Protocol API

CANopen Master

V2.14.0

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC070501API16EN | Revision 16 | English | 2016-05 | Released | Public

Table of Contents

1 Introduction

1.1 About this Document

This manual describes the application interface of the CANopen Master 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 List of Revisions

Rev	Date	Name	Chapter	Revision
13	2013-05-23	RG		Firmware/ stack version V2.11.x.x Reference to netX Dual-Port Memory Interface Manual Revision 12 Added description of new packet for PDO counter support: `CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ/CNF` – Enable/Disable PDO Counter Corrected offset in *Extended Status*
14	2013-09-24	RG		Firmware/ stack version V2.11.x.x Reference to netX Dual-Port Memory Interface Manual Revision 12 Added description of parameter `usErrorCode` in packet `CANOPEN_MASTER_SEND_EMCY_REQ/CNF` – Send Emergency Message Technical data: Maximum number of cyclic input data is still equal to 3584
15	2016-02-16	RG, ES		Firmware/ stack version V2.13.0 Description of *Common Status Block* updated (Version 2 used) Sections `CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ/CNF` – Force compare values, `CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ/CNF` – Configure SYNC Trigger, `CANOPEN_MASTER_SEND_SYNC_REQ/CNF` – Send SYNC and `CANOPEN_MASTER_SYNC_IND/RES` – SYNC Indication added.
16	2016-05-11	RG, ES		Firmware/ stack version V2.14.x.x Reference to netX Dual-Port Memory Interface Manual Revision 12 Added section 4.5 "*Handshake Modes and Synchronization*" New allowed parameter values in section 5.2.13 "*`CANOPEN_MASTER_NODE_NMT_COMMAND_REQ/CNF` – Set NMT State* Text corrections in section 5.2.21 "*`CANOPEN_MASTER_RESET_ERROR_REQ/CNF` – Reset Error Event*"

Table 1: List of Revisions

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 is helpful

1.4 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

1.5 Specifications

The data below applies to CANopen Master firmware and stack version V2.14.0.

The firmware/stack has been designed in order to meet the CiA Work Draft 301 V4.02 specification.

1.5.1 Technical Data

Maximum number of cyclic input data	3584 bytes
Maximum number of cyclic output data	3584 bytes
Maximum number of supported slaves	126
Maximum number of receive PDOs	512
Maximum number of transmit PDOs	512
Exchange of process data	via PDO transfer (synchronized, remotely requested and event driven (change of date))
Acyclic communication	SDO Upload/Download, max. 512 bytes per request
Functions	Emergency message (consumer and producer)
	Node guarding / life guarding, heartbeat
	PDO mapping
	NMT Master
	SYNC protocol (producer)
	Simple boot-up process, reading object 1000H for identification
CAN layer 2 access	Send/receive via API supported
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	no
netX 100, netX 500	yes

PCI

DMA Support for PCI targets	yes

Slot Number

Slot number supported for	CIFX 50-CO

Configuration

Configuration by tool SYCON.net (Download or exported configuration file named config.nxd)

Configuration by tool SyCon (exported configuration file named config.dbm)

Configuration by packets to transfer bus and node parameters

Diagnostic

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

1.6 Terms, Abbreviations and Definitions

Term	Description
AP (-task)	Application (-task) on top of the stack
Boot up	Initial sequence of node during start-up
CAN	Controller Area Network
CAN-DL	CAN Data Link Layer
CiA	CAN in Automation (CAN User Organization located in Erlangen, Germany)
COB-ID	Communication Object Identifier
DPM	Dual Port Memory
EMCY	Emergency
Guarding	Supervision of node
NMT	Network Management
PDO	Process Data Object (process data channel)
PDO-Mapping	Configurable process data per PDO
RxPDO	Receive PDO
SDO	Service Data Object (representing an acyclic data channel)
SYNC	Synchronization cycle of the CANopen slave
TxPDO	Transmit PDO

Table 2: Terms, Abbreviations and Definitions

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

All IP addresses in this document have host byte order.

1.7 References

This document based on the following specification respectively documents:

[1] CAN in Automation e.V., Erlangen: CANopen Application Layer and Communication Profile, CiA Draft Standard 301, Version 4.02, 2005

[2] Hilscher Gesellschaft für Systemautomation mbh: netX Dual-Port Memory Interface Manual. Revision 12, 2012

[3] Hilscher Gesellschaft für Systemautomation mbH: CAN Data Link Packet Interface Protocol API Manual, Revision 3, 2010-2013

Table 3: References

1.8 Legal Notes

1.8.1 Copyright

1.8.2 Important Notes

The 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 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 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 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.

1.8.3 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.

1.8.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, 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:

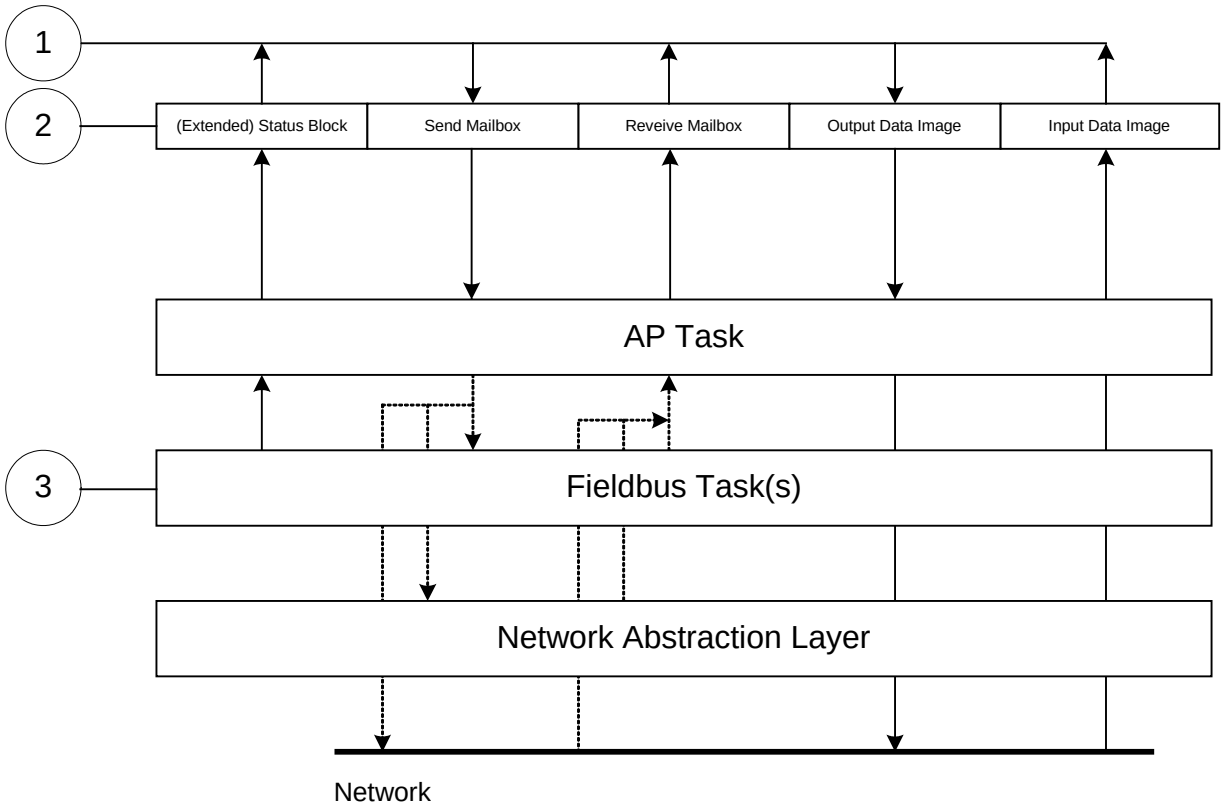

Figure 1: General Access Mechanisms on netX Systems

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 *Obtaining Diagnostic Information from connected Slaves by sending an RCX_GET_SLAVE_CONN_INFO_REQ Packet*on page 52 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 *Obtaining Diagnostic Information from connected Slaves by sending an RCX_GET_SLAVE_CONN_INFO_REQ Packet*. 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 Master-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 Master-Task which you have to use as current value for the first parameter (`pszIdn`) is

ASCII queue name	Description
"QUE_CANOPENMST"	Name of the CANopen Master-Task process queue
"QUE_CANOPENAPM"	Name of the APM-Task process queue

Table 4: 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 Master. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDPACKET_FIFO/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
ulDest	Application mailbox used for confirmation
ulSrc	Queue handle returned by TLR_QUE_IDENTIFY() as described above.
ulSrcId	Used for addressing at a lower level

Table 5: 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 Master 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.

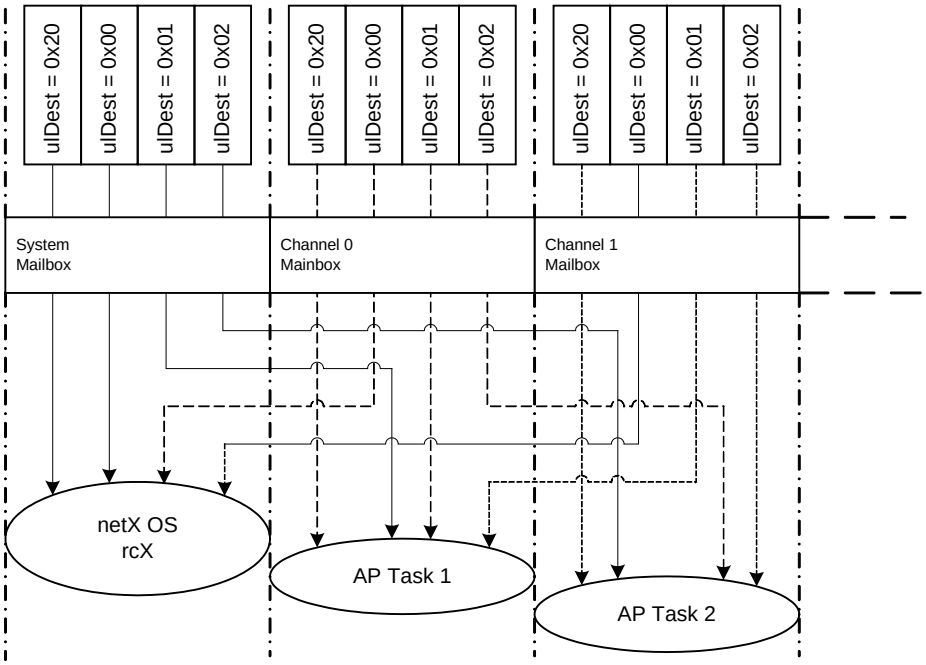

Figure 2: Use of `ulDest` *in Channel and System Mailbox*

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 6: 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:

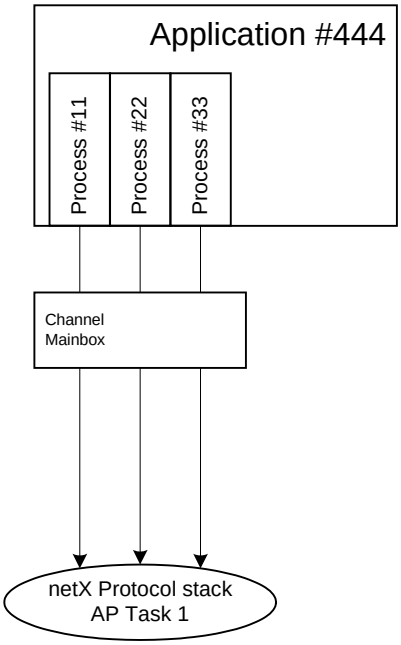

Figure 3: Using `ulSrc` *and* `ulSrcId`

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 7: 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:

3. Start with reading the channel information block within the system channel (usually starting at address `0x0030`).
4. 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.

5. 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.

6. 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.

7. 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, signaling 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). This can be done using the RCX_REGISTER_APP_REQ packet. For more information how to use this packet for registration, see the DPM manual (reference [2]), chapter 4.18 “Register / Unregister an Application. 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).

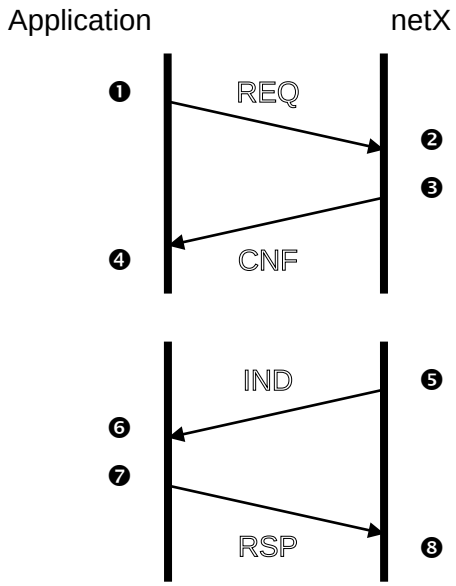

Figure 4: Transition Chart Application as Client

❶ ❷ The host application sends request packets to the netX firmware.

❸ ❹ The netX firmware sends a confirmation packet in return.

❺ ❻ The host application receives indication packets from the netX firmware.

❼ ❽ 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 DPV1 packets).

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).

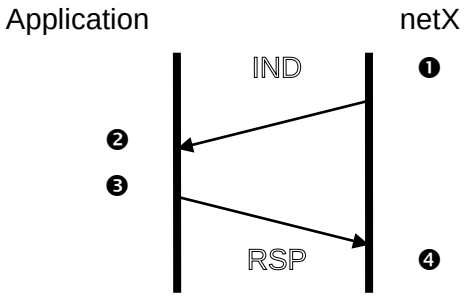

Figure 5: Transition Chart Application as Server

❶ ❷ The netX firmware passes an indication packet through the mailbox.

❸ ❹ 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).

Note: 48 byte are used for status information (16 byte for list of configured slaves, 16 byte for list of activated slaves and 16 byte for list of slaves with faults or errors). The contents of these 48 byte is identical to the contents of the second part of the Extended Status Block beginning at address 0x0100, see *Table 28: Extended Status Block for CANopen-Master – Second part (State Field Definition Block definition of the bit list state fields for CANopen Master).*

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

Table 8: 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 9: 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			**Type: …**
Variable	**Type**	**Value / Range**	**Description**
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
Structure Information			
…	…		User Data Specific To The Command

Table 10: 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 11: 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 (Version 2)

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

Common Status			
0x0028	UINT32	ulErrorCount	Error Count Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT8	bErrorLogInd	Number of available Log Entries Not supported yet (see page 34)
0x002D	UINT8	bErrorPDInCnt	Number of input process data handshake errors
0x002E	UINT8	bErrorPDOutCnt	Number of output process data handshake errors
0x002F	UINT8	bErrorSyncCnt	Number of synchronization handshake errors Not supported yet
0x0030	UINT8	bSyncHskMode	Synchronization Handshake Mode Not supported yet
0x0031	UIN8	bSyncSource	Synchronization Source (see page 35)
0x0032	UINT16	ausReserved[3]	Reserved Set to 0

Table 12: Common Status Structure Definition

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCKtag
{
  UINT32  ulCommunicationCOS;
  UINT32  ulCommunicationState;
  UINT32  ulCommunicationError;
  UINT16  usVersion;
  UINT16  usWatchdogTime;
  UINT8   bPDInHskMode;
  UINT8   bPDInSource;
  UINT8   bPDOutHskMode;
  UINT8   bPDOutSource;
  UINT32  ulHostWatchdog;
  UINT32  ulErrorCount;
  UINT8   bErrorLogInd;
  UINT8   bErrorPDInCnt;
  UINT8   bErrorPDOutCnt;
  UINT8   bErrorSyncCnt;
  UINT8   bSyncHskMode;
  UINT8   bSyncSource;
  UINT16  ausReserved[3];
  union
  {
    NETX_MASTER_STATUS  tMasterStatusBlock;  /* for master implementation    */
    UINT32              aulReserved[6];      /* otherwise reserved           */
  } uStackDepended;
} NETX_COMMON_STATUS_BLOCK;
```

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 n*etX 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 13: Communication State of Change

Communication Change of State Flags (netX System ⇨ Application)

Bit	Definition / Description
0	Ready (RCX_COMM_COS_READY) 0 - … 1 - The *Ready* 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 *Running* flag is set, too.
1	Running (RCX_COMM_COS_RUN) 0 - … 1 -The *Running* flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the *Ready* flag and the *Running* flag are set.
2	Bus On (RCX_COMM_COS_BUS_ON) 0 - … 1 -The *Bus On* 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 *Configuration Locked* 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 *Lock Configuration* 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 *Configuration New* 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 *Restart Required* flag.
5	Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - … 1 -The *Restart Required* flag is set when the channel firmware requests to be restarted. This flag is used together with the *Restart Required Enable* 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 *Restart Required Enable* flag is used together with the *Restart Required* flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the *Enable* mechanism see section 2.3.2 of the netX DPM Interface Manual)).
7 … 31	Reserved, set to 0

Table 14: Meaning of Communication Change of State Flags

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 0x0002

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)

Not supported yet: 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.

Number of Input Process Data Handshake Errors

TBD

Number of Output Process Data Handshake Errors

TBD

Number of Synchronization Handshake Errors

This counter will be incremented if the device detects a not handled synchronization indication. This field is not supported yet.

Synchronization Status

This field is reserved for future use.

3.3.1.2 Master Implementation

In addition to the common status block as outlined in the previous section, a master firmware maintains the following structure.

Master Status Structure Definition

```
typedef struct NETX_MASTER_STATUS_Ttag
{
  UINT32 ulSlaveState;
  UINT32 ulSlaveErrLogInd;
  UINT32 ulNumOfConfigSlaves;
  UINT32 ulNumOfActiveSlaves;
  UINT32 ulNumOfDiagSlaves;
  UINT32 ulReserved;
} NETX_MASTER_STATUS_T;
```

Master Status			
Offset	**Type**	**Name**	**Description**
0x0010	Structure	See common structure in table *Common Status Block*	
0x0038	UINT32	`ulSlaveState`	Slave State OK, FAILED (At Least One Slave)
0x003C	UINT32	`ulSlaveErrLogInd`	Slave Error Log Indicator Slave Diagnosis Data Available: EMPTY, AVAILABLE
0x0040	UINT32	`ulNumOfConfigSlaves`	Configured Slaves Number of Configured Slaves On The Network
0x0044	UINT32	`ulNumOfActiveSlaves`	Active Slaves Number of Slaves Running Without Problems
0x0048	UINT32	`ulNumOfDiagSlaves`	Faulted Slaves Number of Slaves Reporting Diagnostic Issues
0x004C	UINT32	`ulReserved`	Reserved Set to 0

Table 15: Master Status Structure Definition

Slave State

The slave state field is available for master implementations only. It indicates whether the master is in cyclic data exchange to all configured slaves. In case there is at least one slave missing or if the slave has a diagnostic request pending, the status is set to *FAILED*. For protocols that support non-cyclic communication only, the slave state is set to *OK* as soon as a valid configuration is found.

Status and Error Codes		
Code (Symbolic Constant)	**Numerical Value**	**Meaning**
RCX_SLAVE_STATE_UNDEFINED	0x00000000	UNDEFINED
RCX_SLAVE_STATE_OK	0x00000001	OK
RCX_SLAVE_STATE_FAILED	0x00000002	FAILED (at least one slave)
Others are reserved		

Table 16: Status and Error Codes

Slave Error Log Indicator

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.

Note: This function is not yet supported.

Number of Configured Slaves

The firmware maintains a list of slaves to which the master has to open a connection. This list is derived from the configuration database created by SYCON.net (see section 6.1 of the netX Dual-Port Memory Manual). This field holds the number of configured slaves.

Number of Active Slaves

The firmware maintains a list of slaves to which the master has successfully opened a connection.

Ideally, the number of active slaves is equal to the number of configured slaves. For certain field bus systems it could be possible that the slave is shown as activated, but still has a problem in terms of a diagnostic issue. This field holds the number of active slaves.

Number of Faulted Slaves

If a slave encounters a problem, it can provide an indication of the new situation to the master in certain field bus systems. As long as those indications are pending and not serviced, the field holds a value unequal zero. If no more diagnostic information is pending, the field is set to zero.

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.

netX Extended Status Field Definition Structure

```
typedef struct NETX_EXTENDED_STATE_FIELD_DEFINITION_Ttag
{
  UINT8    abExtendedStatus[172];    /* Default, protocol specific inform. area */
  NETX_EXTENDED_STATE_FIELD_T tExtStateField;  /* Extended status structures */
} NETX_EXTENDED_STATE_FIELD_DEFINITION_T;
```

Extended Status Block			
Offset	**Type**	**Name**	**Description**
0x0050	`UINT8[]` (the first 64 bytes correspond to `CANOPEN_MASTER_GLOBAL_STATE_T`)	`abExtendedStatus[172]`	Area containing CANopen-related information. See *Table 18: Extended Status Block for CANopen Master* below (first 64 bytes of `abExtendedStatus`)
0x00DB	`UINT8[]`		Reserved area, currently unused, (rest of `abExtendedStatus`)
0x00FC	`Structure` `NETX_EXTENDED_STATE_FIELD_T`	`tExtStateField`	Structure to define Status Fields and their Properties. Status type and properties are specific to protocol implementation

Table 17: Extended Status Block

Note: Each offset is always related to the begin of correspondent channel start.

The definition of the first structure remains specific to correspondent protocol and contains additional information about network status (i.e. flags, error counters, events etc.).

The second structure begins at offset 0x00FC and provides the definition of the up to 32 independent State Fields. These state fields can be defined to represent a kind of bit-list, byte-list etc. with up to 65535 entities. In this way a common access mechanism for different state definitions and quantities can be provided independent of protocol implementation.

The Extended Status Block for CANopen Master is structured as follows:

At address **0x0050**, the following data structure is stored:

Extended Status Block Structure

