Protocol API

CANopen Slave

V3.8.0

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC111001API07EN | Revision 7 | English | 2020-11 | Released | Public

Table of Contents

1 Introduction

1.1 Abstract

This manual describes the application interface of the CANopen Slave stack.

1.2 List of Revisions

Rev	Date	Name	Revisions
5	2013-10-29	RG/ES	Firmware/stack version V3.6.2.x Reference to Object Dictionary V3.3.2.x Reference to CAN_DL V2.0.27.0 Technical data updated (number of consumers for netX51/52) Small corrections Error corrections in description of hardware switches
6	2016-06-27	HH, RG	Firmware/stack version V3.7.0
			Manual revised: Sections *Status information* and *Special Topics* created.
			Reference to Object Dictionary V3.3.2.x
			Reference to CAN_DL V2.0.27.0
			Section *Bus parameter*: Bit 20 in ulCanOpenFlags added: Enable reject if restricted CAN ID is confugured.
			Section *Manufacturer-specific Error Codes* added.
7	2020-11-19	HHE	Firmware/stack version V3.8.0
			Section Where can I switch between Standard Mode and Extended Mode?: corrected to bit 28.
			COMX 52xx-COS (netX 52) added.

Table 1: List of Revisions

1.3 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

1.4 Intended Audience

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

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

1.5 Specifications

The data below applies to CANopen Slave firmware and stack version 3.7.0. The firmware/stack has been designed in order to meet the CiA Work Draft 301 V4.02 specification (see reference [2]).

1.5.1 Technical Data

Technical Data

Features	Parameter
Maximum number of input data	Depends on the used mode and settings. See below.
Maximum number of output data	Depends on the used mode and settings. See below.
Maximum number of receive PDOs	Depends on the used mode and settings. See below.
Maximum number of transmit PDOs	Depends on the used mode and settings. See below.
Exchange of process data	via PDO transfer (synchronized, remotely requested, event driven (change of date)), requested by application (via packet))
Acyclic communication	SDO Up- and Download (Server only), Emergency message (producer), Timestamp (producer/consumer)
Functions	Node guarding / life guarding, heartbeat 1 producer max. 64 consumer (netX 50/51/100/500) max. 32 consumer (netX 52) PDO Mapping NMT Slave SYNC protocol (consumer) Error behavior in state operational: change to state pre-operational no state change change to state stopped
Baud rates	10 kBit/s to 1 MBit/s Automatic detection
Data transport layer	CAN Frames can be accessed by programming the CAN DL layer, see reference [6]
CAN Frame type	11 Bit 11/29 Bit layer 2 transparent

Table 2: Technical Data - Protocol Stack

Firmware/stack available for netX

netX	Available
netX 50	yes
netX 51	yes (from stack V3.3.1)
netX 52	yes (from stack V3.5.1)
netX 100, netX 500	yes

Table 3: Technical Data – Available for netX

PCI - DMA

Features	Parameter
DMA Support for PCI targets	yes

Table 4: Technical Data – PCI-DMA

Slot Number

Features	Devices
Slot number supported for	CIFX 50-CO, CIFX 50E-CO, CIFX 70E-CO

Table 5: Technical Data – Slot Number

Configuration

For configuration of standard mode with default settings:

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

For configuration of standard mode with default settings and configured settings and extended mode:

- by packet to transfer configuration parameters.

Diagnostic

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

1.5.1.1 Technical Data (Standard Mode)

In standard mode, the following values and limitations apply:

Technical Data for default Settings

Features	Parameter
Default number of input data	512 bytes (netX 50/100/500) 256 bytes (netX 52)
Default number of output data	512 bytes (netX 50/100/500) 256 bytes (netX 52)
Default number of receive PDOs	64 (netX 50/100/500) 32 (netX 52)
Default number of transmit PDOs	64 (netX 50/100/500) 32 (netX 52)

Table 6: Technical Data - Protocol Stack (Standard Mode – Default Settings)

Note: The EDS files for Hilscher standard products contain the functionality that matches the default settings. SYCON.net and the netX Configuration tool only configure the default settings.

Technical Data for configured Settings

Features	Parameter
Maximum number of input data	1020 bytes
Maximum number of output data	1020 bytes
Number of receive PDOs	0 … 255, for mapping objects 2200 … 2203
Number of transmit PDOs	0 … 255, for mapping objects 2000 … 2003

Table 7: Technical Data - Protocol Stack (Standard Mode – Configured Settings)

Note 1: Using other settings than the default settings requires a suitable EDS file.

Note 2: The actual maximum number of IO Data and PDOs depends on the **available amount of memory**.

Note 3: SYCON.net and netX configuration tool do not support the configuration of the extended mode.

1.5.1.2 Technical Data (Extended Mode)

In extended mode, the stack offers extended functionality. To use these functions requires an application program that configures and supports these functions, e.g. to create an own object dictionary.

In extended mode, more input and output data can be used and transmit and receive PDOs can be used.

Note: The actual maximum number of IO Data and PDOs depends on the **available amount of memory**.

To use the extended mode requires creating a suitable EDS file. The knowledge of the EDS specification is required.

Features	Parameter
Maximum number of input data	2048 bytes
Maximum number of output data	2048 bytes
Maximum number of receive PDOs	256
Maximum number of transmit PDOs	256

Table 8: Technical Data - Protocol Stack (Extended Mode)

Other settings than default must be set via “Set Configuration Packet” and object dictionary configuration.

Concerning the extended mode, also see section *Standard Mode vs. Extended Mode* on page 33.

Note: SYCON.net and netX configuration tool do not support the configuration of the extended mode.

1.6 Terms, Abbreviations and Definitions

Term	Description
AP	Application 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
OD	Object Dictionary
ODV3	Object Dictionary Version 3
PDO	Process Data Object (process data channel)
PDO-Mapping	Configurable process data per PDO
RTR	Remote transmission request
RxPDO	Receive PDO
SDO	Service Data Object (representing an acyclic data channel)
SYNC	Synchronization cycle of the CANopen slave
TxPDO	Transmit PDO

Table 9: Terms, Abbreviations and Definitions

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

1.7 References to Documents

This document refers to the following documents:

[1] EN 50325/4 Specification

[2] CAN in Automation e.V., Nuremberg: CANopen Application Layer and Communication Profile, CiA Public Specification 301, Version 4.2.0, 2011.

[3] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, netX Dual-Port Memory Interface, Revision 17, English, 2020.

[4] Hilscher Gesellschaft für Systemautomation mbH: Packet API, netX Dual-Port Memory, Packet-based services (netX 10/50/51/52/100/500), Revision 4, 2020.

[5] Hilscher Gesellschaft für Systemautomation mbH: Protocol API, Object Dictionary, for CANopen and EtherCAT, V3.3, Revision 3, English, 2013.

[6] Hilscher Gesellschaft für Systemautomation mbH: Protocol API, CAN Data Link, Packet Interface, 1.0, Revision 3, English, 2013.

Table 10: References

2 Getting Started

2.1 Task Structure of the CANopen Slave V3 Stack

The figure below shows the internal structure of the tasks which represent the CANopen Slave stack:

