OPERATIONAL      0x00000001L
#define CANOPEN_SLAVE_SET_NMT_STATE_STOP            0x00000002L
#define CANOPEN_SLAVE_SET_NMT_STATE_PRE_OPERATIONAL  0x00000080L
#define CANOPEN_SLAVE_SET_NMT_STATE_RESET_NODE      0x00000081L
#define CANOPEN_SLAVE_SET_NMT_STATE_RESET_COMM      0x00000082L

struct CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_Ttag
{
    TLR_UINT32 ulNmtState;
};

typedef struct CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_CNF_Ttag
    CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_CNF_T;

struct CANOPEN_SLAVE_PACKET_SET_NMT_STATE_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead;                /** packet header.          */
    CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_T tData; /** packet data.          */
};

```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLVOID	Source End Point Identifier, untouched
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x0000291B	CANOPEN_SLAVE_SET_NMT_STATE_CNF - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change
tData	Structure CANOPEN_SLAVE_NODE_NMT_COMMAND_REQ_DATA_T			
	ulNmtState	UINT32		See Table 52: NMT States

Table 54: CANOPEN_SLAVE_PACKET_NODE_NMT_COMMAND_CNF_T – Set NMT State Confirmation

5.2.10 CANOPEN_SLAVE_SET_WATCHDOG_FAIL_REQ/CNF – Set Watchdog Fail

This packet is used by the AP-Task in order to inform the CANopen slave-task that a watchdog failure has been detected. The CANopen slave-task stops the communication with the CANopen network. After a watchdog error has been set, the slave has to be reinitialized before further communication is possible.



Note: Use this packet only when working with linkable object modules. It is not designed for usage in the context of loadable firmware.



Note: This packet is used by the AP-task only and will not be routed from the user application to the CANopen slave-task.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_Ttag
    CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_T;

struct CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /** packet header.          */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... 2 ³² -1	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL VOID	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x000029AA	CANOPEN_SLAVE_SET_WATCHDOG_FAIL_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information

Table 55: CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_T – Set Watchdog Fail Request

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_Ttag
    CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_T;

struct CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /** packet header.          */
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLVOID	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x000029AB	CANOPEN_SLAVE_SET_WATCHDOG_FAIL_CNF-Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change

Table 56: CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_T – Set Watchdog Fail Confirmation

5.2.11 CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ/CNF – Setup PDO Indication

This request can be used by the application for enabling and disabling PDO indications. While enabled, the CANopen slave sends an indication with each received PDO to the application. The PDO indication is described in section CANOPEN_SLAVE_RECEIVE_PDO_IND/RES – Receive PDO in this manual.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_Ttag
    CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_T;

#define CANOPEN_SLAVE_SETUP_PDO_INDICATION_DISABLE 0x000000001L
#define CANOPEN_SLAVE_SETUP_PDO_INDICATION_ENABLE 0x000000002L

struct CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_Ttag
{
    TLR_UINT32 ulSetupPdoIndication;
};

typedef struct CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_Ttag
    CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_T;

struct CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead;
    CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_T tData;
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x000029BA	CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ-Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_T			
	ulSetupPdoIndication	UINT32	0x00000001 0x00000002	Disable PDO Indications Enable PDO Indication

Table 57: CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_T – Setup PDO Indication Request

Packet Structure Reference

```

typedef struct CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_Ttag
    CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_T;

struct CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead
};

```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
	ulSrcId	UINT32	ulCANOPENSLVOID	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x000029BB	CANOPEN_SLAVE_SETUP_PDO_INDICATION_CNF-Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change

Table 58: CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_T – Setup PDO Indication Confirmation

5.2.12 CANOPEN_SLAVE_RECEIVE_PDO_IND/RES – Receive PDO

The following indication is sent from the CANopen slave to the application each time a PDO is received. The indication includes the PDO number, identifier, length and data.

Note: No PDO indications are sent to the application until this functionality is enabled. Enabling PDO indications is described in section CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ/CNF – Setup PDO Indication in this manual.

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_Ttag
    CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_T;

#define CANOPEN_SLAVE_RECEIVE_PDO_IND_MAX_DATA 8

struct CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_Ttag
{
    TLR_UINT32 ulPdoNumber;
    TLR_UINT32 ulIdentifier;
    TLR_UINT32 ulLength;
    TLR_UINT8  abPdoData[CANOPEN_SLAVE_RECEIVE_PDO_IND_MAX_DATA];
};
```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle of AP-Task Process Queue
	ulSrc	UINT32		Source Queue-Handle of CANopen slave-Task Process Queue
	ulDestId	UINT32	ulCANOPENSLV0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	12 .. 20	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 6.2 Codes of the CANopen Slave-Task
	ulCmd	UINT32	0x000029BC	CANOPEN_SLAVE_RECEIVE_PDO_IND- Indication
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing Information
tData	Structure CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_T			
	ulPdoNumber	UINT32	1..64	Number of received PDO
	ulIdentifier	UINT32	0..2047	Identifier of received PDO
	ulLength	UINT32	0..8	Length of received PDO
	abPdoData[8]	UINT8[]		Data of received PDO