```
typedef struct CANOPEN_MASTER_EXTENDED_STATE_Ttag
  CANOPEN_MASTER_EXTENDED_STATE_T;

struct CANOPEN_MASTER_EXTENDED_STATE_Ttag
{
  CANOPEN_MASTER_GLOBAL_STATE_T    tGlobalState;
  CANOPEN_MASTER_ADDITIONAL_INFO_T tAdditionalInfo;
};
```

The Extended Status Block of CANopen is built of 2 parts, the Global Status Block and the Additional Info Block:

Extended Status Block for CANopen Master		
Type	**Name**	**Description**
CANOPEN_MASTER_GLOBAL_STATE_T	tGlobalState	Global status block
CANOPEN_MASTER_ADDITIONAL_INFO_T	tAdditionalInfo	Additional information concerning the CANopen master system

Table 18: Extended Status Block for CANopen Master

3.3.2.1 The Global Status Block

Global Status Block for CANopen Master			
Offset	**Type**	**Name**	**Description**
0x50	unsigned char	bGlobalBits	Global error bits
0x51	unsigned char	bCanState	Main state of the CANopen Master system
0x52	unsigned char	bErrorNodeAddress	Unused
0x53	unsigned char	bErrorEvent	Unused
0x54	unsigned short	usBusErrorCount	Number of detected bus error limit oversteps
0x56	unsigned short	usBusOffCount	Number of CAN-chip reinitializations
0x58	unsigned short	usMsgTimeOut	Number of cancelled CAN messages, because of getting no message acknowledging partner
0x5A	unsigned short	usRxOverFlow	Number of indicated Receive Message Overflow coming from the CAN chip
0x5C	unsigned char[ ]	abGlobalError[4]	Reserved error variables
0x60	unsigned char[ ]	abListProjectedNodes[16]	List of projected nodes
0x70	unsigned char[ ]	abListActivatedNodes[16]	List of activated nodes
0x80	unsigned char[ ]	abListDiagnosticNodes[16]	List of diagnostic nodes

Table 19: Global Status Block

- The bGlobalBits parameter

  The bGlobalBits parameter contains a bit field to explain details of bus and master main errors.

The single bits of the bGlobalBits parameter have the following meaning:

bGlobalBits parameter			
Bit	**Short name**	**Name**	**Description**
D7	MUL		Reserved
D6	TOUT	TIMEOUT-ERROR	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 number of detected timeouts is fixed in the Msg_Time_Out variable. The bit will be set when the first timeout was detected and will not be deleted any more.
D5	NRDY	HOST-NOT-READY-NOTIFICATION	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.
D4	EVE	EVENT-ERROR	The DEVICE has detected transmission errors. The number of detected events are fixed in the Bus_Error_Cnt and Bus Off Cnt variables. The bit will be set when the first event was detected and will not be deleted any more.
D3	FAT		Reserved
D2	NEXC	NON-EXCHANGE-ERROR	At least one node has not reached the data exchange state and no process data are exchange with it.
D1	ACLR	AUTO-CLEAR-ERROR	Device stopped the communication to all nodes and reached the auto-clear end state
D0	CTRL	CONTROL-ERROR	parameterization error or severe run time error

Table 20: The bGlobalBits Parameter

- The bCanState parameter

  The bCanState parameter indicates the operational state of the CANopen master.

bCanState parameter	CANopen Master Status
0x00	OFFLINE
0x40	STOP
0x80	CLEAR
0xC0	OPERATE

Table 21: CAN State

- The bErrorNodeAddress parameter

  The bErrorNodeAddress parameter is unused.

- The bErrorEvent parameter

  The bErrorEvent parameter is unused.

- The usBusErrorCount parameter

  The usBusErrorCount parameter is incremented if the error counter of the CAN chip has reached the microcontroller warning limit because of bus errors.

- The usBusOffCount parameter

This variable is incremented if the CAN chip reports that he is no longer involved in bus activities because of overstepped internal bus error counters and must be reinitialized.

- The `usMsgTimeOut` parameter

  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.

- The `usRxOverFlow` parameter

  If the firmware itself is not fast enough for receiving all CAN message coming from the network in time, this overflow counter will be incremented every time a message is lost.

- The `abGlobalError` parameter

  This parameter is reserved.

- The `abListProjectedNodes` parameter

  The `abListProjectedNodes` parameter represents a field of 16 bytes containing the parameterization state of each node station. The following table shows, which bit is related to which node station address:

Offset / Bit	D7	D6	D5	D4	D3	D2	D1	D0
0x60	7	6	5	4	3	2	1	0
0x61	15	14	13	12	11	10	9	8
0x62	23	22	21	20	19	18	17	16
…								
0x6F	127	126	125	124	123	122	121	120

Table 22: Table explaining the Relation between Node Address and the `abListProjectedNodes` Bit

The If the `abListProjectedNodes` bit of the corresponding node is logically

'1', the node is configured in the master, and serviced in its states.

'0', the node is not configured in the master.

- The `abListActivatedNodes` parameter

  The `abListActivatedNodes` parameter represents a field of 16 bytes and contains the state of each node station. The following table shows, which bit is related to which node address:

Offset / Bit	D7	D6	D5	D4	D3	D2	D1	D0
0x70	7	6	5	4	3	2	1	0
0x71	15	14	13	12	11	10	9	8
0x72	23	22	21	20	19	18	17	16
…								
0x7F	127	126	125	124	123	122	121	120

Table 23: Table explaining the Relation between Node Address and the `abListActivatedNodes` Bit

If the abListActivatedNodes bit of the corresponding node is logically

'1', node is operating, node guarding reports no error.

'0', node is not operating, because it is not configured or has an error.

The values in variable abListActivatedNodes are only valid, if the master runs the main state OPERATE.

- The abListDiagnosticNodes parameter

  The abListDiagnosticNodes parameter represents a field of 16 bytes containing the diagnostic bit of each node. The following table shows the relationship between the node address and the corresponding bit in the variable abListDiagnosticNodes.

Offset / Bit	D7	D6	D5	D4	D3	D2	D1	D0
0x80	7	6	5	4	3	2	1	0
0x81	15	14	13	12	11	10	9	8
0x82	23	22	21	20	19	18	17	16
…								
0x8F	127	126	125	124	123	122	121	120

Table 24: Table explaining the Relationship between Node Address and the abListDiagnosticNodes Bit

If the abListDiagnosticNodes bit of the corresponding node is logically

'1', newly received emergency message are available in the internal diagnostic buffer or one of the diagnostics bit of the node has changed. This data can be read out by the host with a message, which is described in section 5.2.7 “CANOPEN_MASTER_GET_NODE_DIAG_REQ/CNF - Get Node Diagnostic”.

'0', since the last diagnostic buffer read access of the host, no values were changed in this buffer.

The values in variable abListDiagnosticNodes are only valid, if the master is in OPERATE state.

The following relationship is valid between the abListActivatedNodes bit and the abListDiagnosticNodes bit:

	abListActivatedNodes = 0	**abListActivatedNodes = 1**
abListDiagnosticNodes = 0	- node not in operation, no Data IO Exchange between master and node - Perhaps this slave is not configured	- node is present on the bus, node guarding active - PDO exchange between master and node happens as configured
abListDiagnosticNodes = 1	- node is not operating, node guarding failed - The master holds newly received diagnostic data in the internal diagnostic buffer	- node is present on the bus, node guarding or heartbeat is active, PDO exchange - The master holds newly received diagnostic data in the internal diagnostic buffer

Table 25: Relationship between the abListDiagnosticNodes bit and the abListDiagnosticNodes bit

3.3.2.2 Additional Info Block

Additional Info Block for CANopen Master			
Offset	**Type**	**Name**	**Description**
0x90	`unsigned long`	`ulFlags`	Bit field for Flags
0x94	`unsigned long`	`ulReserved`	Reserved area
0x98	`unsigned long`	`ulLastDiagAddress`	Last diagnostic address Address that caused last diagnostic entry
0x9C	`unsigned long`	`ulLastDiagInfo`	Last diagnostic info
0xA0	`unsigned long`	`ulBusOffEveCnt`	Counter for bus off events
0xA4	`unsigned long`	`ulErrorPassiveEveCnt`	Counter for passive event errors
0xA8	`unsigned long`	`ulRxOverflowCnt`	Counter for receive overflows
0xAC	`unsigned long`	`ulTxOverflowCnt`	Counter for transmit overflows
0xB0	`unsigned long[ ]`	`aulReserved[8]`	Reserved area
0xD0	`unsigned long`	`ulMaxRecvIdx`	Maximum Object Index Value for Receive Data Number of highest PDO mapped receive object index
0xD4	`unsigned long`	`ulMaxSendIdx`	Maximum Object Index Value for Send Data Number of highest PDO mapped send object index
0xD8	`unsigned long[ ]`	`aulAddDetail[3]`	Additional detail for diagnostic entry

Table 26: Additional Info Block

ulFlags

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

Additional Info Flags		
Bit	**Name**	**Description**
D9.. D31	Reserved	Reserved for further use
D8	CANOPEN_MASTER_ADD_INFO_ FLAG_WDG	Watchdog error detected
D6.. D7	Reserved	Reserved for further use
D5	CANOPEN_MASTER_ADD_INFO _FLAG_TX_OVERFLOW	Transmit overflow detected
D4	CANOPEN_MASTER_ADD_INFO _FLAG_RX_OVERFLOW	Receive overflow detected
D3	CANOPEN_MASTER_ADD_INFO _FLAG_BUS_OFF	CAN is in Bus-off state
D2	CANOPEN_MASTER_ADD_INFO _FLAG_PASSIVE	CAN is in error passive state
D1	CANOPEN_MASTER_ADD_INFO_ FLAG_CAN_ACTIVE	CAN is activated
D0	CANOPEN_MASTER_ADD_INFO_ FLAG_CAN_INIT	CAN is initialized

Table 27: Additional Info Flags

For the CANopen Master protocol implementation, the Extended Status Area is structured as follows:

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_GLOBAL_STATE_Ttag */
typedef struct CANOPEN_MASTER_GLOBAL_STATE_Ttag
  CANOPEN_MASTER_GLOBAL_STATE_T;

#define CANOPEN_MASTER_GLOBAL_STATE_ERROR_SIZE 4
#define CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE   16

#define CANOPEN_MASTER_GLOBAL_DIAG      0x000000ffL

struct CANOPEN_MASTER_GLOBAL_STATE_Ttag
{
  TLR_UINT8  bGlobalBits;
  TLR_UINT8  bCanState;
  TLR_UINT8  bErrorNodeAddress;
  TLR_UINT8  bErrorEvent;
  TLR_UINT16 usBusErrorCount;
  TLR_UINT16 usBusOffCount;
  TLR_UINT16 usMsgTimeOut;
  TLR_UINT16 usRxOverFlow;
  TLR_UINT8  abGlobalError[CANOPEN_MASTER_GLOBAL_STATE_ERROR_SIZE];
  TLR_UINT8  abListProjectedNodes[CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE];
  TLR_UINT8  abListActivatedNodes[CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE];
  TLR_UINT8  abListDiagnosticNodes[CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE];

};

/*****************************************************************************/
/** type of CANOPEN_MASTER_ADDITIONAL_INFO_Ttag */
typedef struct CANOPEN_MASTER_ADDITIONAL_INFO_Ttag
  CANOPEN_MASTER_ADDITIONAL_INFO_T;

#define CANOPEN_MASTER_ADD_INFO_FLAG_CAN_INIT    0x00000001L
#define CANOPEN_MASTER_ADD_INFO_FLAG_CAN_ACTIVE  0x00000002L
#define CANOPEN_MASTER_ADD_INFO_FLAG_PASSIVE     0x00000004L
#define CANOPEN_MASTER_ADD_INFO_FLAG_BUS_OFF     0x00000008L

#define CANOPEN_MASTER_ADD_INFO_FLAG_RX_OVERFLOW 0x00000010L
#define CANOPEN_MASTER_ADD_INFO_FLAG_TX_OVERFLOW 0x00000020L

#define CANOPEN_MASTER_ADD_INFO_FLAG_WDG         0x00000100L

#define CANOPEN_MASTER_ADD_DETAIL_SIZE           0x00000003L

struct CANOPEN_MASTER_ADDITIONAL_INFO_Ttag
{
  TLR_UINT32 ulFlags;
  TLR_UINT32 ulReserved;
  TLR_UINT32 ulLastDiagAddress;
  TLR_UINT32 ulLastDiagInfo;
  TLR_UINT32 ulBusOffEveCnt;
  TLR_UINT32 ulErrorPassiveEveCnt;
  TLR_UINT32 ulRxOverflowCnt;
  TLR_UINT32 ulTxOverflowCnt;
  TLR_UINT32 aulReserved[8];
  TLR_UINT32 ulMaxRecvIdx;
  TLR_UINT32 ulMaxSendIdx;
  TLR_UINT32 aulAddDetail[CANOPEN_MASTER_ADD_DETAIL_SIZE];
};
```

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_EXTENDED_STATE_Ttag */
typedef struct CANOPEN_MASTER_EXTENDED_STATE_Ttag
  CANOPEN_MASTER_EXTENDED_STATE_T;

struct CANOPEN_MASTER_EXTENDED_STATE_Ttag
{
  CANOPEN_MASTER_GLOBAL_STATE_T    tGlobalState;
  CANOPEN_MASTER_ADDITIONAL_INFO_T tAdditionalInfo;

};
```

Besides the global state flags and error counters additional status bit lists of slaves are defined in this structure. These bit lists contain the current state information of all slave devices the master communicates with (i.e. 16bytes = 128 devices). Despite the fact that the implementation of extended status block is protocol specific, the place and definition of these bit lists are to a greater or lesser extent similar for all Hilscher Fieldbus Master protocol stacks. The layout of this block is still maintained with actual specification and will be supported further. The example below shows a generic way to define the corresponding location of the bit lists located at the offsets 0x60, 0x70 and 0x80 (see Table above). Three state structures are needed to be defined to locate such bit lists i.e. inside of input data block.

Extended Status Block for CANopen-Master – Second part (State Field Definition Block)			
Offset	**Type**	**Name**	**Description/Value**
0x00FC	`unsigned char`	`bReserved[3]`	Reserved. Do not use.
0x00FF	`unsigned char`	`bNumStateStructs`	Number of State Structures defined below = 3
↓	**`NETX_EXTENDED_STATE_STRUCT_T`**	**`atStateStruct[0]`**	**Structure to define State field properties**
0x0100	`unsigned char`	`bStateArea`	=0. State field is located in standard input area of channel 0
	`unsigned char`	`bStateTypeID`	=1. Corresponds to a bit list (one bit per node) of configured nodes
	`unsigned short`	`usNumOfStateEntries`	=128. Corresponds to 128 bits, each representing a slave
	`unsigned long`	`ulStateOffset`	Contains an offset pointer to a state field inside input data area 0, which contains the list of projected nodes. See description of the list of projected nodes area above.
↓	**`NETX_EXTENDED_STATE_STRUCT_T`**	**`atStateStruct[1]`**	**Structure to define State field properties**
0x0108	`unsigned char`	`bStateArea`	=0. State field is located in standard input area of channel 0
	`unsigned char`	`bStateTypeID`	=2. Corresponds to a bit list (one bit per node) of active nodes
	`unsigned short`	`usNumOfStateEntries`	=128. Corresponds to 128 bits, each representing a slave
	`unsigned long`	`ulStateOffset`	Contains an offset pointer to a state field inside input data area 0, which contains the list of activated nodes. See description of the list of activated nodes area above.
↓	**`NETX_EXTENDED_STATE_STRUCT_T`**	**`atStateStruct[2]`**	**Structure to define State field properties**
0x0110	`unsigned char`	`bStateArea`	=0. State field is located in standard input area of channel 0
	`unsigned char`	`bStateTypeID`	=3. Corresponds to a bit list (one bit per node) of diagnostic nodes
	`unsigned short`	`usNumOfStateEntries`	=128. Corresponds to 128 bits, each representing a slave
	`unsigned long`	`ulStateOffset`	Contains an offset pointer to a state field inside input data area 0, which contains the list of diagnostic nodes. See description of the list of diagnostic nodes area above.

Table 28: Extended Status Block for CANopen-Master – Second part (State Field Definition Block definition of the bit list state fields for CANopen Master).

If the location of the state fields is defined to be inside of input data area 0 block (as it is shown in generic example above), the corresponding bit lists will be updated by the stack consistently to the data in this area. Moreover, the data and corresponding state fields can be read out by the host application as one data block i.e. with DMA support.

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 29: 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 Getting started / Configuration

This section explains some essential information you should know when starting to work with the CANopen Master Protocol Interface.

4.1 Overview about Essential Functionality

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

Topic	Section Number	Section Name
Process (PDO) data transfer (Input/Output)	5.2.1	CANOPEN_MASTER_REGISTER_REQ/CNF – Register Application
Set NMT State of Nodes	5.2.13	CANOPEN_MASTER_NODE_NMT_COMMAND_REQ/CNF – Set NMT State
Emergency Handling	5.2.12	CANOPEN_MASTER_SEND_EMCY_REQ/CNF – Send Emergency Message
Acyclic data transfer (SDO)	5.2.10 5.2.11	CANOPEN_MASTER_SDO_UPLOAD_REQ/CNF – SDO Upload CANOPEN_MASTER_SDO_DOWNLOAD_REQ/CNF – SDO Download

Table 30: Overview about essential Functionality (Cyclic and acyclic Data Transfer and Alarm Handling).

4.2 Configuration of the CANopen Master

4.2.1 Using the Packet Interface with *write* Access to the Dual-Port Memory

You can configure the CANopen Master firmware using the packet interface described in this document. You need access via DPM to the protocol stack. In order to configure the devices and to supply them with parameters, first the "CANOPEN_MASTER_SET_NODEPARAM_REQ/CNF – Set Node Parameters" request has to be sent to the protocol stack. More information about this topic can be obtained at section 5.2.6 of this manual.

Subsequently, a "CANOPEN_MASTER_SET_BUSPARAM_REQ/CNF – Set Bus Parameters" request needs to be sent to the protocol stack after all node parameters have been set. For more information how to accomplish this, please refer to section 5.2.5 of this manual.

4.2.2 Using the Configuration Tool SYCON.net

The easiest way to configure the CANopen Master is using Hilscher's configuration tool SYCON.net.

SYCON.net creates a binary configuration file, which can be downloaded into the netX running the CANopen Master firmware. The configuration tool is described in an own manual.

4.3 Task Structure of the CANopen Master Stack

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

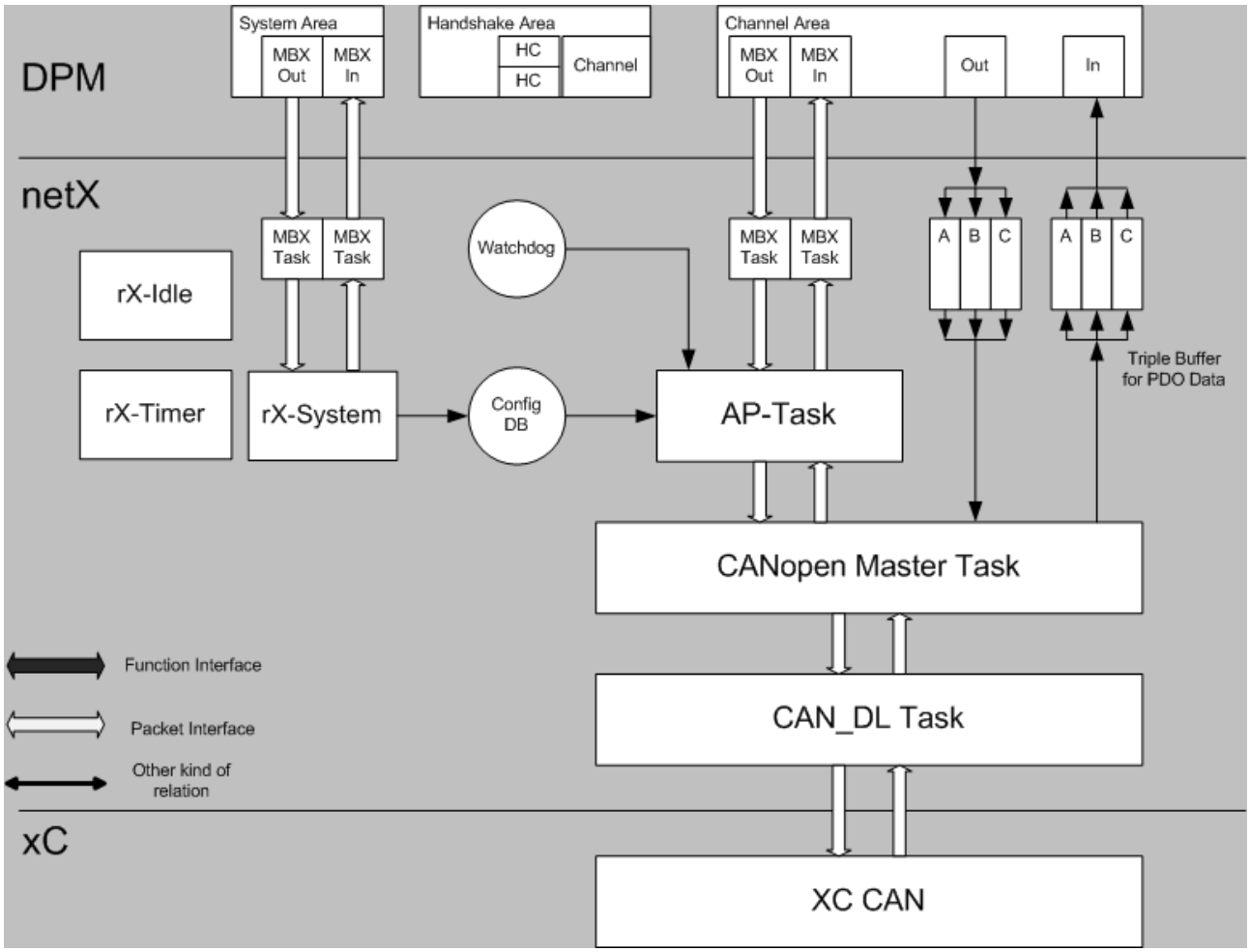

Figure 6: Internal Structure of CANopen Master Firmware

For the explanation of the different kinds of arrows see lower left corner of figure.

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 Master 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 Dual Port Memory interface of the communication channel. In detail, it is responsible for the following:

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

CANopen Master Task

The CANopen Master Task is the center part of CANopen Master 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 Master implementation provides 28 objects for send data and 28 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 master task provides 28 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 Master 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_MASTER_EXCHANGE_DATA_REQ/CNF – Exchange Data” 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
..	..	..
3582	221Bh	7Fh
3583	221Bh	80h

Table 31: 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
..	..	..
3582	201Bh	7Fh
3583	201Bh	80h

Table 32: Mapping of Output Data

4.4.3 Obtaining Diagnostic Information from connected Slaves by sending an RCX_GET_SLAVE_CONN_INFO_REQ Packet

An application which is based on Hilscher's DPM for netX can obtain diagnostic and status information about all slaves connected to and administered by this master from the master firmware as described in general in the netX DPM Manual (Ref. 2). (This information only concerns cyclic data transfer.) The CANopen Master firmware supports this feature.

The netX operating system rcX uses handles in order to access at the slaves. This is done in a possibly unexpected way as these handles are:

- not equal to IP address
- not equal to the connection number

Retrieving the diagnostic information is a two-step-process as you first retrieve the handle using the *Get Slave Handle* request and subsequently you retrieve the diagnostic information using the handle.

1. Retrieve the handle by the *Get Slave Handle* request (RCX_GET_SLAVE_HANDLE_REQ, Command code 0x2F08) which is described in the netX DPM Manual (Ref. 2), chapter 5.2.2.1. In order to do so, you need to choose which kind of list of the above mentioned slave lists you want to obtain. The confirmation packet you will receive (RCX_GET_SLAVE_HANDLE_CNF, Command code 0x2F09) will then deliver an array of handles to the elements of the selected list.
2. This allows to obtain a diagnosis structure for the specific slave by the *Get Slave Connection Information* request (RCX_GET_SLAVE_CONN_INFO_REQ, Command code 0x2F0A). This packet requires the handle of the specific slave taken from the array of handles to the elements of the selected list obtained in the first step. You will then receive the confirmation packet (RCX_GET_SLAVE_CONN_INFO_CNF, Command code 0x2F0B) delivering – besides others – a structure tState containing the following structure with information about the selected slave from the master firmware (see reference 2 for more details).

   For CANopen, this structure looks as follows:

```
/*****************************************************************************/

typedef struct CANOPEN_MASTER_NODE_DIAG_Ttag CANOPEN_MASTER_NODE_DIAG_T;

#define CANOPEN_MASTER_ND_FLAG_SDO_TIMEOUT        0x00000001L
#define CANOPEN_MASTER_ND_FLAG_SDO_ERROR          0x00000002L
#define CANOPEN_MASTER_ND_FLAG_CFG_FAULT          0x00000004L
#define CANOPEN_MASTER_ND_FLAG_HEARTBEAT_STARTED  0x00000008L

#define CANOPEN_MASTER_ND_FLAG_GUARD_ERROR        0x00000010L
#define CANOPEN_MASTER_ND_FLAG_CON_LOST           0x00000020L
#define CANOPEN_MASTER_ND_FLAG_HEARTBEAT_ERROR    0x00000040L
#define CANOPEN_MASTER_ND_FLAG_UNEXPECTED_STATE   0x00000080L

#define CANOPEN_MASTER_ND_FLAG_EMCY_RECEIVED      0x00000100L
#define CANOPEN_MASTER_ND_FLAG_EMCY_BUFF_OVER     0x00000200L
#define CANOPEN_MASTER_ND_FLAG_BOOTUP             0x00000400L
#define CANOPEN_MASTER_ND_FLAG_UNEXPECTED_BOOTUP  0x00000800L

#define CANOPEN_MASTER_ND_FLAG_INVALID_PARAMETER  0x00001000L

#define CANOPEN_MASTER_ND_FLAG_STATE_NOT_HANDLED  0x40000000L
#define CANOPEN_MASTER_ND_FLAG_DEACTIVATED        0x80000000L

#define CANOPEN_MASTER_NODE_NMT_STATE_UNKNOWN           0x00000000L
```

```
#define CANOPEN_MASTER_NODE_NMT_STATE_INITIALISING      0x00000001L
#define CANOPEN_MASTER_NODE_NMT_STATE_STOPPED           0x00000002L
#define CANOPEN_MASTER_NODE_NMT_STATE_OPERATIONAL       0x00000003L
#define CANOPEN_MASTER_NODE_NMT_STATE_PRE_OPERATIONAL   0x00000004L
#define CANOPEN_MASTER_NODE_NMT_STATE_RESET_APPLICATION 0x00000005L
#define CANOPEN_MASTER_NODE_NMT_STATE_RESET_COMM        0x00000006L

struct CANOPEN_MASTER_NODE_DIAG_Ttag
{
  TLR_UINT32    ulNodeFlags;             /* Flags of node (error and info)          */
  TLR_RESULT    eLastDiagInfo;           /* Last diagnostic information             */
  TLR_BOOLEAN32 fDeviceTypeValid;        /* Device Type in information is valid     */
  TLR_UINT32    ulDeviceType;            /* Device Type of node                     */
  TLR_UINT32    ulNmtState;              /* NMT state of node                       */
  TLR_UINT32    ulEmcyCnt;               /* Number of emergency telegrams in buffer */
  CANOPEN_MASTER_EMCY_ENTRY_T atEmcyData[CANOPEN_MASTER_MAX_EMCY_CNT];
  TLR_UINT32    ulAddInfo;
  TLR_UINT32    aulReserved[4];

};
```

This structure contains the following information:

Structure CANOPEN_MASTER_NODE_DIAG_T		
Variable name	**Type**	**Meaning**
ulNodeFlags	UINT32	See below
eLastDiagInfo	TLR_RESULT	Last diagnostic information
fDeviceTypeValid	BOOLEAN32	This Boolean variable indicates whether the device type in the ulDeviceType information is valid or not
ulDeviceType	UINT32	Device type of node
ulNmtState	UINT32	Current internal NMT state of node
ulEmcyCnt	UINT32	Current number of emergency telegrams in buffer
atEmcyData[5]		Data field for emergency entries consisting of 5 blocks each sized 8 bytes
ulAddInfo	UINT32	Additional info
aulReserved[4]	UINT32[ ]	Reserved area

Table 33: CANOPEN_MASTER_NODE_DIAG_T - Node Diagnostic Structure

ulNodeFlags

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

Node Diagnostic Flags		
Bit	**Name**	**Description**
D31	FLAG_DEACTIVATED	Node is deactivated and not handled by the master. This bit does not generate an entry in the diagnostic list.
D30	FLAG_STATE_NOT_HANDLED	At least one state has been omitted during the initialization sequence of the node. This bit does not generate an entry in the diagnostic list.
D13... D29	Reserved	Reserved
D12	FLAG_INVALID_PARAMETER	Parameter set of node is invalid.
D11	FLAG_UNEXPECTED_BOOTUP	Unexpected Boot-up Message from Node received.
D10	FLAG_BOOTUP	Expected Boot-up Message from Node received.
D9	FLAG_EMCY_BUFF_OVER	Emergency buffer overflow
D8	FLAG_EMCY_RECEIVED	Emergency telegram received
D7	FLAG_UNEXPECTED_STATE	Node is in unexpected NMT state
D6	FLAG_HEARTBEAT_ERROR	Error in heartbeat protocol
D5	FLAG_CON_LOST	Node guarding has been lost
D4	FLAG_GUARD_ERROR	A guarding message has been lost. This bit does not generate an entry in the diagnostic list.
D3	FLAG_HEARTBEAT_STARTED	Heartbeat protocol started. This bit does not generate an entry in the diagnostic list.
D2	FLAG_CFG_FAULT	Configuration fault
D1	FLAG_SDO_ERROR	Error during SDO transfer
D0	FLAG_SDO_TIMEOUT	Timeout during SDO transfer

Table 34: Node Diagnostic Flags

eLastDiagInfo

In this information the last diagnostic information of this node station is stored.

fDeviceTypeValid

This variable indicates whether the device type in variable `ulDeviceType` is valid or not.

ulDeviceType

These four bytes are read out from the node while startup. There are several predefined profile numbers existing each described in an own specification manual. Here is an extract of these:

Device Profile for I/O modules: 401 decimal.

Device Profile for Drives and Motion Control: 402 decimal.

Device Profile for Encode: 406 decimal.

ulNmtState

This variable holds the internal NMT state of the node according to the following table:

NMT State	Value
CANOPEN_MASTER_NODE_NMT_STATE_UNKNOWN	0x00000000L
CANOPEN_MASTER_NODE_NMT_STATE_INITIALISING	0x00000001L
CANOPEN_MASTER_NODE_NMT_STATE_STOPPED	0x00000002L
CANOPEN_MASTER_NODE_NMT_STATE_OPERATIONAL	0x00000003L
CANOPEN_MASTER_NODE_NMT_STATE_PRE_OPERATIONAL	0x00000004L
CANOPEN_MASTER_NODE_NMT_STATE_RESET_APPLICATION	0x00000005L
CANOPEN_MASTER_NODE_NMT_STATE_RESET_COMM	0x00000006L

Table 35: Internal NMT State of Node

ulEmcyCnt

This variable contains the number of saved emergency telegrams in the buffer (the data area set up by `atEmcyData[5]` spanning the next following 40 bytes)

atEmcyData[5]

In this area the containments of the emergency messages are saved. Each emergency message contains 8 bytes of data

ulAddInfo

Additional information

aulReserved[4]

This area is reserved for future use.

4.5 Handshake Modes and Synchronization

The CANopen-Master supports the general service RCX_SET_HANDSHAKE_CONFIG_REQ (Defined in header file rcX_Public.h). This request offers the possibility to separately configure handshake modes for input (Parameter bPDInHskMode), output (Parameter bPDOutHskMode) and synchronization (Parameter bSyncHskMode).

It can only be used reasonably in conjunction with a dual-port memory. This provides the host application with the possibility to synchronize with the SYNC event of the CANopen protocol.

The request may not be issued in the following cases:

- The CANopen Master is in state BUS-ON.
- The configuration has been locked

Currently four different modes are supported:

Mode	Data mode	Input handshake mode	Output handshake mode	Sync handshake mode	Remark
1	Buffered	Host controlled	Host controlled	Not used	Standard behavior of the CANopen-Master
2	Buffered	Host controlled	Host controlled	Host triggers SYNC messages	Host triggers sending of SYNC messages via Sync Handshake
3	Buffered	Host controlled	Host controlled	Sync event of CANopen stack used Host is informed via Sync handshake	Host is informed about occurrence of SYNC event via Sync Handshake
4	Buffered	Host controlled	Host controlled	Sync event of CANopen stack used Host is informed via PD Input handshake	Host is informed about occurrence of SYNC event via PD Input Handshake

Table 36: Overview of CANopen Master handshake modes and synchronization

If you do not want to configure the handshake manually using the rcX packet RCX_SET_HANDSHAKE_CONFIG_REQ , there is also the possibility to work with one of the three options defined in *Table 102: Possible values of bSyncTrigger* in section CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ/CNF – Configure SYNC Trigger*:*

- Time-controlled trigger mode (bSyncTrigger=0) corresponds to mode 1
- Time-controlled trigger mode (bSyncTrigger=1) with notification corresponds to modes 3 and 4.
- Application-controlled trigger mode (bSyncTrigger=2) corresponds to mode 2.

4.5.1 Mode 1: Standard Behavior of the CANopen-Master

The SYNC event is sent by the CANopen Master in the configured time interval, no synchronization with the host application takes place.

The configuration is done via the RCX_SET_HANDSHAKE_CONFIG_REQ request packet (see reference [2], section 4.23.1) with the following parameters:

```
bPDInHskMode = 4
bPDInSource = 0
usPDInErrorTh = 0

bPDOutHskMode = 4
bPDOutSource = 0
usPDOutErrorTh = 0

bSyncHskMode = 0
bSyncSource = 0
usSyncErrorTh = 0

aulReserved[2] = {0}
```

4.5.2 Mode 2: Host triggers sending of SYNC messages via Sync Handshake

The SYNC events are initiated by the host application and controlled via the SYNC handshake flags of the dual-port memory. The configured time interval for the SYNC event is not used in this mode, the SYNC handshake flags are in mode *Host-Controlled*.

The configuration is done via the `RCX_SET_HANDSHAKE_CONFIG_REQ` request packet (see reference [2], section 4.23.1) with the following parameters:

```
bPDInHskMode = 4
bPDInSource = 0
usPDInErrorTh = 0

bPDOutHskMode = 4
bPDOutSource = 0
usPDOutErrorTh = 0

bSyncHskMode = 2
bSyncSource = 1
usSyncErrorTh = 0

aulReserved[2] = {0}
```

The SYNC event is initiated by toggling the SYNC handshake flag of the host via the host application. Directly after the SYNC telegram has been handed over to CAN, this is acknowledged by the CANopen Master via the SYNC handshake flag of the device. The process data are controlled via the input and output handshake flags in mode *Host-Controlled* independently from the SYNC handshake.

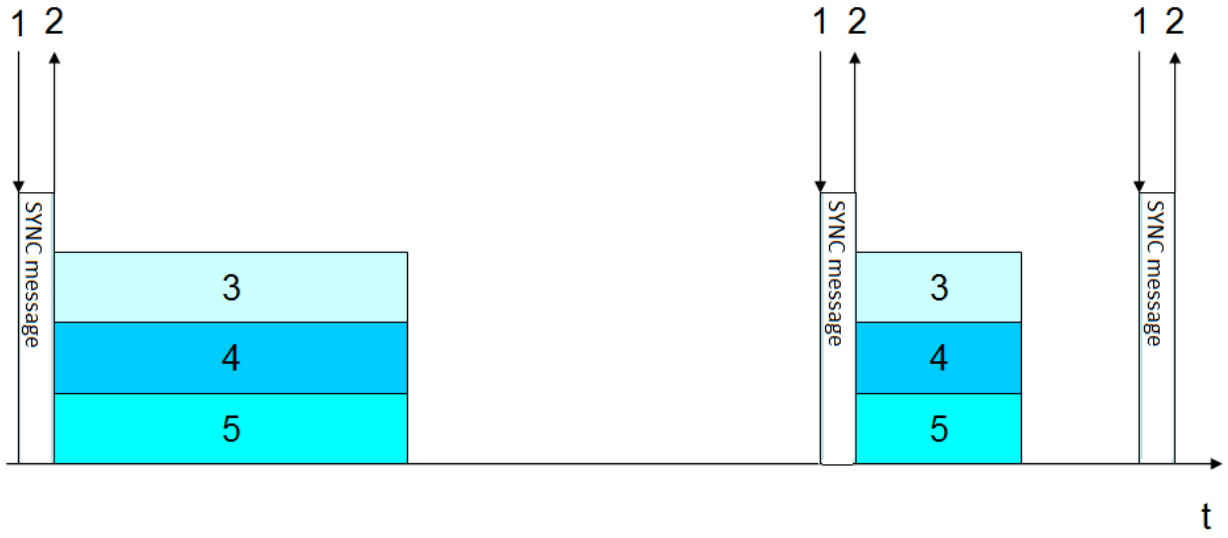

Figure 7: Mode 2: Host triggers sending of SYNC messages via Sync Handshake

1	SYNC handshake from the host application to the CANopen Master
2	Acknowledgement from the SYNC handshake by the CANopen Master
3	Send synchronous TxPDOs
4	Receive synchronous RxPDOs and copy these into a buffer
5	Copy RxPDOs of preceding SYNC event from a buffer to data memory

Table 37: Legend to Figure 7

4.5.3 Mode 3: Host is informed about occurrence of SYNC event via Sync Handshake

The SYNC events take place within the configured time interval and the host application is informed about the occurrence of a SYNC event via the SYNC handshake flags of the dual-port memory. Here, the SYNC handshake flags are in the mode *Device-Controlled*.

The configuration is done via the `RCX_SET_HANDSHAKE_CONFIG_REQ` request packet (see reference [2], section 4.23.1) with the following parameters:

```
bPDInHskMode = 4
bPDInSource = 0
usPDInErrorTh = 0

bPDOutHskMode = 4
bPDOutSource = 0
usPDOutErrorTh = 0

bSyncHskMode = 1
bSyncSource = 1
usSyncErrorTh = 0

aulReserved[2] = {0}
```

By toggling the device SYNC handshake flag the CANopen Master informs the host application about the occurrence of a SYNC event, i.e. the SYNC message and all synchronous PDOs to be sent have been handed over to CAN, and all synchronous PDO received since the last SYNC event have been updated.

The host application should acknowledge the SYNC handshake prior to the next SYNC event by toggling the host SYNC handshake flag. If this does not happen, the error counter `bErrorSyncCnt` within the dual-port memory is incremented. Anyway, there is no influence on the CANopen protocol itself.

The process data are controlled via the input and output handshake flags in mode *Host-Controlled* independently from the SYNC handshake.

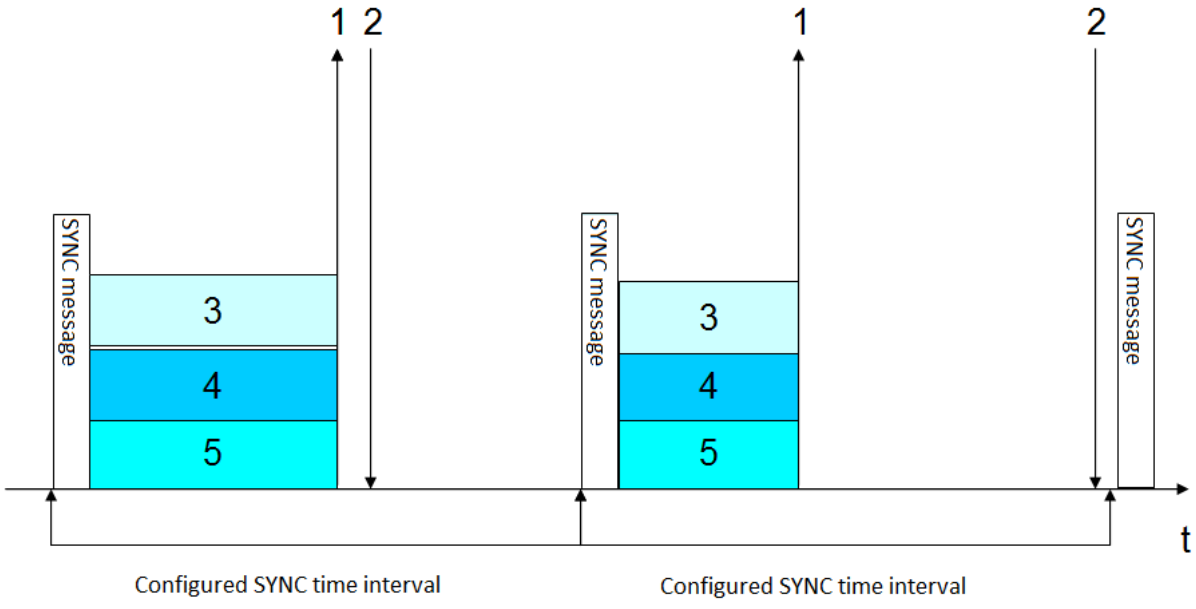

Figure 8: Mode 3: Host is informed about occurrence of SYNC event via Sync Handshake

1	SYNC handshake from the CANopen Master to the host application
2	Acknowledgement from the SYNC handshake by the host application
3	Send synchronous TxPDOs
4	Receive synchronous RxPDOs and copy these into a buffer
5	Copy RxPDOs of preceding SYNC event from a buffer to data memory

Table 38: Legend to Figure 8

4.5.4 Mode 4: Host is informed about occurrence of SYNC event via PD Input Handshake

The SYNC events take place within the configured time interval and the host application is informed about the occurrence of a SYNC event via the input handshake flags of the dual-port memory. The SYNC handshake flags are **not** used in this mode.

The configuration is done via the `RCX_SET_HANDSHAKE_CONFIG_REQ` request packet (see reference [2], section 4.23.1) alternatively with one of the following two parameter sets:

```
bPDInHskMode = 4
bPDInSource = 1
usPDInErrorTh = 0

bPDOutHskMode = 4
bPDOutSource = 0
usPDOutErrorTh = 0

bSyncHskMode = 0
bSyncSource = 0
usSyncErrorTh = 0

aulReserved[2] = {0}

or:

bPDInHskMode = 5
bPDInSource = 0
usPDInErrorTh = 0

bPDOutHskMode = 4
bPDOutSource = 0
usPDOutErrorTh = 0

bSyncHskMode = 0
bSyncSource = 0
usSyncErrorTh = 0

aulReserved[2] = {0}
```

The input and output handshake flags are both in mode *Host-Controlled*. The host application starts the update of the input process data bytoggling the host input handshake flag. However, the CANopen Master toggles the device input handshake flag after the next SYNC event causing the acknowledgement to be delayed until the next SYNC event and all synchronous PDOs to be sent have been handed over to CAN and all synchronous PDOs received since the last SYNC event have been updated. Thus, the host application should start a new read request prior to the next SYNC event by toggling the host input handshake flag once again. (If this does not happen, the error counter `bErrorPDInCnt` within the dual-port memory is incremented. Anyway, there is no influence on the CANopen protocol itself).

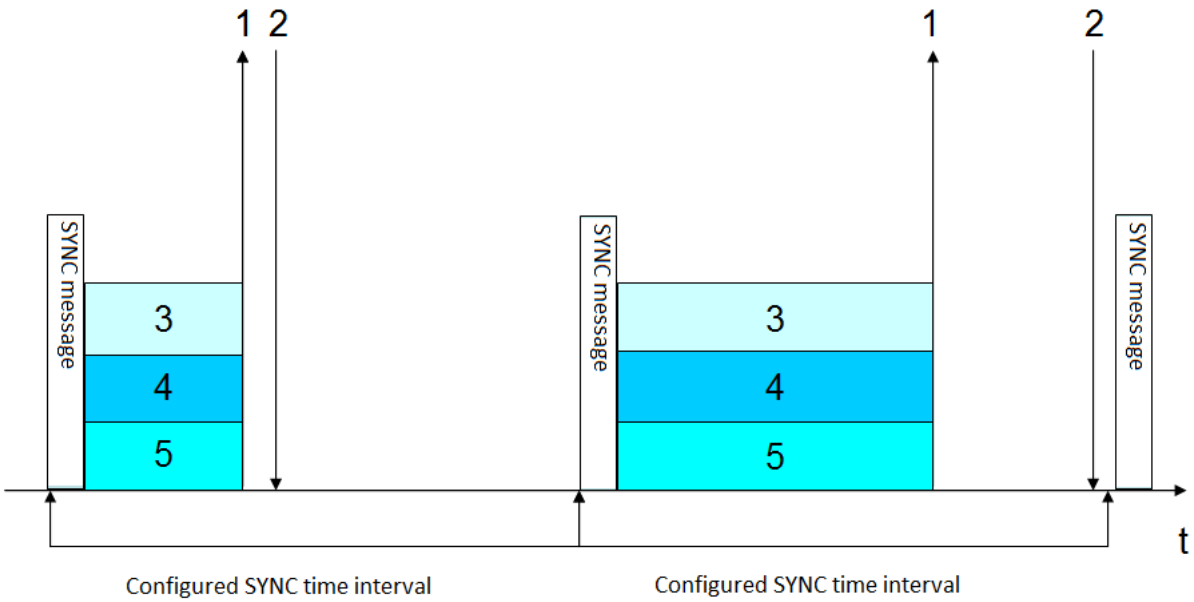

Figure 9; Mode 4: Host is informed about occurrence of SYNC event via PD Input Handshake

1	Input handshake acknowledgement from the CANopen Master to the host application
2	Input handshake from the host application to the CANopen Master
3	Send synchronous TxPDOs
4	Receive synchronous RxPDOs and copy these into a buffer
5	Copy RxPDOs of preceding SYNC event from a buffer to data memory

Table 39: Legend to Figure 9

5 The Application Interface

This chapter defines the application user interface of the CANopen Master 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 Master Stack. Once, the CANopen Master 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 Master Stack queues via packets as well. The AP-Task will not route all commands to the CANopen Master Task. Some commands are used for internal communication between the AP- and CANopen Master-Task only. Other requests are not possible in specific states.

5.1 The CANopen-APM-Task

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

ASCII queue name	Description
"QUE_CANOPENAPM"	Name of the APM-Task process queue

Table 40: APM-Task Process Queue

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

In detail, the following functionality is provided by the `APM` -Task:

Overview over Packets of the APM-Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
5.1.1	`CANOPEN_APM_GET_STATE_REQ/CNF` – Get State of AP-Task	`0x3A00/ 0x3A01`	64
5.1.2	`CANOPEN_APM_WARMSTART_REQ/CNF` – Set Warmstart Parameters	`0x3A02/ 0x3A03`	66
5.1.3	`CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ/CNF` – Enable/Disable PDO Counter	`0x3A04/ 0x3A05`	70

Table 41: Overview over the Packets of the APM-Task of the CANopen Master Protocol Stack

5.1.1 CANOPEN_APM_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_APM_PCK_GET_STATE_REQ_Ttag
  CANOPEN_APM_PCK_GET_STATE_REQ_T;

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

Packet Description

Structure Information CANOPEN_APM_PCK_GET_STATE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENAPM	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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-APM-Task*
ulCmd	UINT32	0x00003A00	CANOPEN_APM_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 42: CANOPEN_APM_PCK_GET_STATE_REQ_T – Get State of AP-Task Request

Packet Structure Reference

```
typedef struct CANOPEN_APM_GET_STATE_CNF_DATA_Ttag
  CANOPEN_APM_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_APM_PCK_GET_STATE_CNF_Ttag
  CANOPEN_APM_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_APM_PCK_GET_STATE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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-APM-Task*
ulCmd	UINT32	0x00003A01	CANOPEN_APM_GET_STATE_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
Structure CANOPEN_APM_GET_STATE_CNF_DATA_T			
ulHighestMappedSendBufferNum	UINT32	0.. 28	Number of the highest PDO mapped send triple buffer.
ulHighestMappedRecvBufferNum	UINT32	0.. 28	Number of the highest PDO mapped receive triple buffer.