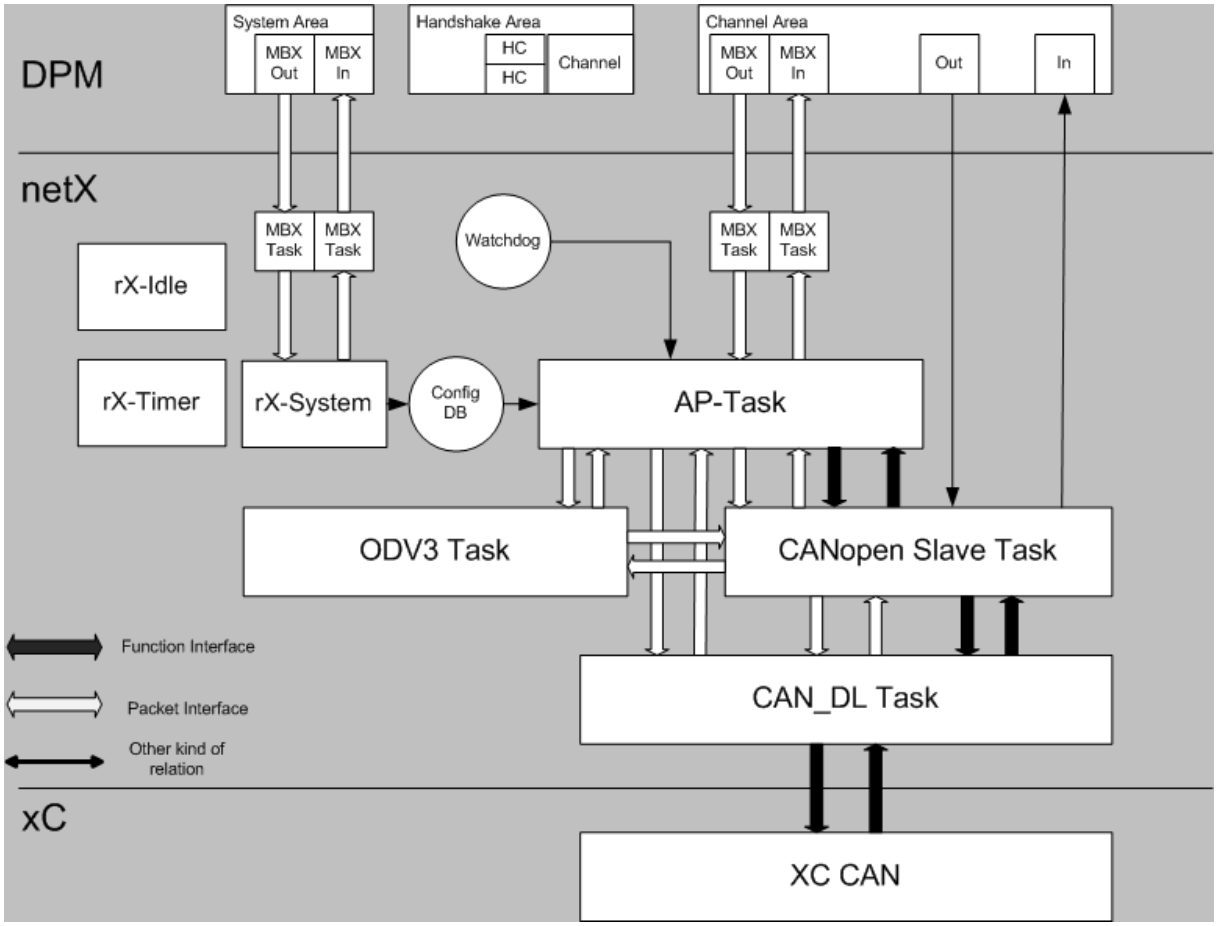


Figure 1: Internal Structure of CANopen Slave V3 Firmware

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

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

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 the protocol stack
- IO Process data exchange
- Channel mailboxes
- Watchdog supervision
- Handling of applications packets
- Send/Receive packets

CANopen Slave Task

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

The packets of the CANopen Slave task are described in section 3.1.1 “CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration”

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.

The CAN_DL Task also provides its own API for low-level programming on level 2 (Data Link Layer) of the OSI model of networking. This is described in a separate manual (namely reference [6]).

ODV3 Task

The ODV3 task handles all SDO accesses (i.e. acyclic accesses) to the CANopen object dictionary as described in the CANopen specification, section 9.5, p.79 (see reference [2]).

2.2 Configuration

The CANopen Slave stack requires configuration parameters e.g. node address. Configuration parameters can be set

- from the application using the ‘Set Configuration’ packet
- using the configuration software SYCON.net (only applicable for *Standard Mode*)
- using the configuration software netX configuration tool (only applicable for *Standard Mode*)

2.3 CANopen – Basic Topics

2.3.1 NMT Slave State Machine

Each NMT Slave device implements the NMT Slave state machine. This state machine is displayed in *Figure 2: NMT Slave State Machine* below.

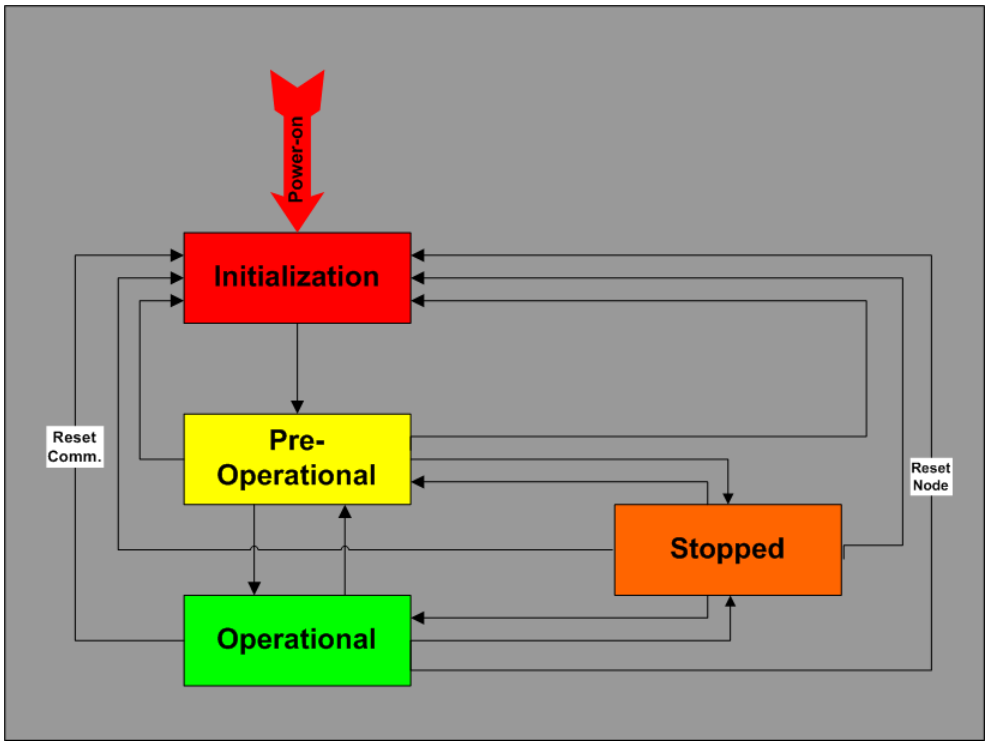


Figure 2: NMT Slave State Machine

After power-on or a reset (via CANOPEN_SLAVE_INITIALIZE_REQ/CNF – Initialization of CANopen Slave) and running through the initialization the state *Pre-operational* is reached automatically.

In *Pre-operational* state,

- No PDO communication is allowed
- SDO communication is allowed (this allows configuration and parameterization)

Switching from *Pre-operational* state to *Operational* state (and vice versa) is done by the NMT Master.

In *Operational* state,

- PDO communication is allowed
- SDO communication is allowed (Access to the Object Dictionary)

In *Stopped* state,

- PDO communication is stopped
- SDO communication is stopped (No access to the Object Dictionary possible)
- Further services are also stopped (Time stamp, EMCY, SYNC).
- However, if active, Node Guarding or life-guarding will be continued.

The Initialization State can be subdivided into 3 sub-states in order to enable a complete or partial reset. This is illustrated by *Figure 3: Initialization State* below:

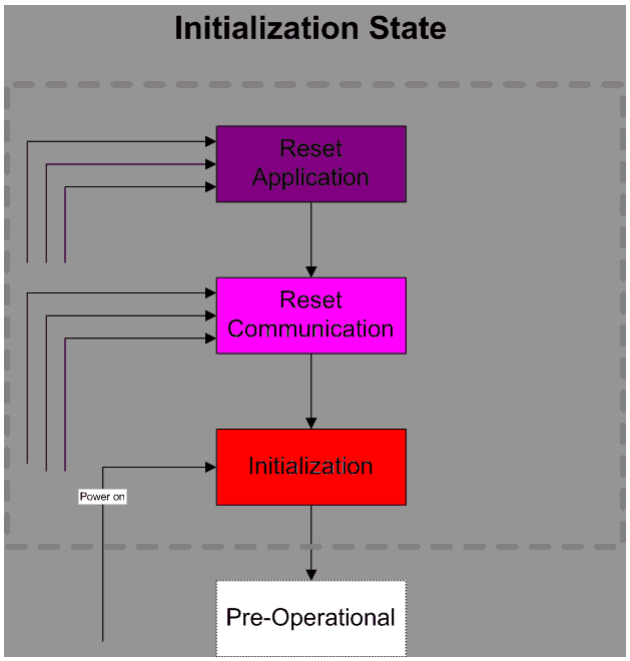


Figure 3: Initialization State of NMT Slave

The Initialization Sub-state (red rectangle in *Figure 3*) performs the basic initialization actions. After power-on, exactly this state is reached and automatically these actions are performed. When all actions are performed, the slave finally reaches the *Pre-operational* state.