Table 59: CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_T – Receive PDO Indication

Packet Structure Reference

```

typedef struct CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_Ttag
    CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_T;

struct CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_Ttag
{
    TLR_PACKET_HEADER_T tHead; /** packet header.          */
};

```

Packet Description

Structure Information CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_T				
Type: Response				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle of CANopen slave-Task Process Queue
	ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
	ulDestId	UINT32	ulCANOPENSLVOID	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		<i>See section 6.2 Codes of the CANopen Slave-Task</i>
	ulCmd	UINT32	0x000029BD	CANOPEN_SLAVE_RECEIVE_PDO_RES - Command
	ulExt	UINT32		Extension, reserved
	ulRout	UINT32		Routing Information, do not change

Table 60: CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_T – Receive PDO Response

6 Status/Error Codes Overview

6.1 Codes of the CANopen-APS-Task

6.1.1 Error Messages

Definition / (Value)	Definition / Description
0x00000000	TLR_S_OK Status ok
0xC0000001	TLR_E_FAIL Common error, detailed error information optionally present in the data area of packet
0xC04A0002	TLR_E_CANOPEN_APS_DATABASE_FOUND Configuration database found.
0xC04A0003	TLR_E_CANOPEN_APS_NODE_ID_PARAMETER Invalid parameter for node id.
0xC04A0004	TLR_E_CANOPEN_APS_BAUDRATE_PARAMETER Invalid parameter for baudrate.
0xC04A0005	TLR_E_CANOPEN_APS_STATE Request not possible in current state.
0x404A0007	TLR_I_CANOPEN_APS_OPEN_DBM_FILE Failed to open configuration database.
0xC04A0008	TLR_E_CANOPEN_APS_DATASET Failed to open configuration dataset.
0xC04A0009	TLR_E_CANOPEN_APS_TABLE_GLOBAL Failed to open GLOBAL configuration dataset.
0xC04A000A	TLR_E_CANOPEN_APS_TABLE_BUS_CAN Failed to open BUS_CAN configuration dataset.
0xC04A000B	TLR_E_CANOPEN_APS_SIZE_TABLE_BUS_CAN Invalid size of BUS_CAN configuration dataset.
0x404A000C	TLR_I_CANOPEN_APS_CONFIG_LOCK Configuration is locked.
0xC04A000D	TLR_E_CANOPEN_APS_PACKET_LENGTH Invalid packet length.
0xC04A000E	TLR_E_CANOPEN_APS_WATCHDOG_PARAMETER Invalid parameter for watchdog supervision.
0xC04A000FL	TLR_E_CANOPEN_APS_WATCHDOG_ACTIVATE Failed to activate watchdog supervision.

Table 61: Error Messages of the AP-Task

6.2 Codes of the CANopen Slave-Task

6.2.1 Error Messages

The following table defined the error messages of the CANopen slave-task:

Definition / (Value)	Description
0x00000000	TLR_S_OK Status ok
0xC0000001	TLR_E_FAIL Common error, detailed error information optionally present in the data area of packet
0xC0430003	TLR_E_CANOPEN_SLAVE_DATA_COUNT Invalid data count.
0xC0430004	TLR_E_CANOPEN_SLAVE_DATA_OFFSET Invalid data offset.
0xC0430005	TLR_E_CANOPEN_SLAVE_DATA_COUNT_WITH_OFFSET Invalid data count in combination with offset.
0xC0430006	TLR_E_CANOPEN_SLAVE_MODE Invalid mode in command.
0xC0430007	TLR_E_CANOPEN_SLAVE_STATE Command is not allowed in current state.
0xC0430009	TLR_E_CANOPEN_SLAVE_BUS_RUNNING Command is not allowed because CANopen is running.
0xC043000A	TLR_E_CANOPEN_SLAVE_BUS_PARAM_ALREADY_SET Bus parameters are already configured.
0xC043000B	TLR_E_CANOPEN_SLAVE_LOCAL_NODE_ID Invalid Node ID for CANopen slave.
0xC043000C	TLR_E_CANOPEN_SLAVE_BAUDRATE Invalid Baudrate.
0xC043000D	TLR_E_CANOPEN_SLAVE_29BIT_SELECTOR Invalid parameter for 29 bit selector.
0x4043000F	TLR_I_CANOPEN_SLAVE_ALREADY_IN_STATE Slave is already in requested state.
0xC0430010	TLR_E_CANOPEN_SLAVE_SEND_EMCY Send emergency-telegram failed.
0xC0430011	TLR_E_CANOPEN_SLAVE_INIT_LIB Failed to initialize CANopen library.
0xC0430012	TLR_E_CANOPEN_SLAVE_ERROR_PASSIVE CANopen is in error-passive state.

Definition / (Value)	Description
0xC0430013	TLR_E_CANOPEN_SLAVE_BUS_OFF CANopen is in bus-off state.
0xC0430014	TLR_E_CANOPEN_SLAVE_PUT_OBJECT_DATA Failed to write object data.
0xC0430015	TLR_E_CANOPEN_SLAVE_SET_OBJECT_DATA_VALID Failed to set object data valid.
0xC0430016	TLR_E_CANOPEN_SLAVE_GET_OBJECT_DATA Failed to get object data.
0xC0430017	TLR_E_CANOPEN_SLAVE_WRITE_PDO_REQ Failed to transmit PDO.
0xC0430018	TLR_E_CANOPEN_SLAVE_GUARD_ERROR Guard error detected.
0xC0430019	TLR_E_CANOPEN_SLAVE_INIT_BUFFER Initialization of buffer failed.
0xC043001A	TLR_E_CANOPEN_SLAVE_DL_REQ_FAILED CAN-DL request failed.
0xC043001B	TLR_E_CANOPEN_SLAVE_INVALID_INDEX Invalid object index.
0xC043001C	TLR_E_CANOPEN_SLAVE_INVALID_SUB_INDEX Invalid sub-index.
0xC043001D	TLR_E_CANOPEN_SLAVE_INVALID_MAP_LENGTH Invalid mapping length.
0xC043001E	TLR_E_CANOPEN_SLAVE_INVALID_PDO_MODE Invalid transmission mode for PDO.
0xC043001F	TLR_E_CANOPEN_SLAVE_INVALID_PDO_LENGTH Invalid length for PDO.
0xC0430020	TLR_E_CANOPEN_SLAVE_NO_WRITE_PERM No write permission for object.
0xC0430021	TLR_E_CANOPEN_SLAVE_NO_READ_PERM No read permission for object.
0xC0430022	TLR_E_CANOPEN_SLAVE_VALUE_TOO_LOW Value for object too low.
0xC0430023	TLR_E_CANOPEN_SLAVE_VALUE_TOO_HIGH Value for object too high.
0xC0430024	TLR_E_CANOPEN_SLAVE_INVALID_PARAMETER Invalid parameter for object.

Definition / (Value)	Description
0xC0430025	TLR_E_CANOPEN_SLAVE_INVALID_PDO_STATE Invalid PDO state.
0x40430026	TLR_I_CANOPEN_SLAVE_INITIALIZE Slave is initializing.
0xC0430027L	TLR_E_CANOPEN_SLAVE_OBJECT_SIZE Invalid size for object.

Table 62: Error Messages of the CANopen slave-Task

7 Contact

Headquarters

Germany

Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support

Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China

Hilscher Ges.f.Systemaut. mbH
Shanghai Representative Office
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support

Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France

Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support

Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India

Hilscher India Pvt. Ltd.
New Delhi - 110 025
Phone: +91 11 40515640
E-Mail: info@hilscher.in

Italy

Hilscher Italia srl
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support

Phone: +39/02 25007068
E-Mail: it.support@hilscher.com

Japan

Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support

Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea

Hilscher Korea Inc.
Suwon-Si, 443-810
Phone: +82-31-204-6190
E-Mail: info@hilscher.kr

Switzerland

Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support

Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA

Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support

Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com