Table 43: CANOPEN_APM_PCK_GET_STATE_CNF_T – Get State of AP-Task Confirmation

5.1.2 CANOPEN_APM_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. This request will be denied if the configuration lock flag is set.

Packet Structure Reference

```
typedef struct CANOPEN_APM_WARMSTART_REQ_DATA_Ttag
  CANOPEN_APM_WARMSTART_REQ_DATA_T;

#define CANOPEN_APM_SYS_FLAG_COM_CONTROLLED_RELEASE   0x00000001L

struct CANOPEN_APM_WARMSTART_REQ_DATA_Ttag  /* Warmstart request data */
{
  TLR_UINT32  ulSystemFlags;    /* System flags             */
  TLR_UINT32  ulWdgTime;        /* Watchdog time            */
  TLR_UINT32  aulReserved[7];   /* Reserved for further use */
};

typedef struct CANOPEN_APM_PCK_WARMSTART_REQ_Ttag
  CANOPEN_APM_PCK_WARMSTART_REQ_T;

struct CANOPEN_APM_PCK_WARMSTART_REQ_Ttag   /* Warmstart request */
{
  TLR_PACKET_HEADER_T               tHead;  /** packet header */
  CANOPEN_APM_WARMSTART_REQ_DATA_T  tData;  /** packet data   */
};
```

Packet Description

Structure Information CANOPEN_APM_PCK_WARMSTART_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENAPM	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	36	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-APM-Task*
ulCmd	UINT32	0x00003A02	CANOPEN_APM_WARMSTART_REQ - Command
ulExt	UINT32		Extension not in use, set to zero for compatibility reasons
ulRout	UINT32		Routing, do not touch
Structure CANOPEN_APM_WARMSTART_REQ_DATA_T			
ulSystemFlags	UINT32		System Flags
		0	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
		1	Communication with controller is allowed only with the BUS_ON flag.
			BIT 1 - 31: Reserved for further use, set to zero
ulWdgTime	UINT32		Watchdog supervision
		0 20 ..65535	Watchdog supervision deactivated Watchdog time in milliseconds
aulReserved[7]	UINT32[]		Reserved for further use, set to zero

Table 44: CANOPEN_APM_PCK_WARMSTART_REQ – Set Warmstart Parameter Request

Packet Structure Reference

```
typedef struct CANOPEN_APM_WARMSTART_CNF_DATA_Ttag
  CANOPEN_APM_WARMSTART_CNF_DATA_T;

#define CANOPEN_APM_SYS_FLAG_COM_CONTROLLED_RELEASE   0x00000001L

struct CANOPEN_APM_WARMSTART_CNF_DATA_Ttag  /* Warmstart confirmation data */
{
  TLR_UINT32  ulSystemFlags;    /* System flags                     */
  TLR_UINT32  ulWdgTime;        /* Watchdog time                    */
  TLR_UINT32  aulReserved[7];   /* Reserved for further use         */
};

typedef struct CANOPEN_APM_PCK_WARMSTART_CNF_Ttag
  CANOPEN_APM_PCK_WARMSTART_CNF_T;

struct CANOPEN_APM_PCK_WARMSTART_CNF_Ttag   /* Warmstart confirmation */
{
  TLR_PACKET_HEADER_T               tHead;  /** packet header */
  CANOPEN_APM_WARMSTART_CNF_DATA_T  tData;  /** packet data   */
};
```

Packet Description

Structure Information CANOPEN_APM_PCK_WARMSTART_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	Source End Point Identifier, untouched
ulLen	UINT32	36	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-APM-Task*
ulCmd	UINT32	0x00003A03	CANOPEN_APM_WARMSTART_CNF - Command
ulExt	UINT32		Extension not in use, set to zero for compatibility reasons
ulRout	UINT32		Routing, do not touch
Structure CANOPEN_APM_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 - 31: Reserved for further use, se to zero
ulWdgTime	UINT32	 0 20..65535	Watchdog supervision Watchdog supervision deactivated Watchdog time in milliseconds
aulReserved[7]	UINT32[ ]		Reserved for further use, set to zero

Table 45: CANOPEN_APM_PCK_WARMSTART_CNF_T – Set Warmstart Parameter Confirmation

5.1.3 CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ/CNF – Enable/Disable PDO Counter

This packet allows to enable (`ulMode= CANOPEN_APM_ENABLE_PDO_COUNTER=1`) or to disable (`ulMode= CANOPEN_APM_DISABLE_PDO_COUNTER=0`) the PDO counter.

The confirmation packet additionally returns the offset of the PDO Counter in the DPM.

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_Ttag */
typedef struct CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_Ttag
  CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_T;

#define CANOPEN_APM_DISABLE_PDO_COUNTER 0x00000000L
#define CANOPEN_APM_ENABLE_PDO_COUNTER  0x00000001L

struct CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_Ttag
{
  TLR_UINT32  ulMode;
};

/*****************************************************************************/
/** type of CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_REQ_Ttag */
typedef struct CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_REQ_Ttag
  CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_REQ_T;

struct CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_REQ_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;  /** packet header */
  CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_T  tData;  /** packet data   */

};
```

Packet Description

Structure Information **CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_REQ_T**			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENAPM	Destination Queue-Handle
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulAPMS0Id	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 *6.1 Codes of the CANopen-APM-Task*
ulCmd	UINT32	0x3A04	CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
Structure CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_T			
ulMode	UINT32	0,1	Mode 0: Disable PDO Counter 1: Enable PDO Counter

Table 46: CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_REQ –Enable/Disable PDO Counter Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_Ttag */
typedef struct CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_Ttag
  CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_T;

#define CANOPEN_APM_DISABLE_PDO_COUNTER 0x00000000L
#define CANOPEN_APM_ENABLE_PDO_COUNTER  0x00000001L

struct CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_Ttag
{
  TLR_UINT32  ulMode;
  TLR_UINT32  ulDpmOffset;
};

/*****************************************************************************/
/** type of CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_CNF_Ttag */
typedef struct CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_CNF_Ttag
  CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_CNF_T;
```

Packet Description

Structure Information **CANOPEN_APM_PCK_ENABLE_DISABLE_PDO_COUNTER_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulCANOPENMST0Id	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 *6.1 Codes of the CANopen-APM-Task*
ulCmd	UINT32	0x00003A05	CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
Structure CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_T			
ulMode	UINT32	0,1	Mode 0: Disable PDO Counter 1: Enable PDO Counter
ulDpmOffset	UINT32		Offset of PDO Counter in the DPM

Table 47: CANOPEN_APM_ENABLE_DISABLE_PDO_COUNTER_CNF – Confirmation of Enable/Disable PDO Counter Request

5.2 The CANopen Master-Task

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

ASCII queue name	Description
"QUE_CANOPENMST"	Name of the CANopen Master-Task process queue

Table 48 CANopen Master-Task Process Queue

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

In detail, the following functionality is provided by the CANopen Master-Task:

Overview over Packets of the CANopen Master-Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
5.2.1	`CANOPEN_MASTER_REGISTER_REQ/CNF` – Register Application	`0x2800/ 0x2801`	75
5.2.2	`CANOPEN_MASTER_EXCHANGE_DATA_REQ/CNF` – Exchange Data	`0x280A/ 0x280B`	78
5.2.3	`CANOPEN_MASTER_STARTSTOP_REQ/CNF` – Start/Stop CANopen Network	`0x2802/ 0x2803`	82
5.2.4	`CANOPEN_MASTER_INITIALIZE_REQ/CNF` – Initialization of CANopen Network	`0x2804/ 0x2805`	84
5.2.5	`CANOPEN_MASTER_SET_BUSPARAM_REQ/CNF` – Set Bus Parameters	`0x2806/ 0x2807`	87
5.2.6	`CANOPEN_MASTER_SET_NODEPARAM_REQ/CNF` – Set Node Parameters	`0x2808/ 0x2809`	93
5.2.7	`CANOPEN_MASTER_GET_NODE_DIAG_REQ/CNF` - Get Node Diagnostic	`0x280E/ 0x280F`	104
5.2.8	`CANOPEN_MASTER_GET_BUFFER_HANDLE_REQ/CNF` – Get Buffer Handle	`0x280C/ 0x280D`	107
5.2.9	`CANOPEN_MASTER_STATE_CHANGE_IND/RES` – Change of State Indication	`0x2812/ 0x2813`	110
5.2.10	`CANOPEN_MASTER_SDO_UPLOAD_REQ/CNF` – SDO Upload	`0x2814/ 0x2815`	117
5.2.11	`CANOPEN_MASTER_SDO_DOWNLOAD_REQ/CNF` – SDO Download	`0x2816/ 0x2817`	120
5.2.12	`CANOPEN_MASTER_SEND_EMCY_REQ/CNF` – Send Emergency Message	`0x2818/ 0x2819`	123

Overview over Packets of the CANopen Master-Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
5.2.13	CANOPEN_MASTER_NODE_NMT_COMMAND_REQ/CNF – Set NMT State	0x281A/ 0x281B	127
5.2.14	CANOPEN_MASTER_SET_WATCHDOG_FAIL_REQ/CNF – Set Watchdog Fail	0x28AA/ 0x28AB	130
5.2.15	CANOPEN_MASTER_SLAVE_ACTIVATE_REQ/CNF – Activate Slave	0x28AC/ 0x28AD	132
5.2.16	CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ/CNF – Enable/Disable PDO Counter	0x28AE/ 0x28AF	135
5.2.17	CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ/CNF – Force compare values	0x28B0/ 0x28B1	139
5.2.18	CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ/CNF – Configure SYNC Trigger	0x28B2/ 0x28B3	142
5.2.19	CANOPEN_MASTER_SEND_SYNC_REQ/CNF – Send SYNC	0x28B4/ 0x28B5	145
5.2.20	CANOPEN_MASTER_SYNC_IND/RES – SYNC Indication	0x28B6/ 0x28B7	147
5.2.21	CANOPEN_MASTER_RESET_ERROR_REQ/CNF – Reset Error Event	0x28B8 0x28B9	149

Table 49: Overview over the Packets of the CANopen Master -Task of the CANopen Master Protocol Stack

5.2.1 CANOPEN_MASTER_REGISTER_REQ/CNF – Register Application

This packet is used in order to register to the CANopen Master-Task. After this request is performed successfully, indication packets are sent from the CANopen Master-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 Master task.

Note: This packet will no longer be supported by the firmware described in this document after September 1, 2009.
Instead use the registering functionality described in the netX Dual-Port-Memory Manual instead (RCX_REGISTER_APP_REQ, code 0x2F10).

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_REGISTER_REQ_DATA_Ttag
  CANOPEN_MASTER_REGISTER_REQ_DATA_T;

struct CANOPEN_MASTER_REGISTER_REQ_DATA_Ttag
{
  TLR_UINT32 ulReserved;
};

typedef struct CANOPEN_MASTER_PACKET_REGISTER_REQ_Ttag
  CANOPEN_MASTER_PACKET_REGISTER_REQ_T;

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_REGISTER_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 51: CANOPEN_MASTER_REGISTER_REQ – Packet Status/Error*
ulCmd	UINT32	0x00002800	`CANOPEN_MASTER_REGISTER_REQ` - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure `CANOPEN_MASTER_REGISTER_REQ_DATA_T`			
ulReserved	UINT32		Reserved for further use, set to zero

Table 50: `CANOPEN_MASTER_PACKET_REGISTER_REQ_T` – Register Application Request

Packet Status/Error

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

Table 51: `CANOPEN_MASTER_REGISTER_REQ` – Packet Status/Error

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_REGISTER_CNF_DATA_Ttag
  CANOPEN_MASTER_REGISTER_CNF_DATA_T;

struct CANOPEN_MASTER_REGISTER_CNF_DATA_Ttag
{
  TLR_UINT32 ulReserved;
};

typedef struct CANOPEN_MASTER_PACKET_REGISTER_CNF_Ttag
  CANOPEN_MASTER_PACKET_REGISTER_CNF_T;

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_REGISTER_CNF			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 53: CANOPEN_MASTER_REGISTER_CNF – Packet Status/Error*
ulCmd	UINT32	0x00002801	CANOPEN_MASTER_REGISTER_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_REGISTER_CNF_DATA_T			
ulReserved	UINT32		Reserved for further use, set to zero

Table 52: CANOPEN_MASTER_PACKET_REGISTER_CNF_T – Register Application Confirmation

Packet Status/Error

Definition / (Value)	**Description**
TLR_S_OK (0x00000000)	Status ok
TLR_E_CANOPEN_MASTER_PACKET_LENGTH 0xC0420002)	Invalid length in packet.
TLR_I_CANOPEN_MASTER_INITIALIZE (0x40420056)	Master is initializing.

Table 53: CANOPEN_MASTER_REGISTER_CNF – Packet Status/Error

5.2.2 CANOPEN_MASTER_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_MASTER_EXCHANGE_DATA_REQ_DATA_Ttag
  CANOPEN_MASTER_EXCHANGE_DATA_REQ_DATA_T;

#define CANOPEN_MASTER_RECV_OBJECT_CNT    28

#define CANOPEN_MASTER_MIN_RECV_IDX       0x2200
#define CANOPEN_MASTER_MAX_RECV_IDX       0x221B

#define CANOPEN_MASTER_MIN_RECV_SUB_IDX   1
#define CANOPEN_MASTER_MAX_RECV_SUB_IDX   128

#define CANOPEN_MASTER_SEND_OBJECT_CNT    28

#define CANOPEN_MASTER_MIN_SEND_IDX       0x2000
#define CANOPEN_MASTER_MAX_SEND_IDX       0x201B

#define CANOPEN_MASTER_MIN_SEND_SUB_IDX   1
#define CANOPEN_MASTER_MAX_SEND_SUB_IDX   128

struct CANOPEN_MASTER_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_MASTER_MAX_SEND_SUB_IDX];
};

typedef struct CANOPEN_MASTER_PACKET_EXCHANGE_DATA_REQ_Ttag
  CANOPEN_MASTER_PACKET_EXCHANGE_DATA_REQ_T;

/** Structure of task command exchange data request*/
struct CANOPEN_MASTER_PACKET_EXCHANGE_DATA_REQ_Ttag
{
  TLR_PACKET_HEADER_T                     tHead;  /** packet header.       */
  CANOPEN_MASTER_EXCHANGE_DATA_REQ_DATA_T tData;  /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_EXCHANGE_DATA_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x0000280A	CANOPEN_MASTER_EXCHANGE_DATA_REQ - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_EXCHANGE_DATA_REQ_DATA_T			
ulRecvIndex	UINT32	2200h...221Bh	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...201Bh	Send object index
ulSendSub Index	UINT32	1.. 128	Send object sub-index
ulSendDataCnt	UINT32	0 ...128	Number data byte to be sent
abSendData [128]	UINT8[ ]		Send data

Table 54: CANOPEN_MASTER_PACKET_EXCHANGE_DATA_REQ_T – Exchange Data Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_EXCHANGE_DATA_CNF_DATA_Ttag
  CANOPEN_MASTER_EXCHANGE_DATA_CNF_DATA_T;

#define CANOPEN_MASTER_RECV_OBJECT_CNT    28

#define CANOPEN_MASTER_MIN_RECV_IDX       0x2200
#define CANOPEN_MASTER_MAX_RECV_IDX       0x221B

#define CANOPEN_MASTER_MIN_RECV_SUB_IDX   1
#define CANOPEN_MASTER_MAX_RECV_SUB_IDX   128

#define CANOPEN_MASTER_SEND_OBJECT_CNT    28

#define CANOPEN_MASTER_MIN_SEND_IDX       0x2000
#define CANOPEN_MASTER_MAX_SEND_IDX       0x201B

#define CANOPEN_MASTER_MIN_SEND_SUB_IDX   1
#define CANOPEN_MASTER_MAX_SEND_SUB_IDX   128

struct CANOPEN_MASTER_EXCHANGE_DATA_CNF_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  abRecvData[CANOPEN_MASTER_MAX_RECV_SUB_IDX];
};

typedef struct CANOPEN_MASTER_PACKET_EXCHANGE_DATA_CNF_Ttag
  CANOPEN_MASTER_PACKET_EXCHANGE_DATA_CNF_T;

/** Structure of task command exchange data confirmation */
struct CANOPEN_MASTER_PACKET_EXCHANGE_DATA_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;  /** packet header.            */
  CANOPEN_MASTER_EXCHANGE_DATA_CNF_DATA_T tData;  /** packet confirmation data. */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_EXCHANGE_DATA_CNF			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x0000280B	CANOPEN_MASTER_EXCHANGE_DATA_CNF- Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
structure CANOPEN_MASTER_EXCHANGE_DATA_CNF_DATA_T			
ulRecvIndex	UINT32	2200h...221Bh	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...201Bh	Send object index
ulSendSub Index	UINT32	1.. 128	Send object sub-index
ulSendData Cnt	UINT32	0..128	Number data byte to be sent
abRecvData [128]	UINT8[ ]		Receive data

Table 55: CANOPEN_MASTER_PACKET_EXCHANGE_DATA_CNF_T –Exchange Data Confirmation

5.2.3 CANOPEN_MASTER_STARTSTOP_REQ/CNF – Start/Stop CANopen Network

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

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_STARTSTOP_REQ_DATA_Ttag
  CANOPEN_MASTER_STARTSTOP_REQ_DATA_T;

#define CANOPEN_MASTER_STOP_CANOPEN     0x00000000L
#define CANOPEN_MASTER_START_CANOPEN    0x00000001L

struct CANOPEN_MASTER_STARTSTOP_REQ_DATA_Ttag
{
  TLR_UINT32 ulMode;
};

typedef struct CANOPEN_MASTER_PACKET_STARTSTOP_REQ_Ttag
  CANOPEN_MASTER_PACKET_STARTSTOP_REQ_T;

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_STARTSTOP_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x00002802	`CANOPEN_MASTER_STARTSTOP_REQ` - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_STARTSTOP_REQ_DATA_T			
ulMode	UINT32	0 1	Depending on this assignment, CANopen is either started or stopped: Stop CANopen Start CANopen

Table 56: CANOPEN_MASTER_PACKET_STARTSTOP_REQ_T – Start/Stop CANopen Network Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_STARTSTOP_CNF_DATA_Ttag
  CANOPEN_MASTER_STARTSTOP_CNF_DATA_T;

#define CANOPEN_MASTER_STOP_CANOPEN     0x00000000L
#define CANOPEN_MASTER_START_CANOPEN    0x00000001L

struct CANOPEN_MASTER_STARTSTOP_CNF_DATA_Ttag
{
  TLR_UINT32 ulMode;
};

typedef struct CANOPEN_MASTER_PACKET_STARTSTOP_CNF_Ttag
  CANOPEN_MASTER_PACKET_STARTSTOP_CNF_T;

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_STARTSTOP_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x00002803	CANOPEN_MASTER_STARTSTOP_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
structure CANOPEN_MASTER_STARTSTOP_CNF_DATA_T			
ulMode	UINT32	0 1	Depending on this assignment, CANopen is either started or stopped: Stop CANopen Start CANopen

Table 57: CANOPEN_MASTER_PACKET_STARTSTOP_CNF_T – Start/Stop CANopen Network Confirmation

5.2.4 CANOPEN_MASTER_INITIALIZE_REQ/CNF – Initialization of CANopen Network

This command is used in order to reset the CANopen Master. All configuration data, except warmstart parameters, are deleted. If the CANopen Master is in operational state, the network will be stopped, before the initialize command is processed.