The Reset Communication Sub-state provides the possibility to perform the basic device initialization. The parameters of the communication profile are set to their power-on values. After this has been finished, the slave switches to Initialization Sub-state and proceeds with the basic initialization actions.

The Reset Application sub-state provides the possibility to perform the profile initialization. The parameters of the manufacturer-specific profile area and of the standardized device profile area are set to their power-on or default values. After this has been finished, the slave switches to Reset Communication Sub-state.

- The following mapping between the state of the CANopen Slave V3 State Machine and the communication status (0x14 in Common Status) is as follows:
- If the CANopen Slave V3 Protocol Stack is in state *Pre-operational*, the communication state is set to IDLE.
- If the CANopen Slave V3 Protocol Stack is in state *Stopped*, the communication state is set to STOP.
- If the CANopen Slave V3 Protocol Stack is in state *Operational*, the communication state is set to OPERATIONAL.

NMT also defines 5 services which allow changing the state to the states

- *Operational* (Service name: Start)
- *Stopped* (Service name: Stop)
- *Pre-operational* (Service name: Enter Pre-operational)

and the following two sub-states of state *Initialization*

- *Reset Application Sub-state* (Service name: Reset Node)
- *Reset Communication Sub-state* (Service name: Reset Communication)

See *Table 56: NMT States* on page 76 for more information concerning this topic.

The CANopen Slave protocol stack supports this with the following two packets:

- Packet CANOPEN_SLAVE_SET_NMT_STATE_REQ/CNF – Set NMT State allows your application to set the NMT state (and thus, to perform these 5 services mentioned above).
- Packet CANOPEN_SLAVE_NMT_STATE_CHANGE_IND/RES – NMT State Change Indication informs you whenever the NMT Master requests a state change and gives you the possibility to react appropriately.

2.3.2 Communication Objects, COB-IDs and Priority of Processing

The CAN standard defines 2048 Communication Objects (COBs). A COB can contain at maximum 8 bytes of data. It represents a unit of data transportation in a CAN network. All data transport is done exclusively via COBs.

Communication Objects provide specific functionalities defined in the CANopen specification such as

- Process Data Objects
- Service Data Objects
- Synchronization Object
- Emergency Object
- Time-Stamp Object
- Boot-up Object
- Network Management Objects such as Node Guarding, Life-Guarding or Heartbeat Objects

which will be explained later in this document

In order to uniquely identify the COBs within the CAN network, the available COBs are numbered from 0 to 2047. These 11 bit numbers uniquely identifying the COBs are called CAN identifiers. The CAN identifier of a COB is stored together with some bits (valid/invalid bit, remote frame support bit, frame format bit) as an object within the object dictionary. This object mainly containing the COB's CAN identifier is also called the COB-ID (i.e. COB-IDentifier). However, the term COB-ID is also often used for the single CAN identifier related to the COB. This manual also uses the term COB-ID in this sense.

The COB-ID plays an important role in CAN bus arbitration: (in the MAC layer): It determines the priority of processing for the COB. Lower values of the COB-ID mean higher priority.

2.3.2.1 Predefined Connection Set

The designer of a CANopen network must assign the COB-IDs to the various objects. He has a lot of freedom to do so. In order to easily setup smaller networks, there is a set of recommended and good running assignments provided by the specification: the predefined connection set. A factory new CANopen device will work with this predefined connection set when reaching the PRE-OPERATIONAL state for the first time (and, if no explicit changes have been stored, every time it reaches this state).

The predefined connection set works as follows:

The 11 bit CAN identifiers (COB-IDs) are separated in to a leading part of 4 bits denominated as the function code (Bits 10 to 7) and the trailing 7 bits (Bits 6 to 0) which represent the Node ID holding the addressing information within the CAN network.

The predefined connection set supports the following communication objects:

- 4 Receive-PDOs
- 4 Transmit-PDOs
- 1 SDO
- 1 Emergency Object
- and the NMT objects.

The following table shows which function codes (and thus ranges of COB-IDs) are assigned to which objects within the predefined connection set.

Additionally, it contains

- the index within the object dictionary where to adjust the communication parameters of that object
- the information whether this object is a broadcast object (communication to all participants on the network) or a peer-to-peer object (communication between master and slave or slave and slave).

Object	Function code (binary notation)	Resulting Range of COB-IDs	Index of communication parameters	Broadcast/Peer-to-Peer
NMT	0000	0	-	Broadcast
SYNC	0001	128	0x1005—0x1007	Broadcast
EMERGENCY	0001	129-255	0x1014—0x1015	Peer-to-Peer
TIME STAMP	0010	256	0x1012—0x1013	Broadcast
PDO1(tx)	0011	385-511	0x1800	Peer-to-Peer
PDO1(rx)	0100	513-639	0x1400	Peer-to-Peer
PDO2(tx)	0101	641-767	0x1801	Peer-to-Peer
PDO2(rx)	0110	769-895	0x1401	Peer-to-Peer
PDO3(tx)	0111	897-1023	0x1802	Peer-to-Peer
PDO3(rx)	1000	1025-1151	0x1402	Peer-to-Peer
PDO4(tx)	1001	1153-1279	0x1803	Peer-to-Peer
PDO4(rx)	1010	1281-1407	0x1403	Peer-to-Peer
SDO (tx)	1011	1409-1535	0x1200	Peer-to-Peer
SDO (rx)	1100	1537-1663	0x1200	Peer-to-Peer
NMT Error Control	1110	1793-1919	0x1016—0x1017	Peer-to-Peer

Table 11: Objects of the Predefined Connection Set (seen from Point of View of the Device)

The following COB-IDs are restricted and may therefore not be used by all configurable COBs and by PDO, SDO, SYNC, TIME STAMP and EMCY:

COB-ID	Cause
0	Used by NMT Service, fixed assignment
1	Reserved
257-384	Reserved for Safety-relevant Data Objects (in CANopen Safety Framework)
1409-1535	Used by default SDO (tx) , fixed assignment
1537-1663	Used by default PDO (rx) , fixed assignment
1760	Reserved
1793-1919	Used by NMT Error Control, fixed assignment
2020-2047	Reserved

Table 12: COB-IDs with Restrictions of Use

2.3.3 Relation between Communication Objects and NMT States

The following table provides the matrix which kinds of communication objects may exist during which states:

State / Object	INITIALISING	PRE-OPERATIONAL	OPERATIONAL	STOPPED
SDO				
SYNC				
TIME STAMP				
EMCY				
Boot-up object				
NMT objects				

- Green fields indicate the object may operate in this state.
- Red fields indicate the object cannot operate in this state.

2.3.4 Events

2.3.4.1 NMT State Change Events

NMT State Change Events happen every time the NMT Slave state changes. For a description of the NMT Slave State Machine, see section 2.3.1 *“NMT Slave State Machine”* on page 12.

Various reasons may have caused this change of NMT state:

- External request by the CANopen Master via NMT object for module control services. This is the most usual case.
- Internal request; an application event initiated a module control service (for instance, a CANOPEN_SLAVE_SET_NMT_STATE_REQ/CNF – Set NMT State request.)
- A hardware reset occurred.

The NMT State Change Event is related to the following packets of the CANopen Slave protocol stack:

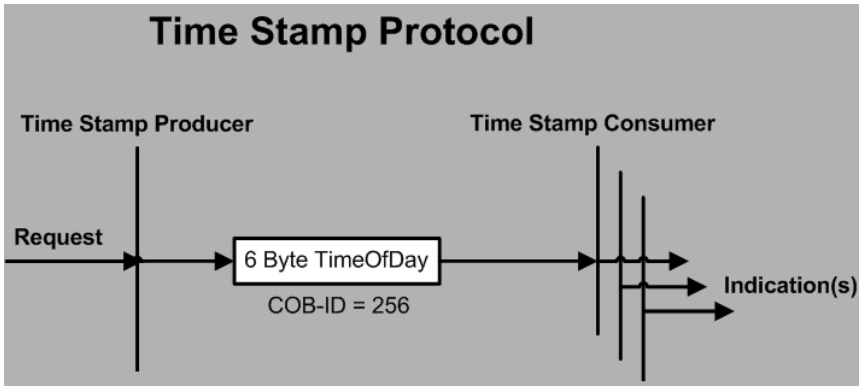
- CANOPEN_SLAVE_SET_NMT_STATE_REQ/CNF – Set NMT State for setting the state of the NMT Slave State Machine from your application.
- CANOPEN_SLAVE_NMT_STATE_CHANGE_IND/RES – NMT State Change Indication for making your application aware of state changes in the NMT Slave State Machine.
- CANOPEN_SLAVE_NMT_COMMAND_IND/RES – NMT Command Indication for enabling the host application to react to the requests for Module Control Services issued by the CANopen Master.