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 Master is configured by configuration database, this configuration is reloaded again after the initialize command is completed.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_INITIALIZE_REQ_Ttag
  CANOPEN_MASTER_PACKET_INITIALIZE_REQ_T;

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

struct CANOPEN_MASTER_INITIALIZE_REQ_DATA_Ttag
{
  TLR_UINT32 ulReserved;
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_INITIALIZE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPEN MST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMS T0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x00002804	CANOPEN_MASTER_INITIALIZE_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_INITIALIZE_REQ_DATA_T			
ulReserved	UINT32		Reserved for further use, set to zero

Table 58: CANOPEN_MASTER_PACKET_INITIALIZE_REQ_T – Initialization of CANopen Master Request

Packet Status/Error

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

Table 59: CANOPEN_MASTER_INITIALIZE_REQ – Packet Status/Error

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_INITIALIZE_CNF_Ttag
  CANOPEN_MASTER_PACKET_INITIALIZE_CNF_T;

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

struct CANOPEN_MASTER_INITIALIZE_CNF_DATA_Ttag
{
  TLR_UINT32 ulReserved;
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_INITIALIZE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x00002805	CANOPEN_MASTER_INITIALIZE_CNF- Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_INITIALIZE_CNF_DATA_T			
ulReserved	UINT32		Reserved for further use, set to zero

Table 60: CANOPEN_MASTER_PACKET_INITIALIZE_CNF_T – Initialization of CANopen Master Confirmation

5.2.5 CANOPEN_MASTER_SET_BUSPARAM_REQ/CNF – Set Bus Parameters

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

Bus Parameter Structure Reference

```
typedef struct CANOPEN_MASTER_CFG_BUS_PARAM_Ttag
  CANOPEN_MASTER_CFG_BUS_PARAM_T;

#define CANOPEN_MASTER_MIN_MASTER_NODE_ID  1
#define CANOPEN_MASTER_MAX_MASTER_NODE_ID  127

#define CANOPEN_MASTER_CFG_BAUD_1000    0x00000000L   /* 1MBaud   */
#define CANOPEN_MASTER_CFG_BAUD_800     0x00000001L   /* 800kBaud */
#define CANOPEN_MASTER_CFG_BAUD_500     0x00000002L   /* 500kBaud */
#define CANOPEN_MASTER_CFG_BAUD_250     0x00000003L   /* 250kBaud */
#define CANOPEN_MASTER_CFG_BAUD_125     0x00000004L   /* 125kBaud */
#define CANOPEN_MASTER_CFG_BAUD_100     0x00000005L   /* 100kBaud */
#define CANOPEN_MASTER_CFG_BAUD_50      0x00000006L   /* 50kBaud  */
#define CANOPEN_MASTER_CFG_BAUD_20      0x00000007L   /* 20kBaud  */
#define CANOPEN_MASTER_CFG_BAUD_10      0x00000008L   /* 10kBaud  */

#define CANOPEN_MASTER_COB_ID_SYNC_MIN        0x00000001L
#define CANOPEN_MASTER_COB_ID_SYNC_MAX_11BIT  0x000007FFL
#define CANOPEN_MASTER_COB_ID_SYNC_MAX_29BIT  0x1FFFFFFFL

#define CANOPEN_MASTER_SYNC_TIMER_OFF         0x00000000L
#define CANOPEN_MASTER_MIN_SYNC_TIMER         0x00000001L
#define CANOPEN_MASTER_MAX_SYNC_TIMER         0x0000FFFFL

#define CANOPEN_MASTER_HEABT_TIME_OFF         0x00000000L
#define CANOPEN_MASTER_MIN_HEABT_TIME         0x00000001L
#define CANOPEN_MASTER_MAX_HEABT_TIME         0x0000FFFFL

#define CANOPEN_MASTER_AUTOCLEAR_DISABLE      0x00000000L
#define CANOPEN_MASTER_AUTOCLEAR_ENABLE       0x00000001L

#define CANOPEN_MASTER_FORMAT_INTEL           0x00000000L
#define CANOPEN_MASTER_FORMAT_MOTOROLA        0x00000001L

struct CANOPEN_MASTER_CFG_BUS_PARAM_Ttag
{
  TLR_UINT32    ulMasterNodeId;       /* Node ID of CANopen Master         */
  TLR_UINT32    ulBaudRate;           /* Baudrate                          */
  TLR_BOOLEAN32 fGlobalStartNode;     /* Send global start node            */
  TLR_BOOLEAN32 f29BitSelector;       /* Enable 29-Bit identifier          */
  TLR_UINT32    ulAcceptCode;         /* Accept code for 29-Bit identifier */
  TLR_UINT32    ulAcceptMask;         /* Accept mask for 29-Bit identifier */
  TLR_UINT32    ulCobIdSync;          /* Master SYNC message               */
  TLR_UINT32    ulSyncTimer;          /* Cycletime of SYNC message         */
  TLR_UINT32    ulProdHeartbeatTimer; /* Producer heartbeat time           */
  TLR_UINT32    ulAutoClear;          /* Auto clear mode on/off            */
  TLR_UINT32    ulIntelMotorola;      /* Intel or Motorola format          */
};
```

Variable	Type	Range	Explanation
ulMasterNodeId	UINT32	1…127	Node ID of CANopen Master
ulBaudRate	UINT32		*See Table 62: Codes and Corresponding Baud Rates of CANopen Network*
fGlobalStartNode	BOOLEAN32		Send global start node
f29BitSelector	BOOLEAN32		Enable 29-Bit identifier (Not supported yet, set to zero)
ulAcceptCode	UINT32		Accept code for 29-Bit identifier (Not supported yet, set to zero)
ulAcceptMask	UINT32		Accept mask for 29-Bit identifier (Not supported yet, set to zero)
ulCobIdSync	UINT32	1..2047	COB-ID for SYNC message
ulSyncTimer	UINT32		SYNC Timer
		0	Master does not produce SYNC messages
		1.. 65535	Producer cycle time of SYNC message in milliseconds
ulProdHeartbeatTimer	UINT32		Heartbeat producer time
		0	Master does not produce heartbeat messages
		1..65535	Producer cycle time of heartbeat message in milliseconds
ulAutoClear	UINT32		Auto clear mode
		0	off
		1	on
ulIntelMotorola	UINT32		Intel or Motorola format
		0	Intel format
		1	Motorola format

Table 61: CANOPEN_MASTER_CFG_BUS_PARAM_T - Bus Parameter Configuration

The variable `ulMasterNodeId` indicating the node ID of the CANopen master 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 62: Codes and Corresponding Baud Rates of CANopen Network

The flag `fGlobalStartNode` decides whether a global CANopen Start_Node signal is sent to the CANopen network by the master after initialization of all nodes. If the flag is logically 1 the master will send it, if it is logically 0 the master will not send it.

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.

The `ulAcceptCode` variable is part of a receive filter for the 29-bit COB-IDs which can be defined optionally. Those are the bits set to filter the IDs. Those bits must have the value ‘1’ in the acceptance code and the reaching COB ID to pass the filter. If a bit is not set in the Acceptance Mask, the filter will pass the message anyway. 29-bit mode is not supported yet.

Similarly, the `ulAcceptMask` variable is part of a receive filter for the 29-bit COB-IDs which optionally can be defined. Here it is possible to define the bits, the filter uses. In other words: All bits not set will not be filtered out. 29-bit mode is not supported yet.

The ulCobIdSync variable contains the COB-ID for the SYNC message.

This parameter has a valid range from 1 to 2047 the recommended values is 80h. This COB-ID is configured via the SDO-Download function to each configured node during the start up process of the bus.

The periodic cycle time of the SYNC message send mechanism can be adjusted with the ulSyncTimer variable. The reload timer value for the SYNC-message defines the cycle time in multiples of 1 msec., when the next SYNC-message will be transferred to the nodes. A value of 0 here indicates that the timer has been disabled.

The variable ulProdHeartbeatTimer enables or disables the Heartbeat request protocol of the device. The value can be either 0 or a time value specified in multiples of a millisecond. The specified value of time itself defines the time between two consecutive requests. If a value unequal 0 is configured the heartbeat request functionality of the device is enabled. In this case it starts right after finishing the initialization when entering the pre-operational state with sending the heartbeat requests as producer based on the values in the variables ulMasterNodeId and ulProdHeartbeatTimer. All other node devices which support heartbeat guarding and are configured to listen to this produced heartbeat requests can guard the device.

The variable ulAutoClear indicates whether auto clear mode is enabled or disabled. The auto clear mode defines the system behavior if one node classified as active has been disconnected. If auto clear mode has been activated then the card will stop the communication to all other nodes too, otherwise it will try to restart the missed node and keep on running.

The variable ulIntelMotorola indicates whether the data are interpreted according to the Intel or motorola format. It changes the interpretation of word oriented PDO data from LSB/MSB format to MSB/LSB format and vice versa.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SET_BUSPARAM_REQ_DATA_Ttag
  CANOPEN_MASTER_SET_BUSPARAM_REQ_DATA_T;

struct CANOPEN_MASTER_SET_BUSPARAM_REQ_DATA_Ttag
{
  CANOPEN_MASTER_CFG_BUS_PARAM_T tCfgBusParam;
};

typedef struct CANOPEN_MASTER_PACKET_SET_BUSPARAM_REQ_Ttag
  CANOPEN_MASTER_PACKET_SET_BUSPARAM_REQ_T;

struct CANOPEN_MASTER_PACKET_SET_BUSPARAM_REQ_Ttag
{
  TLR_PACKET_HEADER_T                    tHead; /** packet header.       */
  CANOPEN_MASTER_SET_BUSPARAM_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SET_BUSPARAM_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	44	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 Master-Task*
ulCmd	UINT32	0x00002806	*CANOPEN_MASTER_SET_BUSPARAM_REQ* - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_SET_BUSPARAM_REQ_DATA_T			
tCfgBusParam	*See Table 61: CANOPEN_MASTER_CFG_BUS_PARAM_T - Bus Parameter*		

Table 63: CANOPEN_MASTER_SET_BUSPARAM_REQ_DATA_T – Set Bus Parameter Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SET_BUSPARAM_CNF_DATA_Ttag
  CANOPEN_MASTER_SET_BUSPARAM_CNF_DATA_T;

struct CANOPEN_MASTER_SET_BUSPARAM_CNF_DATA_Ttag
{
  CANOPEN_MASTER_CFG_BUS_PARAM_T tCfgBusParam;
};

typedef struct CANOPEN_MASTER_PACKET_SET_BUSPARAM_CNF_Ttag
  CANOPEN_MASTER_PACKET_SET_BUSPARAM_CNF_T;

struct CANOPEN_MASTER_PACKET_SET_BUSPARAM_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead; /** packet header.            */
  CANOPEN_MASTER_SET_BUSPARAM_CNF_DATA_T tData; /** packet confirmation data. */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SET_BUSPARAM_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	Source End Point Identifier, untouched
ulLen	UINT32	44	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 Master-Task*
ulCmd	UINT32	0x00002807	CANOPEN_MASTER_SET_BUSPARAM_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_SET_BUSPARAM_REQ_DATA_T			
tCfgBusParam	*See Table 61: CANOPEN_MASTER_CFG_BUS_PARAM_T - Bus Parameter*		

Table 64: CANOPEN_MASTER_PACKET_SET_BUSPARAM_CNF_T –Set Bus Parameter Confirmation

5.2.6 CANOPEN_MASTER_SET_NODEPARAM_REQ/CNF – Set Node Parameters

This packet can be applied for setting the parameters for a node within the CANopen network before the bus parameters are configured. Because a node parameter set might exceed the maximum size of one packet, sequenced messages are supported for this request. If any error is detected during sequenced configuration, the current node parameter sequence will be canceled and must be restarted from the beginning. Further information about packet sequencing can be found in the `Task Layer Reference Manual`. This request will be denied if the configuration lock flag is set.

Each node parameter dataset consists of four sections in the following order:

- Parameter header
- SDO configuration data block
- PDO configuration data block
- Address table data block

Parameter Header Structure Reference

The parameter head consists of the following components:

```
typedef struct CANOPEN_MASTER_ND_PARAM_HEAD_Ttag
  CANOPEN_MASTER_ND_PARAM_HEAD_T;

#define CANOPEN_MASTER_MIN_SLAVE_NODE_ID  1
#define CANOPEN_MASTER_MAX_SLAVE_NODE_ID  127

#define CANOPEN_MASTER_NODE_PARAM_ACTIVE  0x80
#define CANOPEN_MASTER_NODE_PARAM_HRTBT   0x40

#define CANOPEN_MASTER_NODE_CFG_NO_NODE_RESET 0x01
#define CANOPEN_MASTER_NODE_CFG_NO_CMP_DEVICE 0x02
#define CANOPEN_MASTER_NODE_CFG_NO_GUARD_CFG  0x04
#define CANOPEN_MASTER_NODE_CFG_NO_SYNC_ID    0x08
#define CANOPEN_MASTER_NODE_CFG_NO_EMCY_ID    0x10
#define CANOPEN_MASTER_NODE_CFG_NO_CFG_SDO    0x20
#define CANOPEN_MASTER_NODE_CFG_NO_START_NODE 0x40
#define CANOPEN_MASTER_NODE_CFG_NO_TX_RX_REQ  0x80

struct CANOPEN_MASTER_ND_PARAM_HEAD_Ttag
{
  TLR_UINT8  bNodeId;
  TLR_UINT16 Node_Para_Len;
  TLR_UINT8  bNd_Flag;
  TLR_UINT16 usAddInfo;
  TLR_UINT16 usDevProfNum;
  TLR_UINT16 usCobIDEmcy;
  TLR_UINT16 usCobIDGuard;
  TLR_UINT16 usGuardTime;
  TLR_UINT8  bLifeTime;
  TLR_UINT8  bCfgNdState;
  TLR_UINT8  Octet_String[1];
};
```

Structure CANOPEN_MASTER_ND_PARAM_HEAD_T		
Variable name	**Type**	**Explanation**
bNodeId	UINT8	Node ID
Node_Para_Len	UINT16	Length of whole parameter, including all four sections of parameter set
bNd_Flag	UINT8	See below
usAddInfo	UINT16	Variable compared while the startup with the value in the node device type object 1000H
usDevProfNum	UINT16	Variable compared while the startup with the value in the node device type object 1000H
usCobIDEmcy	UINT16	CAN identifier for nodes EMCY-message, to be configured while the start up process
usCobIDGuard	UINT16	CAN identifier for ND-GUARD message, to be configured while the start up process
usGuardTime	UINT16	Guard time to supervise the node in multiples of 1msec
bLifeTime	UINT8	Factor of the guard time
bCfgNdState	UINT8	See below
Octet_String[1];	UINT8[1]	Reserved

Table 65: CANOPEN_MASTER_ND_PARAM_HEAD_T - Node Parameter Header

The first length indicator `usNodeParaLen` fixes the length of the whole data block inclusive the length of the size indicator.

This variable is followed by a special bit field called bNd_Flag. The topmost bit D7 = ACTIVE is declaring the parameter data set as active or inactive. If declared as not active, the device will not start any communication to this Node.

The next bit D6 = HRTBT decides whether the device will use either the traditional Guard Time Protocol or the Heartbeat Protocol to supervise a node existence during PDO communication. If not set the Guarding Protocol is used, if set the Heartbeat Protocol is used.

The other bits in this byte are reserved for future use.

bGlobalBits parameter			
Bit	**Short name**	**Name**	**Description**
D7	ACTIVE	Active	0 = node inactive in the current configuration
			1 = node active in the current configuration
D6	HRTBT	Heartbeat	0 = for Node error control protocol, node guarding is used
			1 = for Node error control protocol, Heartbeat is used
D5-D0			Reserved

Table 66: Node Flag Bit Field

The next two words represent the contents 'device type object 1000H' of the node station. Both values usAddInfo and usDevProfNum will be compared in the startup process of the node by reading the real object 1000H with SDO upload commands. If one of these two variables is not equal to the corresponding entry of the physically existing node, a runtime error will be generated and the node will not be brought into the OPERATIONAL state.

The CAN identifier usCobIDEmcy in the range of 1 to 2047 (decimal) is written into the node object directory during its startup.

The parameter usCobIDGuard has to be set to value 700h + Node-ID.

The Emergency COB ID is used later by the node to send Emergency Messages to the network like for example a low power indication. The standard calculation formula for the usCobIDEmcy is 80h + Node-ID.

The next two variables have different meanings based on the configured D6 = HRTBT bit in the variable bNd_Flag.

HRTBT = 0:

The variable usGuardTime defines the time the device will poll the node at regular time intervals. The value is a multiple of 1msec. If the value for example is set to 500, the master will request the node at every half second. The usGuardTime is automatically configured by the master during the nodes startup process by writing this value into its object 100Ch via SDO download commands. If the value is 0 the guarding protocol is disabled.

The node life time is given by the `usGuardTime` multiplied with the value `bLifeTime`. If the node has not been polled by the master during its life time, a remote node error is indicated by the node. So the `bLifeTime` can be seen as a safety factor the master gets time to send the node guarding request. The `bLifeTime` is automatically configured by the master during the nodes startup process by writing this value into its object 100Dh via SDO download commands. The Life Guarding is disabled if the parameter value is 0.

`HRTBT = 1:`

The variable `usGuardTime` defines the time the device will wait as consumer for heartbeat Indications of this node. The value itself reflects the time in multiples of a millisecond. If the value for example is set to 500, the device will wait in maximum 500 msecs for a heartbeat indication from this node until it would declare the node as not present and faulty. If the value is configured to 0 the device will not supervise the node and heartbeat supervision as consumer and node existence check is disabled.

The value `bLifeTime` has no meaning in heartbeat protocol and is not used.

The bit field in byte `bCfgNdState` configures the startup sequence the device is executing with the node. If the bit is logical '0' the corresponding service(s) is (are) active, if it is logical '1' it is (they are) deactivated.

Node Config State Bit Field		
Bit	**Name**	**Description**
D7	TXRXPDO	Request first TXPDOs and send first RXPDOs after initialization sequence is finished.
D6	START	Send Start Node Request
D5	SDO	Download of further Object data via SDO
D4	EMCY	Configure Emergency COB-ID
D3	SYNC	Configure SYNC COB-ID
D2	GUARD	Configure Guarding Parameter
D1	DEVICE	Check Device Profile
D0	RESET	Send Reset Node Request

Table 67: Node Config State Bit Field

SDO Configuration Data Block Structure Reference

The SDO list serves to change data in the object directory of the node via SDO download transfer. All object values be configured in this list are written into the located object directory entry one after the other while start up the node. Up to 2000 SDO download requests can be configured per node. Each entry in the list must have the following structure:

```
typedef struct CANOPEN_MASTER_SDO_CFG_DATA_Ttag
  CANOPEN_MASTER_SDO_CFG_DATA_T;

#define CANOPEN_MASTER_MAX_SDO_DATA_SIZE      0x04

struct CANOPEN_MASTER_SDO_CFG_DATA_Ttag
{
  TLR_UINT16 usIndex;
  TLR_UINT8  bSubIndex;
  TLR_UINT16 usLength;

  TLR_UINT8 abSdoValue[CANOPEN_MASTER_MAX_SDO_DATA_SIZE];
};

typedef struct CANOPEN_MASTER_ND_SDO_CFG_DATA_Ttag
  CANOPEN_MASTER_ND_SDO_CFG_DATA_T;

#define CANOPEN_MASTER_MAX_SDOS               2000

struct CANOPEN_MASTER_ND_SDO_CFG_DATA_Ttag
{
  TLR_UINT16  Sdo_Cfg_Data_Len;
  CANOPEN_MASTER_SDO_CFG_DATA_T Sdo_Cfg_Data[CANOPEN_MASTER_MAX_SDOS];
};
```

Structure CANOPEN_MASTER_SDO_CFG_DATA_T		
variable name	**type**	**explanation**
usIndex	word	Index of the object to be written
bSubIndex	byte	Sub-Index of the object to be written
usLength	word	length of data to be written 1-4
abSdoValue[4]	octet string	SDO data

Table 68: CANOPEN_MASTER_SDO_CFG_DATA_T - SDO Configuration Data Entry

Structure CANOPEN_MASTER_ND_SDO_CFG_DATA_T		
variable name	**type**	**explanation**
Sdo_Cfg_Data_Len	word	Length of SDO configuration data
CANOPEN_MASTER_SDO_CFG_DATA_T Sdo_Cfg_Data[ ]		*See Table 68: CANOPEN_MASTER_SDO_CFG_DATA_T - SDO Configuration Data*

Table 69: CANOPEN_MASTER_ND_SDO_CFG_DATA_T - SDO Configuration Data Block

The length of the SDO list fixes always the length of the SDO data field in bytes inclusive the size of the length indicator. If no objects should be changed, so that the SDO list is empty, a value of 2 results for the Sdo_Cfg_Data_Len.

Example:

Only the values in the object 6200H sub-index 5 and object 6200H sub-index 6 should be changed, the SDO configuration block should have the following data entries:

variable name	value
Sdo_Cfg_Data_Len	20
usIndex	0x6200
bSubIndex	0x05
usLength	0x04
abSdoValue[ ]	0x10010108
usIndex	0x6200
bSubIndex	0x06
usLength	0x04
abSdoValue[ ]	0x10010208

Table 70: SDO Configuration Example

PDO Configuration Data Block Structure Reference

The PDO list configures the CANopen process data messages (PDO) of this node. In the view of the master it enables the configured CAN identifier to be sent or to be received. So one entry in this list can configure either a transmit process data message = inputs in the view of the master or a receive process data message = outputs in the view of the master.

```
typedef struct CANOPEN_MASTER_PDO_CFG_DATA_Ttag
  CANOPEN_MASTER_PDO_CFG_DATA_T;

#define CANOPEN_MASTER_PDO_TYPE_RX            0x00
#define CANOPEN_MASTER_PDO_TYPE_TX            0xFF

#define CANOPEN_MASTER_PDO_LENGTH_MSK         0x7F
#define CANOPEN_MASTER_PDO_WORD_MSK           0x80

#define CANOPEN_MASTER_PDO_MAX_LEN            0x08

struct CANOPEN_MASTER_PDO_CFG_DATA_Ttag
{
  TLR_UINT8  bTypOfPdo;
  TLR_UINT8  bLength;
  TLR_UINT16 usPdoCOBId;
  TLR_UINT8  bTransType;
  TLR_UINT16 usInhibit;
};

typedef struct CANOPEN_MASTER_ND_PDO_CFG_DATA_Ttag
  CANOPEN_MASTER_ND_PDO_CFG_DATA_T;

#define CANOPEN_MASTER_MAX_CFG_TPDOS           128
#define CANOPEN_MASTER_MAX_CFG_RPDOS           128

#define CANOPEN_MASTER_MAX_CFG_PDOS       \
          CANOPEN_MASTER_MAX_CFG_TPDOS  + \
          CANOPEN_MASTER_MAX_CFG_RPDOS

struct CANOPEN_MASTER_ND_PDO_CFG_DATA_Ttag
{
  TLR_UINT16 Pdo_Cfg_Data_Len;
  CANOPEN_MASTER_PDO_CFG_DATA_T Pdo_Cfg_Data[CANOPEN_MASTER_MAX_CFG_PDOS];
};
```

Structure CANOPEN_MASTER_PDO_CFG_DATA_T		
variable name	**type**	**explanation**
bTypOfPdo	byte	Defines the type of PDO
bLength	byte	Length of the PDO data
usPdoCOBId	word	Used COB-ID for this PDO
bTransType	byte	Type of transmission corresponding CiA draft Standard 301
usInhibit	word	Time the next PDO can be transmitted

Table 71: CANOPEN_MASTER_PDO_CFG_DATA_T - PDO Configuration Data Entry

The first value decides whether the configured PDO is a transmit or a receive PDO.

`bTypeOfPdo = 0` : RXPDO (outputs in the view of the master)

`bTypeOfPdo <> 0` : TXPDO (inputs in the view of the master)

Structure CANOPEN_MASTER_ND_PDO_CFG_DATA_T		
variable name	**type**	**explanation**
Pdo_Cfg_Data_Len	word	Length of PDO configuration data
CANOPEN_MASTER_PDO_CFG_DATA_T Pdo_Cfg_Data[ ]		*See Table 71: CANOPEN_MASTER_PDO_CFG_DATA_T - PDO Configuration Data*

Table 72: CANOPEN_MASTER_ND_PDO_CFG_DATA_T - PDO Configuration Data Block

One CANopen PDO message can hold only 8 bytes of process data information in maximum, so the size indicator Length fixes the number of valid byte in this PDO and defines herewith the number of bytes to copy from and into the process data area of the dual-port memory. The highest bit in the length indicator byte has assigned a special function. It defines if the PDO data is either handled as word oriented data or byte oriented one. If word oriented is chosen, the behavior of the word data interpretation in this PDO can be changed between LSB/MSB and vice versa. This can be done in the bus parameter data set globally for all word oriented PDOs as described in section *CANOPEN_MASTER_SET_BUSPARAM_REQ/CNF – Set Bus Parameters*.

The COB-ID defines the CAN identifier, the node sends or receives this PDO. The CiA draft specification defines several transmission types for PDO. The value in `bTransType` corresponds to the CiA draft specification.

The parameter `usInhibit` defines the minimum time period in milliseconds between two transmissions of the same PDO.

Address Table Structure Reference

One entry in the PDO list results a corresponding entry in the ADDTab. In this table the offset address in the dual-port memory of each PDO is held down. See the following structure:

```
typedef struct CANOPEN_MASTER_ND_PRM_ADD_TAB_Ttag
  CANOPEN_MASTER_ND_PRM_ADD_TAB_T;

#define CANOPEN_MASTER_MAX_CFG_TPDOS           128
#define CANOPEN_MASTER_MAX_CFG_RPDOS           128

#define CANOPEN_MASTER_MAX_CFG_PDOS       \
          CANOPEN_MASTER_MAX_CFG_TPDOS  + \
          CANOPEN_MASTER_MAX_CFG_RPDOS

struct CANOPEN_MASTER_ND_PRM_ADD_TAB_Ttag
{
  TLR_UINT16 Add_Tab_Len;
  TLR_UINT8  bTxPdoCount;
  TLR_UINT8  bRxPdoCount;
  TLR_UINT16 EA_Offset[CANOPEN_MASTER_MAX_CFG_PDOS];
};
```

Structure CANOPEN_MASTER_ND_PRM_ADD_TAB_T		
variable name	**type**	**explanation**
Add_Tab_Len	word	Length of the following add_tab data inclusive the length of the size indicator.
bTxPdoCount	byte	Number of inputs following in the IO_Offset table
bRxPdoCount	byte	Number of output following in the IO_Offset table
EA_Offset[ ]	word array	Byte IO_Offset in the order: first all input offsets then all output offsets.

Table 73: CANOPEN_MASTER_ND_PRM_ADD_TAB_T - Address Table Configuration Data Block

If for example a node has one RXPDO and one TXPDO, the value for bTxPdoCount and the value for bRxPdoCount must be set to 1 followed first by the word input offset address and then the word output offset address. The length indicator Add_Tab_Len in this case is 8.

If a node has more than one of the same type of PDO, first all input offset addresses must be set in the EA_Offset table and than all output offsets can follow.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SET_NODEPARAM_REQ_DATA_Ttag
  CANOPEN_MASTER_SET_NODEPARAM_REQ_DATA_T;

#define CANOPEN_MASTER_MAX_NODEPARAM_DATA      512

/** Structure of task command set node parameter data */
struct CANOPEN_MASTER_SET_NODEPARAM_REQ_DATA_Ttag
{
  TLR_UINT8 abNodeParamData[CANOPEN_MASTER_MAX_NODEPARAM_DATA];
};

typedef struct CANOPEN_MASTER_PACKET_SET_NODEPARAM_REQ_Ttag
  CANOPEN_MASTER_PACKET_SET_NODEPARAM_REQ_T;

/** Structure of task command set node parameter request */
struct CANOPEN_MASTER_PACKET_SET_NODEPARAM_REQ_Ttag
{
  TLR_PACKET_HEADER_T                     tHead; /** packet header.       */
  CANOPEN_MASTER_SET_NODEPARAM_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_SET_NODEPARAM_REQ_T**			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	1.. 512	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet. Must be incremented with each request during sequenced configuration.
ulSta	UINT32		*See section 6.2 Codes of the CANopen Master-Task*
ulCmd	UINT32	0x00002808	`CANOPEN_MASTER_SET_NODEPARAM_REQ` - Command
ulExt	UINT32		Sequencing information
ulRout	UINT32	x	Routing information
Structure `CANOPEN_MASTER_SET_NODEPARAM_REQ_DATA_T`			
abNodeParam Data[512]	UINT8[ ]		Node Parameter Data Area

Table 74: `CANOPEN_MASTER_PACKET_SET_NODEPARAM_REQ_T` – Set Node Parameter Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SET_NODEPARAM_CNF_DATA_Ttag
  CANOPEN_MASTER_SET_NODEPARAM_CNF_DATA_T;

#define CANOPEN_MASTER_MAX_NODEPARAM_DATA      512

struct CANOPEN_MASTER_SET_NODEPARAM_CNF_DATA_Ttag
{
  TLR_UINT8 abNodeParamData[CANOPEN_MASTER_MAX_NODEPARAM_DATA];
};

typedef struct CANOPEN_MASTER_PACKET_SET_NODEPARAM_CNF_Ttag
  CANOPEN_MASTER_PACKET_SET_NODEPARAM_CNF_T;

/** Structure of task command set node parameter confirmation */
struct CANOPEN_MASTER_PACKET_SET_NODEPARAM_CNF_Ttag
{
  TLR_PACKET_HEADER_T                     tHead; /** packet header.            */
  CANOPEN_MASTER_SET_NODEPARAM_CNF_DATA_T tData; /** packet confirmation data. */
};
```

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_SET_NODEPARAM_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	Source End Point Identifier, untouched
ulLen	UINT32	1.. 512	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 Master-Task*
ulCmd	UINT32	0x00002809	CANOPEN_MASTER_SET_NODEPARAM_CNF - Command
ulExt	UINT32		Sequencing information
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_SET_NODEPARAM_CNF_DATA_T			
abNodeParam Data[512]	UINT8[ ]		Node Parameter Data Area

Table 75: CANOPEN_MASTER_PACKET_SET_NODEPARAM_CNF_T –Set Node Parameter Confirmation

5.2.7 CANOPEN_MASTER_GET_NODE_DIAG_REQ/CNF - Get Node Diagnostic

This packet can be applied for getting diagnostic data for a node in the CANopen network.

The variables of structure `tNodeDiag` of type `CANOPEN_MASTER_NODE_DIAG_T` returned by the confirmation packet are explained in detail in section 4.4.3 „*Obtaining Diagnostic Information from connected Slaves by sending an RCX_GET_SLAVE_CONN_INFO_REQ Packet*“ of this document, see at page 53.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_GET_NODE_DIAG_REQ_DATA_Ttag
  CANOPEN_MASTER_GET_NODE_DIAG_REQ_DATA_T;

#define CANOPEN_MASTER_NODE_DIAG_FLAG_PEEK 0x00000001L

struct CANOPEN_MASTER_GET_NODE_DIAG_REQ_DATA_Ttag
{
  TLR_UINT32 ulNodeId;
  TLR_UINT32 ulFlags;
};

typedef struct CANOPEN_MASTER_PACKET_GET_NODE_DIAG_REQ_Ttag
  CANOPEN_MASTER_PACKET_GET_NODE_DIAG_REQ_T;

struct CANOPEN_MASTER_PACKET_GET_NODE_DIAG_REQ_Ttag
{
  TLR_PACKET_HEADER_T                      tHead; /** packet header.            */
  CANOPEN_MASTER_GET_NODE_DIAG_REQ_DATA_T tData; /** packet request data.      */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_GET_NODE_DIAG_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x0000280E	CANOPEN_MASTER_GET_NODE_DIAG_REQ - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_GET_NODE_DIAG_REQ_DATA_T			
ulNodeId	UINT32	1..127	Node ID of the node for which diagnostic data are intended to be collected
ulFlags	UINT32		Flags BIT 0: PEEK 0 Node is deleted from diagnostic list after request 1 Node remains in diagnostic list after request BIT 1 - 31: Reserved for further use, se to zero

Table 76: CANOPEN_MASTER_PACKET_GET_NODE_DIAG_REQ_T – Get Node Diagnostic Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_GET_NODE_DIAG_CNF_Ttag
  CANOPEN_MASTER_PACKET_GET_NODE_DIAG_CNF_T;

 struct CANOPEN_MASTER_PACKET_GET_NODE_DIAG_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead; /** packet header.            */
  CANOPEN_MASTER_GET_NODE_DIAG_CNF_DATA_T tData; /** packet confirmation data. */
};

typedef struct CANOPEN_MASTER_GET_NODE_DIAG_CNF_DATA_Ttag
  CANOPEN_MASTER_GET_NODE_DIAG_CNF_DATA_T;

struct CANOPEN_MASTER_GET_NODE_DIAG_CNF_DATA_Ttag
{
  TLR_UINT32                ulNodeId;
  TLR_UINT32                ulFlags;
  CANOPEN_MASTER_NODE_DIAG_T tNodeDiag;
};
```

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_GET_NODE_DIAG_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	Source End Point Identifier, untouched
ulLen	UINT32	92	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 Master-Task*
ulCmd	UINT32	0x0000280F	*CANOPEN_MASTER_GET_NODE_DIAG_CNF* - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_GET_NODE_DIAG_CNF_DATA_T			
ulNodeId	UINT32		Node ID of the node for which diagnostic data are intended to be collected
ulFlags	UINT32		Bit Field for Flags
tNodeDiag	CANOPEN_MASTER_NODE_DIAG_T		See Table 33: CANOPEN_MASTER_NODE_DIAG_T - Node Diagnostic Structure

Table 77: CANOPEN_MASTER_GET_NODE_DIAG_CNF_DATA_T – Get Node Diagnostic Confirmation

5.2.8 CANOPEN_MASTER_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 Master 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_MASTER_GET_BUFFER_HANDLE_REQ_DATA_Ttag
  CANOPEN_MASTER_GET_BUFFER_HANDLE_REQ_DATA_T;

struct CANOPEN_MASTER_GET_BUFFER_HANDLE_REQ_DATA_Ttag
{
  TLR_UINT32 ulReserved;
};

typedef struct CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_REQ_Ttag
  CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_REQ_T;

struct CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_REQ_Ttag
{
  TLR_PACKET_HEADER_T                          tHead; /** packet header. */
  CANOPEN_MASTER_GET_BUFFER_HANDLE_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x0000280C	CANOPEN_MASTER_GET_BUFFER_HANDLE_REQ - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_GET_BUFFER_HANDLE_REQ_DATA_T			
ulReserved	UINT32		Reserved, set to zero

Table 78: CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_REQ_T – Get Buffer Handle Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_GET_BUFFER_HANDLE_CNF_DATA_Ttag
  CANOPEN_MASTER_GET_BUFFER_HANDLE_CNF_DATA_T;

#define CANOPEN_MASTER_RECV_OBJECT_CNT    28
#define CANOPEN_MASTER_SEND_OBJECT_CNT    28

struct CANOPEN_MASTER_GET_BUFFER_HANDLE_CNF_DATA_Ttag
{
  TLR_UINT32 aulRecvBuffer[CANOPEN_MASTER_RECV_OBJECT_CNT];
  TLR_UINT32 aulSendBuffer[CANOPEN_MASTER_SEND_OBJECT_CNT];
};

typedef struct CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_CNF_Ttag
  CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_CNF_T;

struct CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_CNF_Ttag
{
  TLR_PACKET_HEADER_T                        tHead; /** packet header. */
  CANOPEN_MASTER_GET_BUFFER_HANDLE_CNF_DATA_T tData; /** packet request data.  */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_GET_BUFFER_HANDLE_CNF			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	Source End Point Identifier, untouched
ulLen	UINT32	224	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 Master-Task*
ulCmd	UINT32	0x0000280D	CANOPEN_MASTER_GET_BUFFER_HANDLE_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_GET_BUFFER_HANDLE_CNF_DATA_T			
aulRecvBuffer[28]	UINT32[ ]		Receive Buffer
aulSendBuffer[28]	UINT32[ ]		Send Buffer

Table 79: CANOPEN_MASTER_GET_BUFFER_HANDLE_CNF – Get Buffer Handle Confirmation

5.2.9 CANOPEN_MASTER_STATE_CHANGE_IND/RES – Change of State Indication

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

- The master state
- The extended master 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 receive this indication, the CANOPEN_MASTER_REGISTER_REQ/CNF – Register Application request has to be done by the AP-Task.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_STATE_CHANGE_IND_DATA_Ttag
  CANOPEN_MASTER_STATE_CHANGE_IND_DATA_T;

struct CANOPEN_MASTER_STATE_CHANGE_IND_DATA_Ttag
{
  CANOPEN_MASTER_STATE_T          tMasterState;    /* Master state          */
  CANOPEN_MASTER_EXTENDED_STATE_T tExtMasterState; /* Extended Master state */
};

typedef struct CANOPEN_MASTER_PACKET_STATE_CHANGE_IND_Ttag
  CANOPEN_MASTER_PACKET_STATE_CHANGE_IND_T;

struct CANOPEN_MASTER_PACKET_STATE_CHANGE_IND_Ttag
{
  TLR_PACKET_HEADER_T                    tHead; /** packet header.            */
  CANOPEN_MASTER_STATE_CHANGE_IND_DATA_T tData; /** packet indication data.   */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_STATE_CHANGE_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of AP-Task Process Queue
ulSrc	UINT32		Source Queue-Handle of CANopen Master-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	216	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 Master-Task*
ulCmd	UINT32	0x00002812	`CANOPEN_MASTER_STATE_CHANGE_IND` - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure `CANOPEN_MASTER_STATE_CHANGE_IND_DATA_T`			
tMasterState	CANOPEN_MASTER_ STATE_T		Structure for Master state, see explanation below.
tExtMaster State	CANOPEN_MASTER_ EXTENDED_STATE_T		Structure for Extended Master state, see explanation below.

Table 80: `CANOPEN_MASTER_PACKET_STATE_CHANGE_IND_T` – Change of State Indication

CANopen Master State Structure Reference

```
typedef struct CANOPEN_MASTER_STATE_Ttag
  CANOPEN_MASTER_STATE_T;

#define CANOPEN_MASTER_STATE_FLAG_RDY        0x00000001L
#define CANOPEN_MASTER_STATE_FLAG_RUN        0x00000002L
#define CANOPEN_MASTER_STATE_FLAG_COM        0x00000004L
#define CANOPEN_MASTER_STATE_FLAG_BUS_ON     0x00000008L
#define CANOPEN_MASTER_STATE_FLAG_COMM_ERROR 0x00000010L

struct CANOPEN_MASTER_STATE_Ttag
{
  NETX_MASTER_STATUS          tNetxMasterState;    /* netX master state */
  TLR_UINT32                  ulCanState;          /* CAN state         */
  CANOPEN_MASTER_IO_STATUS_T  tIoStatus;           /* IO Status         */
  TLR_UINT32                  ulFlags;
  TLR_UINT32                  ulErrorCount;
  TLR_UINT32                  ulCommError;
  TLR_UINT32                  ulRunLedState;
  TLR_UINT32                  ulErrLedState;
  TLR_UINT32                  ulRecvDataCnt;
  TLR_UINT32                  ulSendDataCnt;

  TLR_UINT32                  ulReserved;
 };
```

netX Master Status Structure Reference

```
typedef struct tagNETX_MASTER_STATUS
{
  UINT32 ulSlaveState;
  UINT32 ulSlaveErrLogInd;
  UINT32 ulNumOfConfigSlaves;
  UINT32 ulNumOfActiveSlaves;
  UINT32 ulNumOfDiagSlaves;
  UINT32 ulReserved;
} NETX_MASTER_STATUS;
```

For more information, refer to section 3.3.1.2 about the master implementation of the common status.

CANopen Master IO Status Structure Reference

```
typedef struct CANOPEN_MASTER_IO_STATUS_Ttag
  CANOPEN_MASTER_IO_STATUS_T;

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

Extended Master State Structure Reference

```
typedef struct CANOPEN_MASTER_EXTENDED_STATE_Ttag
  CANOPEN_MASTER_EXTENDED_STATE_T;

struct CANOPEN_MASTER_EXTENDED_STATE_Ttag
{
  CANOPEN_MASTER_GLOBAL_STATE_T    tGlobalState;
  CANOPEN_MASTER_ADDITIONAL_INFO_T tAdditionalInfo;
};
```

The extended master state structure consists of two components, namely:

- The Global State Block
- The Additional Information Block

The extended master state is described in detail in section 3.3.2 “Extended Status”.

Global State Block Structure Reference

```
typedef struct CANOPEN_MASTER_GLOBAL_STATE_Ttag
  CANOPEN_MASTER_GLOBAL_STATE_T;

#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_CTRL 0x01
#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_ACLR 0x02
#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_NEXC 0x04
#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_FAT  0x08

#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_EVE  0x10
#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_NRDY 0x20
#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_TOUT 0x40
#define CANOPEN_MASTER_GLOBAL_STATE_FLAG_MUL  0x80

#define CANOPEN_MASTER_GLOBAL_CAN_STATE_OFFLINE 0x00
#define CANOPEN_MASTER_GLOBAL_CAN_STATE_STOP    0x40
#define CANOPEN_MASTER_GLOBAL_CAN_STATE_CLEAR   0x80
#define CANOPEN_MASTER_GLOBAL_CAN_STATE_OPERATE 0xC0

#define CANOPEN_MASTER_GLOBAL_STATE_ERROR_SIZE 4
#define CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE   16

#define CANOPEN_MASTER_GLOBAL_DIAG      0x000000ffL

struct CANOPEN_MASTER_GLOBAL_STATE_Ttag
{
  TLR_UINT8  bGlobalBits;
  TLR_UINT8  bCanState;
  TLR_UINT8  bErrorNodeAddress;
  TLR_UINT8  bErrorEvent;
  TLR_UINT16 usBusErrorCount;
  TLR_UINT16 usBusOffCount;
  TLR_UINT16 usMsgTimeOut;
  TLR_UINT16 usRxOverFlow;
  TLR_UINT8  abGlobalError[CANOPEN_MASTER_GLOBAL_STATE_ERROR_SIZE];
  TLR_UINT8  abListProjectedNodes[CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE];
  TLR_UINT8  abListActivatedNodes[CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE];
  TLR_UINT8  abListDiagnosticNodes[CANOPEN_MASTER_GLOBAL_NODE_LIST_SIZE];
};
```

For more information refer to section 3.3.2.1 “The Global Status Block” of this manual.

Additional Information Block Structure Reference

```
typedef struct CANOPEN_MASTER_ADDITIONAL_INFO_Ttag
  CANOPEN_MASTER_ADDITIONAL_INFO_T;

#define CANOPEN_MASTER_ADD_INFO_FLAG_CAN_INIT    0x00000001L
#define CANOPEN_MASTER_ADD_INFO_FLAG_CAN_ACTIVE  0x00000002L
#define CANOPEN_MASTER_ADD_INFO_FLAG_PASSIVE     0x00000004L
#define CANOPEN_MASTER_ADD_INFO_FLAG_BUS_OFF     0x00000008L

#define CANOPEN_MASTER_ADD_INFO_FLAG_RX_OVERFLOW 0x00000010L
#define CANOPEN_MASTER_ADD_INFO_FLAG_TX_OVERFLOW 0x00000020L