2.3.4.2 Time Stamp Events

The time stamp service allows to send time stamps from one node (CANopen Master or Slave) to another one. In terms of the Producer Consumer Model, a CANopen Slave can act (within a push model) both as a producer and as a consumer,

In either case, the data transferred between producer and consumer are the current calendar date and time divided in to a milliseconds part and a days part. These are transferred within one communication object with a 6 Byte on the CANopen network. The time stamp communication object is usually assigned to the COB-ID 256.

For details of specifying the date and time parameters see the descriptions of the packets mentioned below.



The Time Stamp Event is related to the following packets of the CANopen Slave protocol stack:

- CANOPEN_SLAVE_SEND_TIME_STAMP_REQ/CNF – Send Time Stamp allows your application to send a time stamp according to the CANopen time stamp protocol. In this case, the CANopen Slave acts as a producer. A time stamp event is caused at the side of the consumer(s) (recipient(s) of the time stamp message.
- CANOPEN_SLAVE_RECV_TIME_STAMP_IND/RES – Receive Time Stamp Indication for making your application aware of the reception of a time stamp sent by the Time Stamp Producer via the Time Stamp Protocol. In this case, the CANopen Slave acts as a consumer. Here, the time stamp event occurs at the CANopen Slave and needs to be handled there by your application.

2.3.4.3 NMT Error Control Events

One of the tasks of Network Management (NMT) is the detection of failures within the CANopen network such as absent stations no more regularly transmitting PDOs. For this purpose, NMT provides the three NMT Error Control Events offering different error control services based on the periodic transmission of messages.

The CANopen Slave protocol stack supports the following kinds of NMT Error Control Events:

- Node Guarding / Life Guarding
- Heartbeat

It is mandatory to support one of these services. (However, a supported service may be deactivated at rub-time.)

The NMT Error Control Events are related to the following packet of the CANopen Slave protocol stack:

- CANOPEN_SLAVE_ERR_CTRL_EVENT_IND/RES – Error Control Event Indication for making your application aware of changes of the error control state within the entire CANopen network.

Node Guarding

The concept of node guarding consists mainly of the NMT Master maintaining a database containing the expected states of all connected NMT Slave devices (besides other information) and comparing these with their actual states which are regularly requested from the NMT Slave devices.

The Node Guarding Protocol works as follows:

The NMT Master cyclically polls in order to check whether the NMT Slave is still present at the bus. This polling is denominated as the guarding request. Actually, this is accomplished by sending a CAN remote frame to the NMT Slave.

The NMT Slave reacts by sending a CAN Data Frame with 1 Byte data. The most significant bit is toggled every time, i.e. it is 0 when it was 1 last time and vice versa. Bits 0 to 6 are used to transmit the state to the NMT State.

This is illustrated in *Figure 4: Node Guarding Protocol*.

The time between two guarding requests of the Master is denominated as the guard time.

Node Guarding can take place during the following states of the slave:

- Pre-operational
- Operational
- Stopped

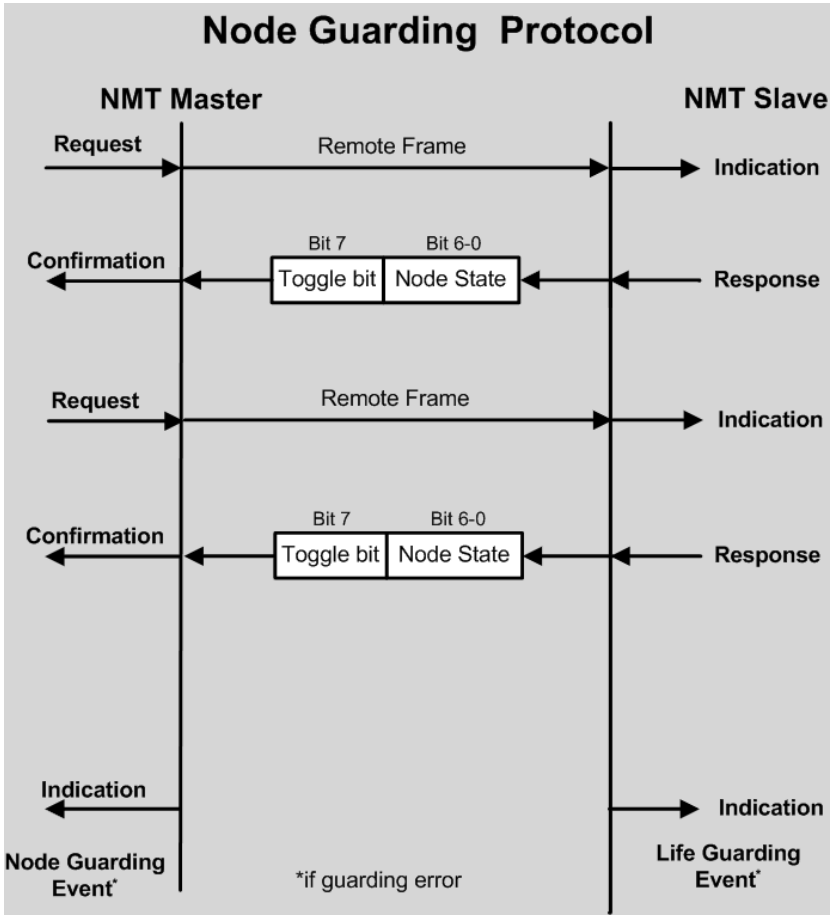


Figure 4: Node Guarding Protocol

Life Guarding

Additionally, the slaves try to detect possible failure of the Master. If they do not receive guarding requests within an (adjustable) time, they recognize this as failure of the Master (life-guarding event). This time is denominated as life time of the CANopen node. It must be longer than the guard time. The life time is usually specified as life-time factor which is then multiplied with the guard time in order to determine the life-time of the NMT Slave.

The guard time can be specified in object 0x100C of the object dictionary. The life time factor can be adjusted in object 0x100D. Both values set to 0 indicates the guarding feature is currently disabled. (In this case, it would be mandatory to use the Heartbeat feature described subsequently.

Heartbeat

Alternatively, there is another error control service available which avoids sending remote frames: the Heartbeat Mechanism.

In the CANopen network, there may be one Heartbeat Producer cyclically sending heartbeat requests. The time between to such Heartbeat requests is denominated as the *Heartbeat Producer Time*.

There may be multiple Heartbeat Consumers within the CANopen network on which the Heartbeat signal is received and causes an indication. The Heartbeat Consumer supervises these indications. If in an (adjustable) time no such indication occurs, the Heartbeat Consumer assumes the Heartbeat Producer has failed and indicates a Heartbeat event. This time is denominated as Heartbeat Consumer Time. It is specified in multiples of 1 ms in object 0x1016 of the object dictionary.

The Producer Heartbeat Time set to 0 indicates the Heartbeat feature is currently disabled. (In this case, it would be mandatory to use the Node Guarding feature described above.