#define CANOPEN_MASTER_ADD_INFO_FLAG_WDG         0x00000100L

#define CANOPEN_MASTER_ADD_DETAIL_SIZE           0x00000003L

struct CANOPEN_MASTER_ADDITIONAL_INFO_Ttag
{
  TLR_UINT32 ulFlags;
  TLR_UINT32 ulReserved;
  TLR_UINT32 ulLastDiagAddress;
  TLR_UINT32 ulLastDiagInfo;
  TLR_UINT32 ulBusOffEveCnt;
  TLR_UINT32 ulErrorPassiveEveCnt;
  TLR_UINT32 ulRxOverflowCnt;
  TLR_UINT32 ulTxOverflowCnt;
  TLR_UINT32 aulReserved[8];
  TLR_UINT32 ulMaxRecvIdx;
  TLR_UINT32 ulMaxSendIdx;
  TLR_UINT32 aulAddDetail[CANOPEN_MASTER_ADD_DETAIL_SIZE];
};
```

For more information refer to section 3.3.2.2 “The Additional Info Block” of this manual.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_STATE_CHANGE_RES_Ttag
  CANOPEN_MASTER_PACKET_STATE_CHANGE_RES_T;

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

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_STATE_CHANGE_RES_T**			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x00002813	CANOPEN_MASTER_STATE_CHANGE_RES - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change

Table 81: CANOPEN_MASTER_PACKET_STATE_CHANGE_RES_T – Change of State Response

5.2.10 CANOPEN_MASTER_SDO_UPLOAD_REQ/CNF – SDO Upload

This packet handles SDO uploads, requested by the user application. Up to 512 bytes can be transferred with one SDO data transfer. If the requested node responds with an error, the error code is returned within the first 4 bytes of parameter `abSdoData`.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SDO_REQ_DATA_Ttag
  CANOPEN_MASTER_SDO_REQ_DATA_T;

#define CANOPEN_MASTER_MAX_SDO_DATA       512

struct CANOPEN_MASTER_SDO_REQ_DATA_Ttag
{
  TLR_UINT32 ulNodeId;
  TLR_UINT32 ulIndex;
  TLR_UINT32 ulSubIndex;
  TLR_UINT32 ulDataCnt;
  TLR_UINT8  abSdoData[CANOPEN_MASTER_MAX_SDO_DATA];
};

typedef struct CANOPEN_MASTER_PACKET_SDO_REQ_Ttag
  CANOPEN_MASTER_PACKET_SDO_REQ_T;

struct CANOPEN_MASTER_PACKET_SDO_REQ_Ttag
{
  TLR_PACKET_HEADER_T           tHead; /** packet header.            */
  CANOPEN_MASTER_SDO_REQ_DATA_T tData; /** packet request data.      */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SDO_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	16	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 Master-Task*
ulCmd	UINT32	0x00002814	CANOPEN_MASTER_SDO_UPLOAD_REQ - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_SDO_REQ_DATA_T			
ulNodeId	UINT32	1 ... 127	Node ID of the node of the CANopen network to be read
ulIndex	UINT32	0 ... 65535	Index
ulSubIndex	UINT32	0 ... 255	Sub index
ulDataCnt	UINT32	1 ... 512	Number of data bytes to read
abSdoData [512]	UINT8[ ]		Unused

Table 82: CANOPEN_MASTER_PACKET_SDO_REQ_T – SDO Upload Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SDO_CNF_DATA_Ttag
  CANOPEN_MASTER_SDO_CNF_DATA_T;

struct CANOPEN_MASTER_SDO_CNF_DATA_Ttag
{
  TLR_UINT32 ulNodeId;
  TLR_UINT32 ulIndex;
  TLR_UINT32 ulSubIndex;
  TLR_UINT32 ulDataCnt;
  TLR_UINT8  abSdoData[CANOPEN_MASTER_MAX_SDO_DATA];
};

typedef struct CANOPEN_MASTER_PACKET_SDO_CNF_Ttag
  CANOPEN_MASTER_PACKET_SDO_CNF_T;

struct CANOPEN_MASTER_PACKET_SDO_CNF_Ttag
{
  TLR_PACKET_HEADER_T           tHead; /** packet header.            */
  CANOPEN_MASTER_SDO_CNF_DATA_T tData; /** packet request data.      */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SDO_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	Source End Point Identifier, untouched
ulLen	UINT32	16 ... 528	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 Master-Task*
ulCmd	UINT32	0x00002815	CANOPEN_MASTER_SDO_UPLOAD_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_SDO_CNF_DATA_T			
ulNodeId	UINT32	1 ... 127	Node ID of the node to be read
ulIndex	UINT32	0 ... 65535	Index
ulSubIndex	UINT32	0 ... 255	Sub index
ulDataCnt	UINT32	1 ... 512	Data count
abSdoData [512]	UINT8[ ]		SDO upload data or if available, the error code the node returns via CANopen

Table 83: CANOPEN_MASTER_PACKET_SDO_CNF_T – SDO Upload Confirmation

5.2.11 CANOPEN_MASTER_SDO_DOWNLOAD_REQ/CNF – SDO Download

This packet handles SDO downloads, requested by the user application. Up to 512 bytes can be transferred with one SDO data transfer. If the requested node responds with an error, the error code is returned within the first 4 bytes of parameter `abSdoData`.

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SDO_REQ_DATA_Ttag
  CANOPEN_MASTER_SDO_REQ_DATA_T;

#define CANOPEN_MASTER_MAX_SDO_DATA       512

struct CANOPEN_MASTER_SDO_REQ_DATA_Ttag
{
  TLR_UINT32 ulNodeId;
  TLR_UINT32 ulIndex;
  TLR_UINT32 ulSubIndex;
  TLR_UINT32 ulDataCnt;
  TLR_UINT8  abSdoData[CANOPEN_MASTER_MAX_SDO_DATA];
};

typedef struct CANOPEN_MASTER_PACKET_SDO_REQ_Ttag
  CANOPEN_MASTER_PACKET_SDO_REQ_T;

struct CANOPEN_MASTER_PACKET_SDO_REQ_Ttag
{
  TLR_PACKET_HEADER_T           tHead; /** packet header.            */
  CANOPEN_MASTER_SDO_REQ_DATA_T tData; /** packet request data.      */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SDO_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	17 ... 528	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 Master-Task*
ulCmd	UINT32	0x00002816	CANOPEN_MASTER_SDO_DOWNLOAD_REQ - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_SDO_REQ_DATA_T			
ulNodeId	UINT32	1 ... 127	Node ID of the node of the CANopen network to be read
ulIndex	UINT32	0 ... 65535	Index
ulSubIndex	UINT32	0 ... 255	Sub index
ulDataCnt	UINT32	1 ... 512	Number of data bytes to write
abSdoData [512]	UINT8[ ]		SDO download data

Table 84: CANOPEN_MASTER_PACKET_SDO_REQ_T – SDO Download Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_SDO_CNF_DATA_Ttag
  CANOPEN_MASTER_SDO_CNF_DATA_T;

struct CANOPEN_MASTER_SDO_CNF_DATA_Ttag
{
  TLR_UINT32 ulNodeId;
  TLR_UINT32 ulIndex;
  TLR_UINT32 ulSubIndex;
  TLR_UINT32 ulDataCnt;
  TLR_UINT8  abSdoData[CANOPEN_MASTER_MAX_SDO_DATA];
};

typedef struct CANOPEN_MASTER_PACKET_SDO_CNF_Ttag
  CANOPEN_MASTER_PACKET_SDO_CNF_T;

struct CANOPEN_MASTER_PACKET_SDO_CNF_Ttag
{
  TLR_PACKET_HEADER_T           tHead; /** packet header.            */
  CANOPEN_MASTER_SDO_CNF_DATA_T tData; /** packet request data.      */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SDO_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	Source End Point Identifier, untouched
ulLen	UINT32	16, 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 Master-Task*
ulCmd	UINT32	0x00002817	CANOPEN_MASTER_SDO_DOWNLOAD_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change
Structure CANOPEN_MASTER_SDO_CNF_DATA_T			
ulNodeId	UINT32	1 ... 127	Node ID of the node to be read
ulIndex	UINT32	0 ... 65535	Index
ulSubIndex	UINT32	0 ... 255	Sub index
ulDataCnt	UINT32	1 ... 512	Data count
abSdoData [512]	UINT8[ ]		The error code the node returns via CANopen

Table 85: CANOPEN_MASTER_PACKET_SDO_CNF_T – SDO Download Confirmation

5.2.12 CANOPEN_MASTER_SEND_EMCY_REQ/CNF – Send Emergency Message

This packet sends an emergency telegram to the CANopen network.

The emergency error codes in variable `usErrorCode` have the following meaning:

Error Code	Meaning
00xx	Error reset or no error
10xx	Generic error
20xx	Current
21xx	Current, device input side
22xx	Current inside the device
23xx	Current, device output side
30xx	Voltage
31xx	Main voltage
32xx	Voltage inside the device
33xx	Output voltage
40xx	Temperature
41xx	Ambient temperature
42xx	Device temperature
50xx	Device hardware
60xx	Device software
61xx	Internal software
62xx	User software
63xx	Data set
70xx	Additional modules
80xx	Monitoring
81xx	Communication
8110	CAN overrun (objects lost)
8120	CAN in Error Passive Mode
8130	Life Guard Error or Heartbeat Error
8140	Recovered from bus-off
8150	Transmit COB-ID collision
82xx	Protocol error
8210	PDO not processed due to length error
8220	PDO length exceeded
90xx	External error
F0xx	Additional functions
FFxx	Device-specific

Table 86: Emergency Error Codes (Variable `usErrorCode`)

The bits of the error register in variable `bErrorRegister` have the following meaning:

Error	Code
CANOPEN_MASTER_ERROR_REGISTER_GENERIC_BIT	0x01
CANOPEN_MASTER_ERROR_REGISTER_CURRENT_BIT	0x02
CANOPEN_MASTER_ERROR_REGISTER_VOLTAGE_BIT	0x04
CANOPEN_MASTER_ERROR_REGISTER_TEMPERATURE_BIT	0x08
CANOPEN_MASTER_ERROR_REGISTER_COMM_ERROR_BIT	0x10
CANOPEN_MASTER_ERROR_REGISTER_DEV_PROFILE_BIT	0x20
CANOPEN_MASTER_ERROR_REGISTER_RESERVED_BIT	0x40
CANOPEN_MASTER_ERROR_REGISTER_MANU_SPEC_BIT	0x80

Table 87: Error Register (Variable `bErrorRegister`)

Packet Structure Reference

```
#define CANOPEN_MASTER_EMCY_DATA_SIZE 5

typedef struct CANOPEN_MASTER_SEND_EMCY_REQ_DATA_Ttag
  CANOPEN_MASTER_SEND_EMCY_REQ_DATA_T;

struct CANOPEN_MASTER_SEND_EMCY_REQ_DATA_Ttag
{
  TLR_UINT16 usErrorCode;
  TLR_UINT8  abManErrorCode[CANOPEN_MASTER_EMCY_DATA_SIZE];
  TLR_UINT8  bErrorRegister;
};

typedef struct CANOPEN_MASTER_PACKET_SEND_EMCY_REQ_Ttag
  CANOPEN_MASTER_PACKET_SEND_EMCY_REQ_T;

struct CANOPEN_MASTER_PACKET_SEND_EMCY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                tHead; /** packet header.            */
  CANOPEN_MASTER_SEND_EMCY_REQ_DATA_T tData; /** packet request data.      */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SEND_EMCY_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x00002818	CANOPEN_MASTER_SEND_EMCY_REQ - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_SEND_EMCY_REQ_DATA_T			
usErrorCode	UINT16		Error Code
abManError Code[5]	UINT8[ ]		Area for Error Code
bErrorRegister	UINT8	Bit mask	*See Table 87: Error Register*

Table 88: CANOPEN_MASTER_PACKET_SEND_EMCY_REQ_T – Send Emergency Message Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_SEND_EMCY_CNF_Ttag
  CANOPEN_MASTER_PACKET_SEND_EMCY_CNF_T;

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SEND_EMCY_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x00002819	CANOPEN_MASTER_SEND_EMCY_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change

Table 89: CANOPEN_MASTER_PACKET_SEND_EMCY_CNF_T – Send Emergency Message Confirmation

5.2.13 CANOPEN_MASTER_NODE_NMT_COMMAND_REQ/CNF – Set NMT State

This packet allows you to control the state of the NMT nodes in the CANopen network. Normally the device itself as a master device takes care on the different states of the nodes, but sometimes it may be necessary to control the nodes in their states manually from the host application.

Which operation will be performed at the chosen node depends on the value of the variable `ulNmtState`, which may have the values described in the following table:

Value	Symbolic Name	Meaning	Applicable for
1	CANOPEN_MASTER_NMT_COMMAND_START	Starts the node	Slave, Master
2	CANOPEN_MASTER_NMT_COMMAND_STOP	Stops the node	Slave, Master
128	CANOPEN_MASTER_NMT_COMMAND_ENTER_PREOPERATIONAL	Sets the node into the preoperational state	Slave, Master
129	CANOPEN_MASTER_NMT_COMMAND_RESET_NODE	Resets the node	Slave only
130	CANOPEN_MASTER_NMT_COMMAND_RESET_COMMUNICATION	Resets the communication to the node	Slave only

Table 90: NMT States

These commands are standardized and have been specified in the CANopen Application Layer Profile.

The node to be controlled can be addressed by the `ulNodeId` variable (applicable address value range is 1-127), as an NMT node is uniquely identified in the network by its Node ID. There are also two special cases which are supported since stack version 2.14.:

- The value 0 addresses all connected nodes including the master.
- The value 128 addresses all connected nodes, but not the master.

Note: It is not possible to reset the master using this packet. If you intend to do that, you have to perform a channel-init.

Packet Structure Reference

```
#define CANOPEN_MASTER_NMT_COMMAND_NODE_ID_ALL_NODES        0 /* All nodes, including
Master*/
#define CANOPEN_MASTER_NMT_COMMAND_MIN_SINGLE_NODE_ID       1
#define CANOPEN_MASTER_NMT_COMMAND_MAX_SINGLE_NODE_ID       127
#define CANOPEN_MASTER_NMT_COMMAND_NODE_ID_ALL_REMOTE_NODES 128

typedef struct CANOPEN_MASTER_NODE_NMT_COMMAND_REQ_DATA_Ttag
  CANOPEN_MASTER_NODE_NMT_COMMAND_REQ_DATA_T;

struct CANOPEN_MASTER_NODE_NMT_COMMAND_REQ_DATA_Ttag
{
  TLR_UINT32 ulNodeId;
  TLR_UINT32 ulNmtState;
};
typedef struct CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_REQ_Ttag
  CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_REQ_T;

struct CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_REQ_Ttag
{
  TLR_PACKET_HEADER_T                        tHead; /** packet header.          */
  CANOPEN_MASTER_NODE_NMT_COMMAND_REQ_DATA_T tData; /** packet request data.    */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x0000281A	*CANOPEN_MASTER_NODE_NMT_COMMAND_REQ* - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_NODE_NMT_COMMAND_REQ_DATA_T			
ulNodeId	UINT32	0 1..127 128	All connected nodes, including the master are accessed Node ID of the node to be accessed All connected nodes are accessed
ulNmtState	UINT32	1-2, 128-130	*See Table 90: NMT States*

Table 91: CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_REQ_T – Set NMT State Request

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_CNF_Ttag
  CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_CNF_T;

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

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x0000281B	CANOPEN_MASTER_NODE_NMT_COMMAND_CNF - Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change

Table 92: CANOPEN_MASTER_PACKET_NODE_NMT_COMMAND_CNF_T – Set NMT State Confirmation

5.2.14 CANOPEN_MASTER_SET_WATCHDOG_FAIL_REQ/CNF – Set Watchdog Fail

This packet is used by the AP-Task in order to inform the CANopen Master-Task that a watchdog failure has been detected. The CANopen Master-Task stops the CANopen network. After a watchdog error has been set, the master has to bee 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 Master task

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_REQ_Ttag
  CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_REQ_T;

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task.*
ulCmd	UINT32	0x000028AA	CANOPEN_MASTER_SET_WATCHDOG_FAIL_REQ - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 93: CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_REQ_T – Set Watchdog Fail Request

Packet Status/Error

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

Table 94: CANOPEN_MASTER_SET_WATCHDOG_FAIL_REQ – Packet Status/Error

Packet Structure Reference

```
typedef struct CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_CNF_Ttag
  CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_CNF_T;

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

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x000028AB	CANOPEN_MASTER_SET_WATCHDOG_FAIL_CNF– Command
ulExt	UINT32		Extension, reserved
ulRout	UINT32		Routing information, do not change

Table 95: CANOPEN_MASTER_PACKET_SET_WATCHDOG_FAIL_CNF_T – Set Watchdog Fail Confirmation

5.2.15 CANOPEN_MASTER_SLAVE_ACTIVATE_REQ/CNF – Activate Slave

This packet can be used to activate (by setting `ulMode` to `CANOPEN_MASTER_SLAVE_ACTIVATE = 1`) or deactivate (by setting `ulMode` to `CANOPEN_MASTER_SLAVE_DEACTIVATE = 2`) the communication to a specific slave from the CANopen Master. The slave to be activated/ deactivated is addressed by its NodeId (specified in variable `ulNodeId`).

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_SLAVE_ACTIVATE_REQ_DATA_Ttag */
typedef struct CANOPEN_MASTER_SLAVE_ACTIVATE_REQ_DATA_Ttag
CANOPEN_MASTER_SLAVE_ACTIVATE_REQ_DATA_T;

#define CANOPEN_MASTER_SLAVE_ACTIVATE   0x00000001L
#define CANOPEN_MASTER_SLAVE_DEACTIVATE 0x00000002L

struct CANOPEN_MASTER_SLAVE_ACTIVATE_REQ_DATA_Ttag
{
  TLR_UINT32 ulNodeId;
  TLR_UINT32 ulMode;

}; /** type of CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_REQ_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_REQ_Ttag
  CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_REQ_T;

struct CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_REQ_Ttag
{
  TLR_PACKET_HEADER_T tHead;                      /** packet header. */
  CANOPEN_MASTER_SLAVE_ACTIVATE_REQ_DATA_T tData; /** packet data    */
};

/*****************************************************************************/
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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	0	*See section 6.2 Codes of the CANopen Master-Task*
ulCmd	UINT32	0x000028AC	*CANOPEN_MASTER_SLAVE_ACTIVATE_REQ* - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_NODE_NMT_COMMAND_REQ_DATA_T			
ulNodeId	UINT32	1..127	Node ID of the node to be accessed
ulMode	UINT32	1,2	Activate / deactivate node 1: Activate node 2: Deactivate node

Table 96: CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_REQ_T – Slave Activate Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_CNF_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_CNF_Ttag
CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_CNF_T;

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

/*****************************************************************************/
```

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x000028AD	*CANOPEN_MASTER_SLAVE_ACTIVATE_CNF* - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 97: CANOPEN_MASTER_PACKET_SLAVE_ACTIVATE_CNF_T – Confirmation to Slave Activate Request

5.2.16 CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ/CNF – Enable/Disable PDO Counter

This packet allows to enable (`ulMode= CANOPEN_MASTER_ENABLE_PDO_COUNTER=1`) or to disable (`ulMode= CANOPEN_MASTER_DISABLE_PDO_COUNTER=0`) the PDO counter.

The confirmation packet additionally returns a handle to the PDO counter buffer (`hPdoCounterBuffer`).

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_Ttag */
typedef struct CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_Ttag
  CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_T;

#define CANOPEN_MASTER_DISABLE_PDO_COUNTER 0x00000000L /* Disable PDO counter */
#define CANOPEN_MASTER_ENABLE_PDO_COUNTER  0x00000001L /* Enable PDO counter  */

#define CANOPEN_MASTER_PDO_COUNTER_ELEMENTS        128
#define CANOPEN_MASTER_PDO_COUNTER_ELEMENT_SIZE sizeof(TLR_UINT8)

/* structures for all needed packets of PDO-Counter-Functionality */
struct CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_Ttag
{
  TLR_UINT32 ulMode;

};

/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_REQ_Ttag */
typedef struct CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_REQ_Ttag
  CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_REQ_T;

struct CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_REQ_Ttag
{
  TLR_PACKET_HEADER_T                                tHead;  /** packet header. */
  CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_T tData;   /** packet data    */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task.*
ulCmd	UINT32	0x28AE	CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
Structure CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ_DATA_T			
ulMode	UINT32	0,1	Mode 0: Disable PDO Counter 1: Enable PDO Counter

Table 98: CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_REQ –Enable/Disable PDO Counter Request

Packet Structure Reference

```
/*****************************************************************************/

#define CANOPEN_MASTER_DISABLE_PDO_COUNTER 0x00000000L /* Disable PDO counter */
#define CANOPEN_MASTER_ENABLE_PDO_COUNTER  0x00000001L /* Enable PDO counter  */

#define CANOPEN_MASTER_PDO_COUNTER_ELEMENTS        128
#define CANOPEN_MASTER_PDO_COUNTER_ELEMENT_SIZE sizeof(TLR_UINT8)

/* structures for all needed packets of PDO-Counter-Functionality */
struct CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_Ttag
{
  TLR_UINT32 ulMode;
  TLR_HANDLE hPdoCounterBuffer;
};

/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_CNF_Ttag */
typedef struct CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_CNF_Ttag
  CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_CNF_T;

struct CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_CNF_Ttag
{
  TLR_PACKET_HEADER_T                                  tHead;   /** packet header. */
  CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_T tData;    /** packet data    */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_ENABLE_DISABLE_PDO_COUNTER_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task.*
ulCmd	UINT32	0x28AF	CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
Structure CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_CNF_DATA_T			
ulMode	UINT32	0,1	Mode 0: Disable PDO Counter 1: Enable PDO Counter
hPdoCounterBuffer	TLR_HANDLE		Handle to PDO Counter buffer

Table 99: CANOPEN_MASTER_ENABLE_DISABLE_PDO_COUNTER_CNF – Confirmation of Enable/Disable PDO Counter Request

5.2.17 CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ/CNF – Force compare values

This packet can be used to set compare data for acyclic TxPDOs with transmission types 0, 254 and 255. The packet supplies 8 bytes being used as compare values (array parameter `abPdoData[]`). These compare values are used after the next handshake to decide whether the acyclic TXPDOs are sent or not.

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ_DATA_Ttag  */
typedef struct CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ_DATA_Ttag
  CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ_DATA_T;

#define CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ            0x000028B0
#define CANOPEN_MASTER_SET_COMPARE_IMAGE_PDO_NUM_ALL_PDO  0x0000

/* structures for all needed packets of set compare image */
struct CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ_DATA_Ttag
{
  TLR_UINT16 usPdoNummer; /*PDO-Number */
  TLR_UINT8  abPdoData[CANOPEN_MASTER_PDO_MAX_LEN];  /*Value to be set             */
};

/** type of <code>CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_REQ_Ttag</code> */
typedef struct CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_REQ_Ttag
  CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_REQ_T;

struct CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_REQ_Ttag
{
  TLR_PACKET_HEADER_T                          tHead;  /** packet header. */
  CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ_DATA_T tData;  /** packet data    */
};
/*****************************************************************************/
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPENMST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	10	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 Master-Task*
ulCmd	UINT32	0x000028B0	CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_SET_COMPARE_IMAGE_REQ_DATA_T			
usPdoNummer	UINT16	0	PDO-Number (Currently all PDOs (0) is supported only)
abPdoData[]	UINT8[CANOPEN_MASTER_PDO_MAX_LEN]		Value to be used for comparison

Table 100: CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_REQ_T – Force compare values

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_CNF_Ttag  */
typedef struct CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_CNF_Ttag
  CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_CNF_T;

#define CANOPEN_MASTER_SET_COMPARE_IMAGE_CNF            0x000028B1

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

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	ulAPMS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENMST0Id	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 Master-Task*
ulCmd	UINT32	0x000028B1	CANOPEN_MASTER_SET_COMPARE_IMAGE_CNF – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 101: CANOPEN_MASTER_PACKET_SET_COMPARE_IMAGE_CNF_T – Confirmation of force compare values request

5.2.18 CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ/CNF – Configure SYNC Trigger

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

This packet can be used to configure the SYNC trigger. There are three trigger modes available which can be set in parameter bSyncTrigger:

Mode	Value	Meaning
CANOPEN_MASTER_SYNC_ TRIGGER_TIME_CONTROLLED	0	*Time-controlled trigger mode* Triggering is done time-controlled, i.e. it takes place in regular time intervals. No indications are sent on the occurance of the SYNC trigger event.
CANOPEN_MASTER_SYNC_ TRIGGER_NOTIFIED_TIME_CONTROLLED	1	*Time-controlled trigger mode with notification* Triggering is done time-controlled with indication (see section *CANOPEN_MASTER_SYNC_IND/RES – SYNC Indication* on page 147).
CANOPEN_MASTER_SYNC_ TRIGGER_APPLICATION_TRIGGERED	2	*Application-controlled trigger mode* Triggering is controlled exclusively by the application (see section *CANOPEN_MASTER_SEND_SYNC_REQ/CNF – Send SYNC* on page 145). The configured time interval is not used at all.

Table 102: Possible values of bSyncTrigger

If you send this packet to set the trigger mode, this setting will be effective until the next CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ packet is sent. The cycle time is set by ulSyncTimer (see section *CANOPEN_MASTER_SET_BUSPARAM_REQ/CNF – Set Bus Parameters* on page 87) or by SYCON.net. For more information on the handshake modes and synchronization options supported by the CANopen Master protocol stack, see section 4.5 ”*Handshake Modes and Synchronization*” on page 56.