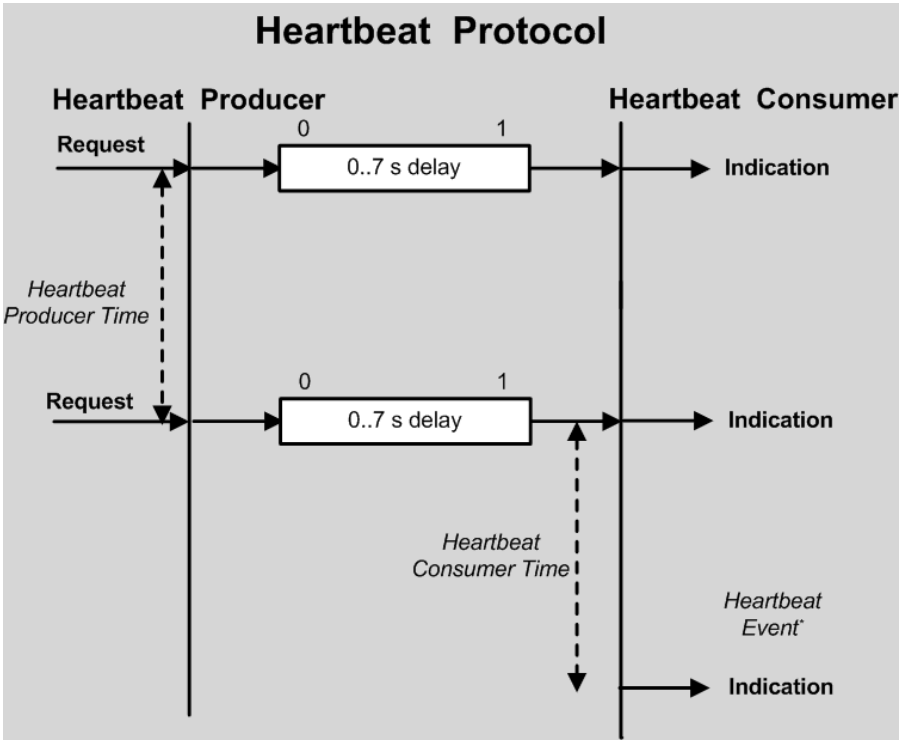


Figure 5: Heartbeat Protocol

Each CANopen device must implement Node Guarding/life-guarding or the heartbeat mechanism or both. If both are implemented, they may not both be active at the same time. It is recommended to use the heartbeat mechanism.

The CANOPEN_SLAVE_ERR_CTRL_EVENT_IND/RES – Error Control Event Indication informs you whenever anode guarding or heartbeat error occurs on the CANopen network or a CANopen Slave on the network starts or stops supervision by node guarding/life guarding or by heartbeat.

2.3.4.4 Receive PDO Events

The PDO service allows to send application objects synchronously or asynchronously from one node (CANopen Master or Slave) to another one. In terms of the Producer Consumer Model, a CANopen Slave can act within a push model as a producer, and within a pull model both as a producer and as a consumer

The Receive PDO Event is related to the following packets of the CANopen Slave protocol stack:

- CANOPEN_SLAVE_SEND_TXPDO_REQ – Send TxPDO Request for sending a TxPDO to one or more communication partner(s) on the CANopen network and causing a Receive PDO Event there.
- CANOPEN_SLAVE_RECV_RXPDO_REQ/CNF – Receive RxPDO Request for sending an RTR frame in order to request an RxPDO from a communication partner on the CANopen network.
- CANOPEN_SLAVE_RECV_RXPDO_IND/RES – Receive RxPDO Indication for informing your application about the reception of a PDO (coming from another communication partner on the CANopen network).

The protocols of the 3 packets are illustrated at their packet descriptions.

As parameters, the requests contain an array of up to 16 affected PDO numbers. The confirmation packet contains a status code indicating whether or not the processing of the PDO has been successful. In case of failure you can evaluate this status code as error code for determining the cause of the error.

For details of PDOs, see section *Process Data Objects (PDO)* on page 26.

2.3.4.5 NMT Command Events

NMT Command Events are used to inform the CANopen Slaves that the NMT Master requests a change of the Slave's NMT state.

The NMT Command Event service allows to NMT State change requests from the CANopen Master to one or more CANopen Slaves using the NMT Protocol. In terms of the Producer Consumer Model, a CANopen Slave acts within a push model as a consumer. The CANopen Master acts as producer.

When receiving an NMT command request, the application should react in the following manner:

1. Determine which service needs to be executed
2. Execute requested service
3. Send response packet.

For determining which service should be executed, the value of variable `ulNmtCommand` must be evaluated according to the following table:

Value	Requested Service	Action to take
1	Start CANopen Slave	Set state to *Operational*
2	Stop CANopen Slave	Set state to *Stopped*
128	Enter state *Pre-operational*	Set state to *Pre-operational*
129	*Reset node*	Set state to *Initialising*, sub-state *Reset node*
130	*Reset communication*	Set state to *Initialising*, sub-state *Reset communication*

Table 13: NMT States

If the application has been previously registered for this event, the automatic switching to the new state will be inhibited. For switching to the new local NMT state, the application is responsible exclusively. You can then use the new local NMT state of the CANopen Slave after service execution as value for parameter `ulNmtState` in the response packet.

If the application has not been registered for this event previously, switching to the new state will be done automatically.

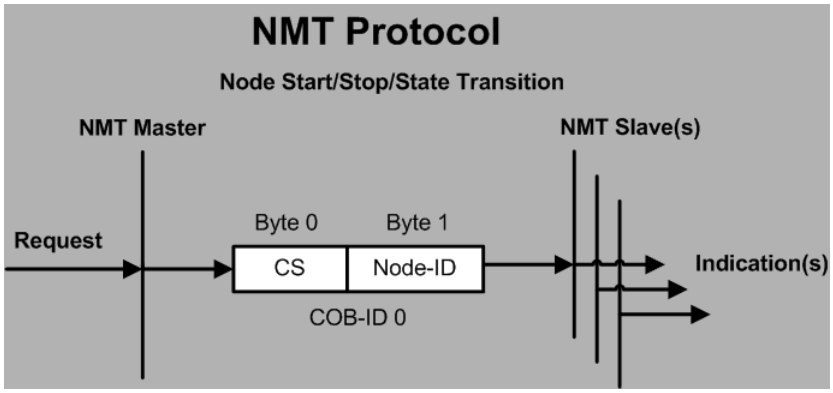


Figure 6: Node Start/Stop State Transition

The parameters are transferred within one communication object with 2 Byte on the CANopen network. The NMT communication object is always assigned to the COB-ID 0 which is the COB-ID with the highest priority at all.

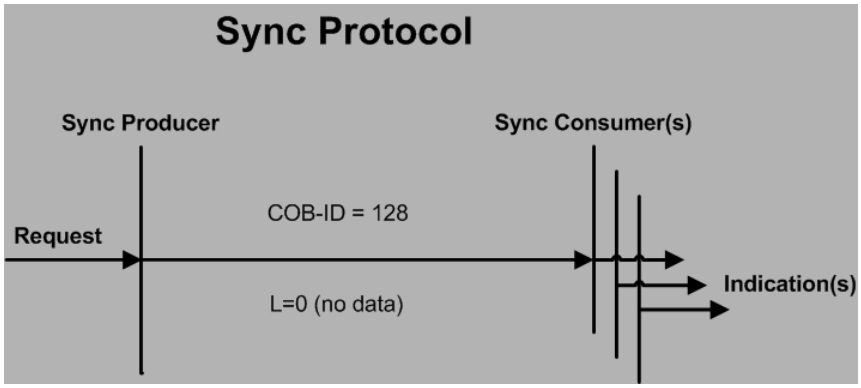
The NMT Command Events are related to the following packets of the CANopen Slave protocol stack:

- CANOPEN_SLAVE_NMT_COMMAND_IND/RES – NMT Command Indication for making your application aware of external requests for a state change in the NMT Slave State Machine. The NMT Command Event occurs at the CANopen Slave.
- CANOPEN_SLAVE_NMT_STATE_CHANGE_IND/RES – NMT State Change Indication indicating a change of the state within the CANopen Slave's NMT State Machine and allowing the application to react and perform all necessary application-internal changes required after the change of NMT state.

2.3.4.6 Synchronization Event (SYNC)

The Synchronization Event is used to synchronize PDO processing with a highly precise time signal.

This allows performing synchronous PDOs (Cyclic and acyclic). For details on synchronous PDOs, see section "*Process Data Objects (PDO)"* on page 26. The part of the CANopen Slave device that provides the SYNC signal is denominated as the SYNC Producer. SYNC Consumers are all parts of the Slave which perform synchronized data processing.



The Synchronization Event does not transfer any data, therefore the signal is transferred within one communication object with a data length of 0 Byte (or 1 Byte) on the CANopen network.

The recommended COB-ID for the NMT communication object is 128. This is the same value as used in the predefined connection set.

The object dictionary contains three relevant values concerning the SYNC event.

- The actual value of the COB-ID of the SYNC object can be retrieved from object 0x1005 within the object dictionary.
- Object 0x1006 contains the communication cycle time. This value represents the time between two subsequent SYNC events (this object is not supported by the CANopen Slave V3 protocol stack).
- Object 0x1007 contains the synchronous window length. The synchronous window length specifies the time within which the synchronous PDOs may be processed. Its value must be smaller than the applied communication cycle time. (this object is not supported by the CANopen Slave V3 protocol stack)

The SYNC Event is related to the working with synchronous PDOs, see section *Synchronous vs. asynchronous Data Transmission* on page 29.