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ_DATA_Ttag */
typedef struct CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ_DATA_Ttag
  CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ_DATA_T;

#define CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ              0x000028B2

#define CANOPEN_MASTER_SYNC_TRIGGER_TIME_CONTROLLED            0x00
#define CANOPEN_MASTER_SYNC_TRIGGER_NOTIFIED_TIME_CONTROLLED   0x01
#define CANOPEN_MASTER_SYNC_TRIGGER_APPLICATION_TRIGGERED      0x02

struct CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ_DATA_Ttag
{
  TLR_UINT8 bSyncTrigger;
};
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_REQ_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_REQ_Ttag
  CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_REQ_T;

struct CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_REQ_Ttag
{
  TLR_PACKET_HEADER_T                        tHead;  /** packet header. */
  CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ_DATA_T tData; /** packet data*/

};
/*****************************************************************************/
```

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_REQ_T**			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPEN MST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMS T0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	1	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 Master-Task*
ulCmd	UINT32	0x000028B2	CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ_DATA_T			
bSyncTrigger	UINT8	0…2	SYNC trigger mode (see *Table 102: Possible values of bSyncTrigger*)

Table 103: CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_REQ_T – Configure SYNC Trigger Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_CNF_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_CNF_Ttag
CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_CNF_T;

#define CANOPEN_MASTER_SET_SYNC_TRIGGER_CNF              0x000028B3

struct CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;   /** packet header. */

};
/*****************************************************************************/
```

Packet Description

Structure Information **CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	`ulAPMS0Id`	Destination End Point Identifier, untouched
ulSrcId	UINT32	`ulCANOPENMST0Id`	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 Master-Task*
ulCmd	UINT32	`0x000028B3`	`CANOPEN_MASTER_SET_SYNC_TRIGGER_CNF` – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 104: `CANOPEN_MASTER_PACKET_SET_SYNC_TRIGGER_CNF_T` – Confirmation to Configure SYNC Trigger Request

5.2.19 CANOPEN_MASTER_SEND_SYNC_REQ/CNF – Send SYNC

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

This packet can be used to initiate SYNC events from the host application in application-controlled trigger mode (see preceding section *CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ/CNF – Configure SYNC Trigger* on page 142). When the application sends this packet, the SYNC telegram is sent to the network.

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SEND_SYNC_REQ_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SEND_SYNC_REQ_Ttag
  CANOPEN_MASTER_PACKET_SEND_SYNC_REQ_T;

#define CANOPEN_MASTER_SEND_SYNC_REQ                    0x000028B4

struct CANOPEN_MASTER_PACKET_SEND_SYNC_REQ_Ttag
{
  TLR_PACKET_HEADER_T tHead;  /** packet header. */

};
/*****************************************************************************/
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SEND_SYNC_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPEN MST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMS T0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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 *Codes of the CANopen Master-Task* on page 154.
ulCmd	UINT32	0x000028B4	CANOPEN_MASTER_SEND_SYNC_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 105: CANOPEN_MASTER_PACKET_SEND_SYNC_REQ_T – Initialization of CANopen Master Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SEND_SYNC_CNF_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SEND_SYNC_CNF_Ttag
  CANOPEN_MASTER_PACKET_SEND_SYNC_CNF_T;

#define CANOPEN_MASTER_SEND_SYNC_CNF                    0x000028B5

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SEND_SYNC_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	`ulAPMS0Id`	Destination End Point Identifier, untouched
ulSrcId	UINT32	`ulCANOPENMST0Id`	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 Master-Task*
ulCmd	UINT32	`0x000028B5`	`CANOPEN_MASTER_SEND_SYNC_CNF` – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 106: `CANOPEN_MASTER_PACKET_SEND_SYNC_CNF_T` – Initialization of CANopen Master Request

5.2.20 CANOPEN_MASTER_SYNC_IND/RES – SYNC Indication

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

This packet indicates that the SYNC telegram has been sent on the network.

This packet is only sent if the SYNC-Mode has been set to ‚CANOPEN_MASTER_SYNC_TRIGGER_NOTIFIED_TIME_CONTROLLED' (`bSyncTrigger = 1`) using the `CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ` packet, see section *CANOPEN_MASTER_SET_SYNC_TRIGGER_REQ/CNF – Configure SYNC Trigger* on page 142 for more information.

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SYNC_IND_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SYNC_IND_Ttag
  CANOPEN_MASTER_PACKET_SYNC_IND_T;

#define CANOPEN_MASTER_SYNC_IND                          0x000028B6

struct CANOPEN_MASTER_PACKET_SYNC_IND_Ttag
{
  TLR_PACKET_HEADER_T tHead;  /** packet header. */

};
/*****************************************************************************/
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SYNC_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
`ulDest`	UINT32	0x20/ `QUE_CANOPENMST`	Destination Queue-Handle of CANopen Master-Task Process Queue
`ulSrc`	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
`ulDestId`	UINT32	`ulCANOPENMST0Id`	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
`ulSrcId`	UINT32	`ulAPMS0Id`	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 Master-Task*
`ulCmd`	UINT32	`0x000028B6`	`CANOPEN_MASTER_SYNC_IND` – Command
`ulExt`	UINT32	0	Reserved
`ulRout`	UINT32	x	Routing information

Table 107: CANOPEN_MASTER_PACKET_SYNC_IND_T – SYNC Indication

Packet Structure Reference

```
/*****************************************************************************/
/** type of CANOPEN_MASTER_PACKET_SYNC_RES_Ttag */
typedef struct CANOPEN_MASTER_PACKET_SYNC_RES_Ttag
  CANOPEN_MASTER_PACKET_SYNC_RES_T;

#define CANOPEN_MASTER_SYNC_RES                          0x000028B7

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_SYNC_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMST0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulAPMS0Id	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 Master-Task*
ulCmd	UINT32	0x000028B7	CANOPEN_MASTER_SYNC_RES– Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 108: CANOPEN_MASTER_PACKET_SYNC_RES_T – Initialization of CANopen Master Request

5.2.21 CANOPEN_MASTER_RESET_ERROR_REQ/CNF – Reset Error Event

Since stack version 2.14, this packet offers the opportunity to reset the error counters and error indicating bits in the global bus status field.

The CANopen Master clears the following diagnostic information:

In structure CANOPEN_MASTER_GLOBAL_STATE_T (see section 3.3.2.1„The Global Status Block“ on page 38):

- The flags CANOPEN_MASTER_GLOBAL_STATE_FLAG_EVE and CANOPEN_MASTER_GLOBAL_STATE_FLAG_TOUT in bGlobalBits are cleared
- The following variables are set to 0:
  - usBusErrorCount
  - usBusOffCnt
  - usMsgTimeOut
  - usRxOverflow

In structure CANOPEN_MASTER_ADDITIONAL_INFO_T (see section 3.3.2.2„Additional Info Block“ on page 42):

- The flags CANOPEN_MASTER_ADD_INFO_FLAG_RX_OVERFLOW and CANOPEN_MASTER_ADD_INFO_FLAG_TX_OVERFLOW in ulFlags are cleared
- The following variables are set to 0:
  - ulLastDiagAddress
  - ulLastDiagInfo
  - ulBusOffEveCnt
  - ulErrorPassiveEveCnt
  - ulRxOverflowCnt
  - ulTxOverflowCnt
  - aulAddDetail

The confirmation packet will indicate the successful clearing of the status field.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_MASTER_RESET_ERROR_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_MASTER_RESET_ERROR_REQ_DATA_Ttag
  CANOPEN_MASTER_RESET_ERROR_REQ_DATA_T;

struct CANOPEN_MASTER_RESET_ERROR_REQ_DATA_Ttag
{
  TLR_UINT32 ulReserved;    /* Reserved, set to zero */

};
/*****************************************************************************/
/** type of <code>CANOPEN_MASTER_PACKET_RESET_ERROR_REQ_Ttag</code> */
typedef struct CANOPEN_MASTER_PACKET_RESET_ERROR_REQ_Ttag
CANOPEN_MASTER_PACKET_RESET_ERROR_REQ_T;

struct CANOPEN_MASTER_PACKET_RESET_ERROR_REQ_Ttag
{
  TLR_PACKET_HEADER_T                   tHead;  /** packet header. */
  CANOPEN_MASTER_RESET_ERROR_REQ_DATA_T tData;  /** packet data.   */
};
```

Packet Description

Structure Information CANOPEN_MASTER_PACKET_RESET_ERROR_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ QUE_CANOPEN MST	Destination Queue-Handle of CANopen Master-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	ulCANOPENMS T0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPMS0Id	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	0	See section *Codes of the CANopen Master-Task* on page 154.
ulCmd	UINT32	0x000028B8	CANOPEN_MASTER_RESET_ERROR_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information
Structure CANOPEN_MASTER_RESET_ERROR_REQ_DATA_T			
ulReserved	UINT32	0	Reserved, set to zero

Table 109: CANOPEN_MASTER_PACKET_SEND_SYNC_REQ_T – Reset Error Event Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_MASTER_PACKET_RESET_ERROR_CNF_Ttag</code> */
typedef struct CANOPEN_MASTER_PACKET_RESET_ERROR_CNF_Ttag
CANOPEN_MASTER_PACKET_RESET_ERROR_CNF_T;

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

Packet Description

Structure Information CANOPEN_MASTER_PACKET_RESET_ERROR_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	`ulAPMS0Id`	Destination End Point Identifier, untouched
ulSrcId	UINT32	`ulCANOPENMST0Id`	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 Master-Task*
ulCmd	UINT32	`0x000028B9`	`CANOPEN_MASTER_RESET_ERROR_CNF` – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing information

Table 110: `CANOPEN_MASTER_PACKET_RESET_ERROR_CNF_T` – Confirmation to Reset Error Event Request

5.3 CAN-DL Task

If working with Loadable Firmware, you can also use the functionality of the CAN-DL Task for programming CAN on the level of Data Link Layer (Layer 2 in OSI Layer Model).

The packet interface of CAN DL is described within a separate manual, the CAN Data Link Packet Interface Protocol API Manual. See reference [3].

The following packets of CAN DL can be used without restrictions:

- `CAN_DL_CMD_DATA_REQ`
- `CAN_DL_CMD_DATA_HI_REQ`
- `CAN_DL_CMD_DIAG_REQ`
- `CAN_DL_CMD_TX_ABORT_REQ`
- `CAN_DL_CMD_AP_REGISTER_REQ`
- `CAN_DL_CMD_EVENT_ACK_REQ`

The following packets of CAN DL will be denied as long as the ‘Configuration Lock’ flag is set:

- `CAN_DL_CMD_ENABLE_RXID_REQ`
- `CAN_DL_CMD_SET_EVENTS_TO_INDICATE_REQ`

Whether or not indications are sent to your application, depends on which CAN-DL events have been notified!

Contrary to the CANopen Master Task, the CAN-DL Task also supports 29 bit CAN identifiers. If it is intended to use these 29 bit CAN identifiers, the application has to registered at the CAN-DL Task using `CAN_DL_CMD_AP_REGISTER_REQ` with parameter `ulInitMode` set to 0.

6 Status/Error Codes Overview

6.1 Codes of the CANopen-APM-Task

6.1.1 Error Messages

Hexadecimal 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
0xC0490004	TLR_E_CANOPEN_APM_WATCHDOG_PARAMETER Invalid parameter for watchdog.
0x40490005	TLR_I_CANOPEN_APM_OPEN_DBM_FILE Database file not found.
0xC0490006	TLR_E_CANOPEN_APM_DATASET Failed to open configuration dataset.
0xC0490007	TLR_E_CANOPEN_APM_TABLE_GLOBAL Failed to open GLOBAL configuration dataset.
0xC0490008	TLR_E_CANOPEN_APM_TABLE_BUS_CAN Failed to open BUS_CAN configuration dataset.
0xC0490009	TLR_E_CANOPEN_APM_TABLE_BUS_CAN_EXT Failed to open BUS_CAN_EXT configuration dataset.
0xC049000A	TLR_E_CANOPEN_APM_ TABLE_NODES Failed to open NODES configuration dataset.
0xC049000B	TLR_E_CANOPEN_APM_WATCHDOG_ACTIVATE Failed to activate watchdog supervision.
0xC049000C	TLR_E_CANOPEN_APM_SIZE_TABLE_BUS_CAN Invalid size of BUS_CAN configuration dataset.
0xC049000D	TLR_E_CANOPEN_APM_SIZE_TABLE_BUS_CAN_EXT Invalid size of BUS_CAN_EXT configuration dataset.
0xC049000E	TLR_E_CANOPEN_APM_NODE_ALREADY_CONFIGURED Node already configured.
0xC049000F	TLR_E_CANOPEN_APM_INVALID_NODE_ID Invalid Node ID.
0xC0490010	TLR_E_CANOPEN_APM_DATABASE_FOUND Configuration database found.
0xC0490011	TLR_E_CANOPEN_APM_REQUEST_RUNNING Request already running.

Table 111: Error Messages of the AP-Task

6.2 Codes of the CANopen Master-Task

6.2.1 Error Messages

The following table defined the error messages of the CANopen Master-Task:

Hexadecimal 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
0xC0420003	TLR_E_CANOPEN_MASTER_DATA_COUNT Invalid data count.
0xC0420004	TLR_E_CANOPEN_MASTER_DATA_OFFSET Invalid data offset.
0xC0420005	TLR_E_CANOPEN_MASTER_DATA_COUNT_WITH_OFFSET Invalid data count in combination with offset.
0xC0420006	TLR_E_CANOPEN_MASTER_MODE Invalid mode in command.
0xC0420007	TLR_E_CANOPEN_MASTER_STATE Command is not allowed in current state.
0xC0420008	TLR_E_CANOPEN_MASTER_NO_VALID_BUS_PARAM No valid bus configuration parameterized.
0xC0420009	TLR_E_CANOPEN_MASTER_REQUEST_RUNNING A request is already running.
0xC042000A	TLR_E_CANOPEN_MASTER_BUS_RUNNING Command is not allowed because CANopen is running.
0xC042000B	TLR_E_CANOPEN_MASTER_BUS_PARAM_ALREADY_SET Bus parameters are already configured.
0xC042000C	TLR_E_CANOPEN_MASTER_LOCAL_NODE_ID Invalid Node ID for CANopen Master.
xC042000D	TLR_E_CANOPEN_MASTER_BAUDRATE Invalid Baud rate.
0xC042000E	TLR_E_CANOPEN_MASTER_29BIT_SELECTOR Invalid parameter for 29 bit selector.
0xC042000F	TLR_E_CANOPEN_MASTER_SYNC_TIMER_VALUE Invalid parameter for SYNC timer.
0xC0420010L	TLR_E_CANOPEN_MASTER_COB_ID_SYNC Invalid parameter for COB-ID SYNC.
0xC0420011	TLR_E_CANOPEN_MASTER_PROD_HEARTBEAT_TIME Invalid parameter for Producer Heartbeat time.
0xC0420012	TLR_E_CANOPEN_MASTER_PACKET_SEQUENCE Invalid packet sequence detected.

Hexadecimal Value	Definition Description
0xC0420013	TLR_E_CANOPEN_MASTER_NODE_PARAM_SET_SIZE Invalid size of Node parameter set.
0xC0420014	TLR_E_CANOPEN_MASTER_NODE_PARAM_HEADER_SIZE Invalid size of Node parameter header.
0xC0420015	TLR_E_CANOPEN_MASTER_NODE_ALREADY_CONFIGURED Node is already configured.
0xC0420016	TLR_E_CANOPEN_MASTER_SLAVE_NODE_ID Invalid Node ID for Slave.
0xC0420017	TLR_E_CANOPEN_MASTER_NODE_ID_EQUAL Node ID of Slave is equal to Master Node ID.
0xC0420018	TLR_E_CANOPEN_MASTER_PARAMETER_SET_LENGTH Length of parameter set is different from length in parameter header.
0xC0420019	TLR_E_CANOPEN_MASTER_SDO_PARAMETER_SET_LENGTH Invalid size of SDO parameter set.
0xC042001A	TLR_E_CANOPEN_MASTER_PDO_PARAMETER_SET_LENGTH Invalid size of PDO parameter set.
0xC042001B	TLR_E_CANOPEN_MASTER_ADDR_TABLE_SET_LENGTH Invalid size of address table.
0xC042001C	TLR_E_CANOPEN_MASTER_ADDR_TABLE_LENGTH_INCONSISTENT Address table size is inconsistent.
0xC042001E	TLR_E_CANOPEN_MASTER_TPDO_CNT Invalid number of transmitted PDOs.
0xC042001F	TLR_E_CANOPEN_MASTER_RPDO_CNT Invalid number of received PDOs.
0xC0420020	TLR_E_CANOPEN_MASTER_COB_ID_EMCY Invalid value for COB-ID Emergency.
0xC0420021	TLR_E_CANOPEN_MASTER_COB_ID_GUARD Invalid value for COB-ID Guard.
0xC0420022	TLR_E_CANOPEN_MEMORY_ALLOCATION No memory for parameter set.
0xC0420023	TLR_E_CANOPEN_SDO_DATA_CNT Invalid value for SDO data count.
0xC0420024	TLR_E_CANOPEN_PDO_DATA_CNT Invalid value for PDO data count.
0xC0420025	TLR_E_CANOPEN_ADDR_TAB_DATA_CNT Invalid value for address table data count.
0xC0420026	TLR_E_CANOPEN_ADDR_TAB_PDO_CNT Invalid value for address table PDO count.
0xC0420027	TLR_E_CANOPEN_MASTER_NODE_SDO_TIMEOUT Timeout during SDO transfer.
0xC0420028	TLR_E_CANOPEN_MASTER_NODE_SDO_ERROR Error during SDO transfer.

Hexadecimal Value	Definition Description
0xC0420029	TLR_E_CANOPEN_MASTER_NO_PDO_AVAILABLE No further PDO available.
0xC042002A	TLR_E_CANOPEN_MASTER_AUTO_CLEAR_ACTIVE Master is in auto clear state.
0xC042002B	TLR_E_CANOPEN_MASTER_WATCHDOG_FAIL Watchdog failure detected.
0xC042002C	TLR_E_CANOPEN_MASTER_INVALID_INDEX Invalid index for request.
0xC042002D	TLR_E_CANOPEN_MASTER_NODE_STATE Request not possible in current Node state.
0xC042002E	TLR_E_CANOPEN_MASTER_NODE_NOT_CONFIGURED Node is not configured.
0xC042002F	TLR_E_CANOPEN_MASTER_SDO_REQUEST_FAILED SDO request failed.
0x40420030	TLR_I_CANOPEN_MASTER_ALREADY_IN_STATE Master is already in requested state.
0xC0420031	TLR_E_CANOPEN_MASTER_COB_ID_PDO Invalid value for PDO COB-ID.
0xC0420032	TLR_E_CANOPEN_MASTER_SEND_EMCY Send emergency-telegram failed.
0xC0420033	TLR_E_CANOPEN_MASTER_INIT_SDO_REQUEST Failed to initialize SDO request.
0xC0420034	TLR_E_CANOPEN_MASTER_SET_NMT_STATE Set NMT state failed.
0xC0420035	TLR_E_CANOPEN_MASTER_ERROR_PASSIVE CANopen is in error-passive state.
0xC0420036	TLR_E_CANOPEN_MASTER_BUS_OFF CANopen is in bus-off state.
(0x40420037	TLR_I_CANOPEN_MASTER_NODE_DEACTIVATED Node is deactivated in configuration.
0xC0420038	TLR_E_CANOPEN_MASTER_DL_REQ_FAILED CAN-DL request failed.
0xC0420039	TLR_E_CANOPEN_MASTER_PUT_OBJECT_DATA Failed to write object data.
0xC042003A	TLR_E_CANOPEN_MASTER_SET_OBJECT_DATA_VALID Failed to set object data valid.
0xC042003B	TLR_E_CANOPEN_MASTER_INIT_LIB Failed to initialize CANopen library.
0xC042003C	TLR_E_CANOPEN_MASTER_SET_COB_ID_FAILED COB-ID could not be set.

Hexadecimal Value	Definition Description
0xC042003D	TLR_E_CANOPEN_MASTER_ADD_REMOTE_NODE_ REQUEST Failed to add remote Node.
0xC042003E	TLR_E_CANOPEN_MASTER_SET_HEARTBEAT_TIME Heartbeat time could not be set.
0xC042003F	TLR_E_CANOPEN_MASTER_DD_GUARDING_SLAVE Node could not be added to Node guarding list.
0xC0420040	TLR_E_CANOPEN_MASTER_SET_GUARDING_TIME Node guard time could not be set.
0xC0420041	TLR_E_CANOPEN_MASTER_START_NODE_GUARD Node guarding could not be started.
0xC0420042	TLR_E_CANOPEN_MASTER_RESET_NODE Reset Node failed.
0xC0420043	TLR_E_CANOPEN_MASTER_RESET_COMMUNICATION Failed to reset communication of Node.
0xC0420044	TLR_E_CANOPEN_MASTER_SET_NODE_PREOPERATIONAL Failed to set Node to preoperational state.
0xC0420045	TLR_E_CANOPEN_MASTER_STOP_NODE Failed to set Node to stop state.
0xC0420046	TLR_E_CANOPEN_MASTER_START_NODE Failed to set Node to operational state.
0xC0420047	TLR_E_CANOPEN_MASTER_SET_EMCY_COB_ID Failed to set Emergency COB-ID.
0xC0420048	TLR_E_CANOPEN_MASTER_START_SYNC Failed to start SYNC-telegram.
0xC0420049	TLR_E_CANOPEN_MASTER_STOP_SYNC Failed to stop SYNC-telegram.
0xC042004A	TLR_E_CANOPEN_MASTER_NODE_UNEXPECTED_STATE Node is not in expected state.
0xC042004B	TLR_E_CANOPEN_MASTER_NODE_LOST_CONNECTION Connection to Node lost.
0xC042004C	TLR_E_CANOPEN_MASTER_NODE_GUARDING_ERROR Node guarding error.
0xC042004D	TLR_E_CANOPEN_MASTER_NODE_HEARTBEAT_ERROR Heartbeat error.
0x4042004E	TLR_I_CANOPEN_MASTER_NODE_HEARTBEAT_STARTED Heartbeat supervision of Node started
0xC042004F	TLR_E_CANOPEN_MASTER_NODE_UNEXPECTED_BOOTUP Unexpected Boot up message from Node received.
0xC0420050	TLR_E_CANOPEN_MASTER_WRITE_PDO_REQ Failed to transmit PDO.

Hexadecimal Value	Definition Description
0xC0420051	TLR_E_CANOPEN_MASTER_READ_PDO_REQ Failed to request PDO.
0xC0420052	TLR_E_CANOPEN_MASTER_INIT_BUFFER Initialization of buffer failed
0x40420053	TLR_I_CANOPEN_MASTER_NODE_STATE_NOT_ HANDLED State of Node not handled.
0xC0420054	TLR_E_CANOPEN_MASTER_NODE_DEVICE_TYPE Node Device Type unequal to configured Device Type.
0x40420055	TLR_I_CANOPEN_MASTER_NODE_EMERGENCY_RECEIVED Emergency message received from Node
0x40420056	TLR_I_CANOPEN_MASTER_INITIALIZE Master is initializing.
0x40420057	TLR_I_CANOPEN_MASTER_NODE_BOOTUP Boot up message from Node received.

Table 112: Error Messages of the CANopen Master-Task

6.3 Codes of the CAN DL-Task

6.3.1 Error Messages

The error messages of the CAN DL-Task are listed in reference #3.

7 Appendix

7.1 List of Tables

7.2 List of Figures

7.3 Contacts

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 Systemautomation (Shanghai) Co. Ltd.
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.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

Italy
Hilscher Italia S.r.l.
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.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
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