2.3.4.7 Emergency Event (EMCY)

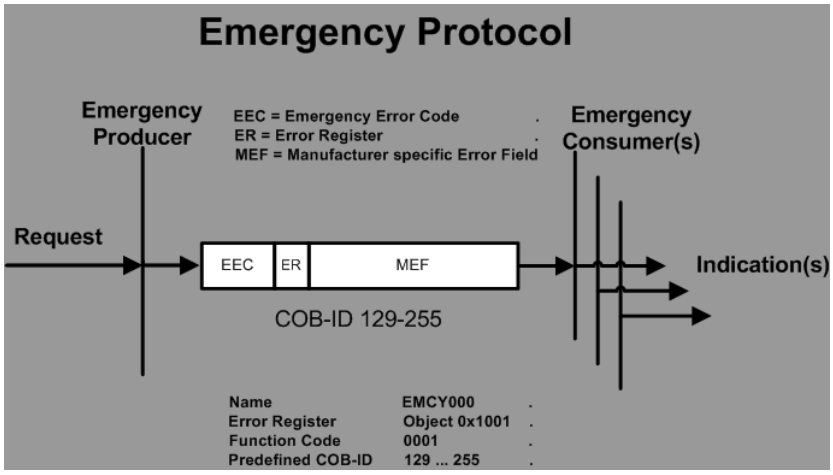


Table 14: Emergency Protocol

The parameters are transferred within one communication object on the CANopen network. The NMT communication object is usually assigned to a COB-ID in the range between 129 and 255.It delivers

- An emergency error code
- An error register
- 5 bytes of manufacturer-specific error information

The Emergency Event (EMCY) is related to the following packets of the CANopen Slave protocol stack:

- CANOPEN_SLAVE_SEND_EMCY_REQ/CNF – Send Emergency Message allowing you to send an emergency telegram from your application to any consumer within the CANopen network.

2.3.5 Process Data Objects (PDO)

Process Data Objects (short: PDO) provide the fastest way for data transmission in CANopen. One single PDO can be considered as a segment of the whole process data with a length of 8 Bytes. On low level (CAN-DL), each PDO is transmitted as a single CAN telegram, and the length of CAN telegrams is limited to 8 Bytes. Due to this fact there is an 8 byte limit of the PDO length.

One of the most important advantages of the PDO concept is that PDOs are transmitted without any protocol overhead. This allows good performance.

2.3.5.1 Producer Consumer Model

PDO communication follows the Producer Consumer Model:

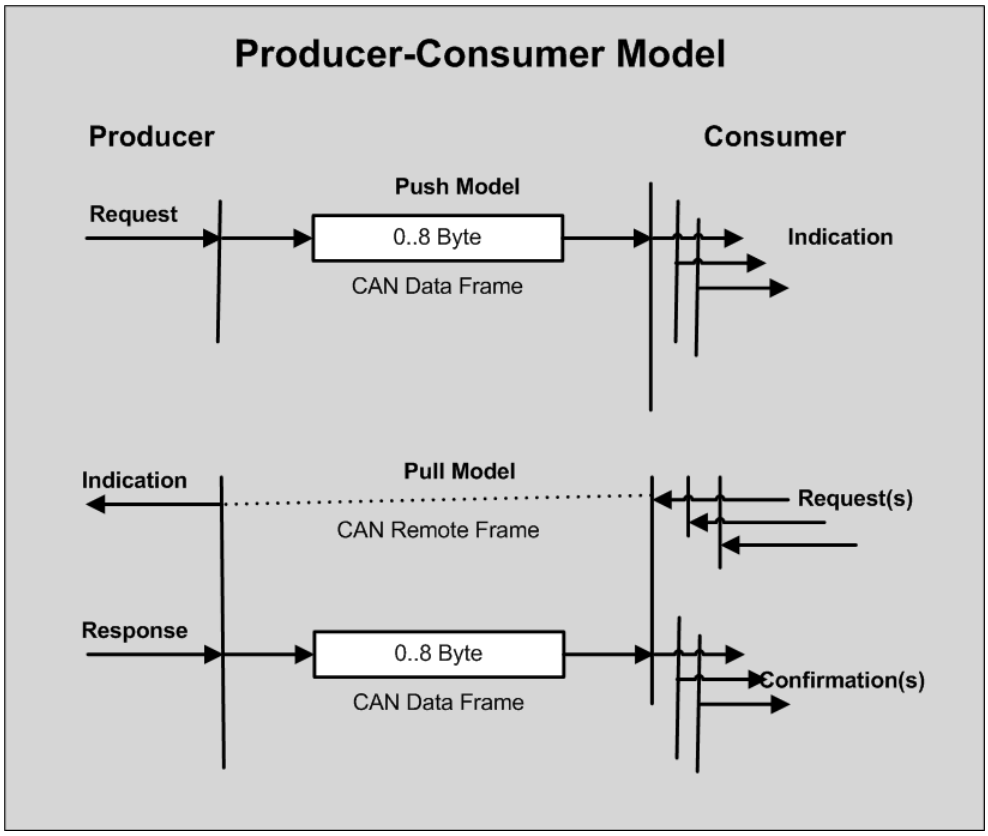


Figure 7: Producer Consumer Model

In the Producer Consumer Model, each node may send data at any time (for instance in case of a pre-defined event happening). Each station may listen to the transmitted messages. When a message is received, the node decides on its own whether the message is accepted, or not.

This offers the advantage of greater flexibility and easy establishment of event-driven processing for instance compared to the Master-Slave Model.

Data transmission is done from a device (the *Producer*) to one or more other devices (the *Consumer(s)*) without any confirmation (non-confirmed mode).The broadcast mechanism of the CAN low-level protocol is used to accomplish this .The assignment is made via identifiers which must match between Producer and Consumer.

2.3.5.2 Services

There are two services available for dealing with PDOs:

- The Write Service

  The Write Service is implemented by the packet CANOPEN_SLAVE_SEND_TXPDO_REQ – Send TxPDO Request. It acts on the Transmit-PDOs (TxPDOs).

- and the Read Service.

  The Read Service is implemented by the packets `CANOPEN_SLAVE_RECV_RXPDO_REQ/CNF` – Receive RxPDO Request and `CANOPEN_SLAVE_RECV_RXPDO_IND/RES` – Receive RxPDO Indication. It acts on the Receive-PDOs (RxPDOs).

Note: The perspective of view to be applied in this context is that of the CANopen device. An IO device acting as CANopen Slave would send its input data via its TxPDOs and receive its output data via its RxPDOs.

The CANopen Slave protocol stack limits the number of available TxPDOs and RxPDOs to 32 each (for Hilscher devices with the netX 52 processor) and to 64 each (for any other netX-based Hilscher devices).

The numbers and the length of the PDOs are application-specific and must be defined in the device's profile.

2.3.5.3 Modes of Communication

A PDO can be scheduled in three different ways. It can be driven:

- By an internal event or an event timer
- By a remote request
- Synchronously (cyclic or acyclic)

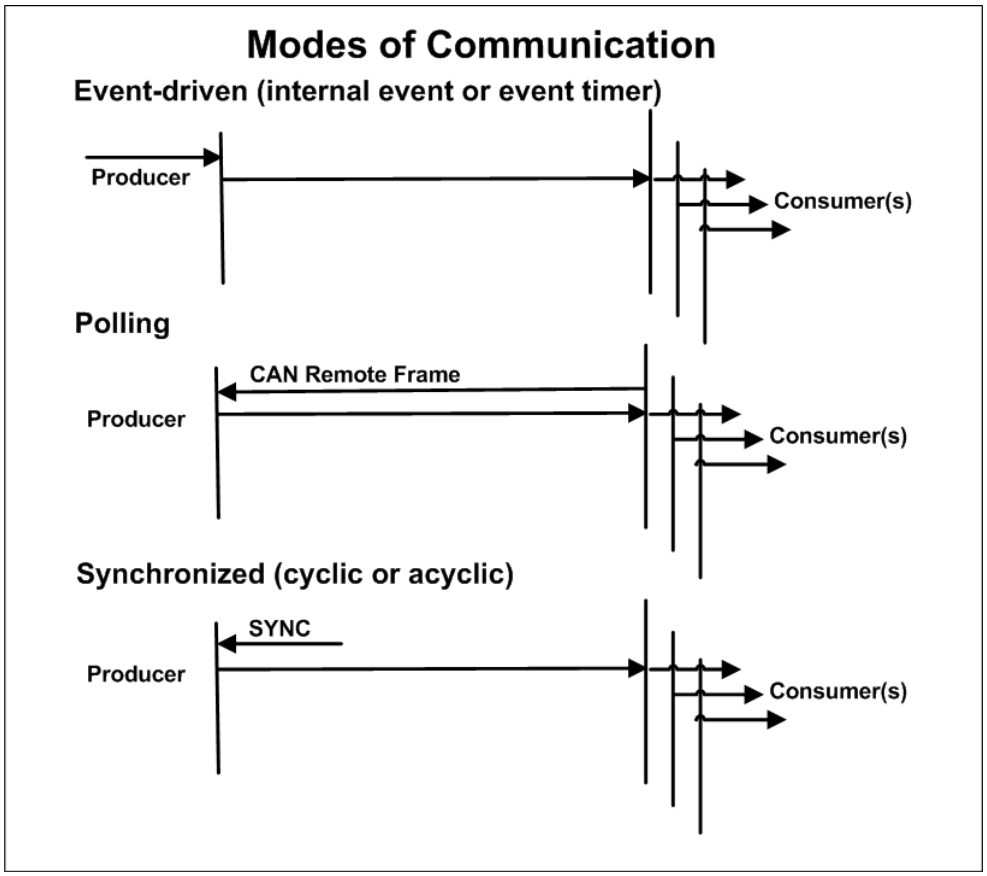


Figure 8: Modes of Communication: Event-driven, Polling, Synchronized

Event-driven communication

A PDO can be transmitted when a specific event that has been previously defined occurs locally at the producer. This communication mode is called event-driven PDO communication. The driving event may be any local event such as a change of an input or elapsing of a specific timer.

This case is equivalent to the push model described above.

Polling by CAN remote frames

A consumer can also stimulate the PDO transmission. This case is equivalent to the pull model described above. This is internally accomplished by sending a RTR Frame to the producer. The producer will then automatically "answer" by transmission of the PDO in a CAN Data Frame.

Synchronized communication

Transmission of a PDO can be triggered periodically (i.e. a repeated and precise time synchronization signal is received). For this purpose, CANopen provides the SYNC object.

The PDO can be coupled to the SYNC object. This forces the CANopen Slave to process the PDOs (TxPDOs and RxPDOs).

Let us now clarify the terms *synchronous*, *asynchronous*, *cyclic* and *acyclic* in context with data transmission

2.3.5.4 Synchronous vs. asynchronous Data Transmission

CAN distinguishes between synchronous and asynchronous data transmission.

- Synchronous data transmission means in this context coupled to a synchronization signal (such as the one supplied by the SYNC object). Synchronous data transmission takes place within a well-defined time window following the SYNC signal. PDOs offer the only mechanism for synchronous data transmission. It is strongly recommended that the priority of synchronous PDOs should be higher than that of the asynchronous PDOs.
- Asynchronous data transmission means in this context event-driven. Asynchronous data transmission can generally take place at any time without restriction.

2.3.5.5 Cyclic and acyclic Data Transmission

CAN also distinguishes between two forms of synchronous data transmission, namely cyclic data transmission and acyclic data transmission.

- Cyclic data transmission means periodic data transmission coupled to the SYNC signal. However, this does not necessary mean one transmission at every SYNC event. You can define the transmission type, i.e. a number of SYNC events to occur between to synchronous data transmissions. This number may be chosen from the range from 1 to 240. Thus transmission type 1 means one transmission at every SYNC event, transmission type 2 means one transmission at every second SYNC event and so on.
- Acyclic data transmission means data transmission triggered by an application-specific event. Message transmission occurs synchronized to the SYNC event but not periodically.

2.3.5.6 Transmission Types

There are also some other transmission types for special cases. The following table introduces the complete rules that apply for the transmission types

Type #	Cyclic	Acyclic	Synch.	Asynch.	RTR only
0		x	x		
1…240[1]	x		x		
241…251			reserved		
252			x		x
253				x	x
254[2]				x	
255[3]				x	

Table 15: PDO Transmission Types

1. This value specifies exactly the number of SNC events between two transmissions of the PDO.
2. Device-specific application event
3. Device-profile defines application event

Transmission types 254 and 255 differ in the following aspect:

- Transmission type 254 is used for device-specific application events.
- Transmission type 255 is used for application events defined in the device profile.

2.3.5.7 Inhibit Time and Event Timer

Additionally, there are two important features that can be assigned only to TxPDOs, namely as the

- the Inhibit Time
- and the Event Timer.

The Inhibit Time is defined as the minimum time required to elapse between two consecutive invocations of the service of a PDO. If this time is set to a value significantly larger than the time need to process the synchronous PDO, this feature gives acyclic or asynchronous PDOs with lower priority than the synchronous PDO better chances to be processed quickly. Values are specified in units of 100 µs. As long as the PDO exists (this is indicated by bit 31 of subindex 1 of the PDO's index being 0), this value must not be changed any more!

The Event Timer

For asynchronous TxPDOs (i.e. such event-driven TxPDOs with transmission types 254 and 255), additionally an Event Timer can be set. Values are specified in units of 1 ms. If the specified time has elapsed, this will be considered as an event and will therefore cause the PDO to be transmitted.

The Inhibit Time may specify the earliest time transmission can take place and, similarly, the Event Timer may specify the latest time transmission can take place. This allows setting up a "virtual transmission window" for the timely execution of the TxPDO's transmission in case of asynchronous. (This feature can be used for reducing bus-load).

2.3.5.8 Communication Parameters

In the object dictionary, there is a PDO communication parameter assigned to each PDO. It describes the communication behavior of the PDO.

The communication parameter can be found at:

- `0x1400-0x15FF` (`0x1400` for the first RxPDO)
- `0x1800-0x19FF` (`0x1800` for the first TxPDO

The next PDO has an index which is increased by one and so forth.

The communication parameter has the following content:

Index #	Subindex	Description	Data Type
0x1400-… 0x15FF (RxPDO) or 0x1800 … 0x19FF (TxPDO)	0x00	Number of entries	Unsigned8
	0x01	CANopen identifier (COB-ID)	Unsigned32
	0x02	Transmission Type	Unsigned8
	0x03	Inhibit Time	Unsigned16
	0x04	Reserved	Unsigned8
	0x05	Event Timer	Unsigned16

Table 16: PDO Communication Parameter

2.3.5.9 PDO Mapping

The PDOs provide the interface to the application objects. They are assigned to the entries in the object dictionary. The process of assignment is denominated as PDO mapping and is practically accomplished via a specific mapping structure in the object dictionary.

This mapping structure can be found at:

- `0x1600-0x17FF` (0x1600 for the first RxPDO)
- `0x1A00-0x1BFF` (0x1A00 for the first TxPDO

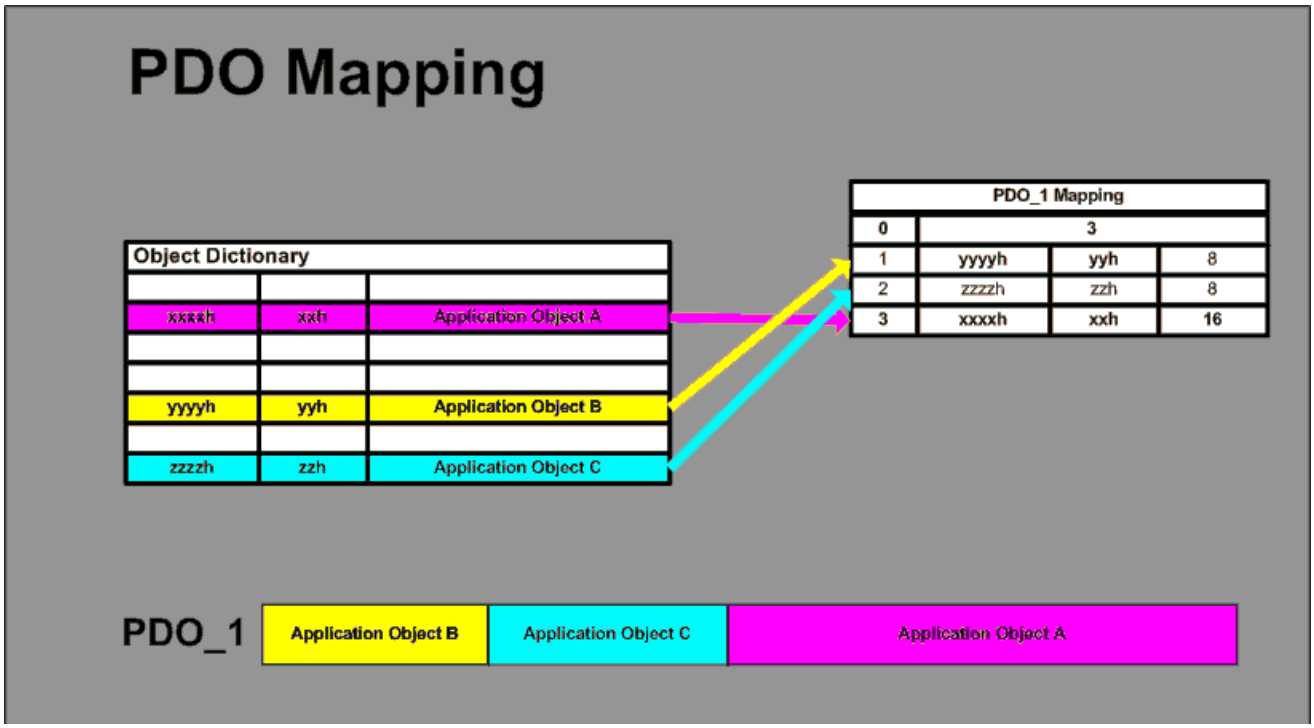


Figure 9: PDO Mapping

Figure 9: PDO Mapping explains the relationship between object dictionary (left upper part), PDO mapping structure (right upper part) and the resulting PDO containing the application objects to be mapped (lower part).

One entry in the PDO mapping table requires 32 bit. It consists of:

- 16 bit containing the index of the object dictionary entry containing the application object to be mapped
- 8 bit containing the subindex of the object dictionary entry containing the application object to be mapped
- 8 bit containing the length information.

The PDO mapping table can at maximum contain 8 of such entries, therefore the number of application objects in one PDO is also limited to 8.

2.4 Standard Mode vs. Extended Mode

The CANopen Slave V3 Protocol Stack offers two modes:

- *Standard Mode* (Object-based DPM content)
- *Extended Mode* (PDO based DPM content)

2.4.1 How to decide between Operation in Standard Mode and Extended Mode

The choice of the mode has a large influence on the behavior, the internal operation and the configuration facilities of the stack. The choice which one of these to use should be made according to the following rules:

Standard Mode

1. If your intention is to run existing applications from CANopen Slave Stack V2, you should definitely decide to use *Standard Mode*. Running existing applications with only slight changes is possible as *Standard Mode* offers a high degree of compatibility to the CANopen Slave Stack V2.
2. Also, if you want to avoid having to fill in larger amounts of objects into the object dictionary, it is recommended to use the *Standard Mode*.

Extended Mode

On the other hand, it is recommended to use the *Extended Mode*, if you want to implement a complete device profile, if you want to work with an object dictionary of your own, if you want to map objects which had been added to the object dictionary by your application or if you do not have any legacy applications. Only *Extended Mode* allows to fully exploit the new possibilities of the CANopen Slave V3 Protocol Stack.

2.4.2 Where can I switch between Standard Mode and Extended Mode?

If the Extended Mode Flag (= BIT28 of the CANopen flags in `CANOPEN_SLAVE_BUSPARAM_DATA_T`) is set to TRUE (i.e. set to 1), then the CANopen Slave V3 Protocol Stack is in *Extended Mode*. If this flag is set to FALSE (i.e. set to zero), then the CANopen Slave V3 Protocol Stack is in *Standard Mode*. For more information, see *Table 26: Bus parameter structure CANOPEN_SLAVE_BUSPARAM_DATA_T*.

Let us now explain in detail the most important differences between *Standard Mode* and *Extended Mode*.

2.4.3 Standard Mode

The *Standard Mode* is characterized by the following features:

- The *Standard Mode* has been designed for maximum compatibility to existing applications for CANopen Slave V2.
- The bus parameter structure CANOPEN_SLAVE_STD_BUSPARAM_DATA_T is applied, see *Table 26: Bus parameter structure CANOPEN_SLAVE_BUSPARAM_DATA_T* on page 44.
- Configuration can be done with tools (System configurator *SYCON.net* or *netX Configuration Tool*) or by *"Set Configuration"* packet.
- The DPM contains the content of the objects 0x2000 to 0x2003 (TxPDOs) and 0x2200 to 0x2203 (RxPDOs).
- The object dictionary is completely filled with objects up to the maximum extent supported at that time. According to the entries made in the EDS file, all entries are automatically created within the object dictionary. For a precise list of supported entries, see subsection 2.4.5 "*Object Dictionary with Firmware Functionality*" on page 38 below.
- If further objects are added later, these cannot be PDO-mapped in *Standard Mode*.

2.4.3.1 Configuring the Standard Mode

The standard bus parameter structure CANOPEN_SLAVE_STD_BUSPARAM_DATA_T is used. It consists of the following items:

Parameter	Type	Meaning	Range of Values/ Default
ulVendorId	UINT32	Vendor code if corresponding bit in parameter ulCanopenFlags is set	
ulProductCode	UINT32	Product code if corresponding bit in parameter ulCanopenFlags is set	
ulSerialNumber	UINT32	Serial number if corresponding bit in parameter ulCanopenFlags is set	
ulRevisionNumber	UINT32	Revision number if corresponding bit in parameter ulCanopenFlags is set	
ulDeviceType	UINT32	Device Type if corresponding bit in parameter ulCanopenFlags is set	
bObject2000Size	UINT8	Size of object 0x2000	
bObject2001Size	UINT8	Size of object 0x2001	
bObject2002Size	UINT8	Size of object 0x2002	
bObject2003Size	UINT8	Size of object 0x2003	
bObject2200Size	UINT8	Size of object 0x2200	
bObject2201Size	UINT8	Size of object 0x2201	
bObject2202Size	UINT8	Size of object 0x2202	
bObject2203Size	UINT8	Size of object 0x2203	
usNumOfRxPdo	UINT16	Number of Receive PDOs	0...256
usNumOfTxPdo	UINT16	Number of Transmit PDOs	0...256
aulReserved[2]	UINT32[2]	Reserved, set to zero	

Table 17: Standard bus parameter structure CANOPEN_SLAVE_STD_BUSPARAM_DATA_T

The ulVendorId parameter provides the value for the Vendor ID entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 1) if bit 4 in parameter ulCanopenFlags is set. Otherwise, the vendor ID is set to zero.

The `ulProductCode` parameter provides the value for the product code entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 2) if bit 5 in parameter `ulCanopenFlags` is set. Otherwise, the product code is set to zero.

The `ulSerialNumber` parameter provides the value for the serial number entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 4) if bit 6 in parameter `ulCanopenFlags` is set. Otherwise, the serial number is set to zero.

The `ulRevisionNumber` parameter provides the value for the revision number entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 3) if bit 7 in parameter `ulCanopenFlags` is set. Otherwise, the revision number is set to default.

2.4.3.2 Handling of Process Data in Standard Mode

The CANopen slave implementation V3 provides 4 objects (0x2000…0x2003) for send data and 4 objects of receive data (0x2200…0x2203); each object has up to 256 bytes of process data (Standard: 128 bytes, minimum: 0 bytes) and is transferred via PDO according to the active network configuration. For netX 52: each object has up to 64 bytes of process data (Standard: 64 bytes, minimum: 0 bytes)

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

Mapping of Input and Output Image to Send and Receive Objects

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

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

Table 18: Mapping of Input Data (in case of unchanged object lengths and no netX 52 applied)

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

Table 19: Mapping of Output Data (in case of unchanged object lengths and no netX 52 applied)

DPM input image byte offset	Receive object index	Receive object sub-index
0	2200h	01h
1	2200h	02h
..	..	..
64	2200h	40h
65	2201h	01h
..	..	..
254	2203h	3Fh
255	2203h	40h

Table 20: Mapping of Input Data (in case of unchanged object lengths and netX 52 applied)

DPM output image byte offset	Send object index	Send object sub-index
0	2000h	01h
1	2000h	02h
..	..	..
64	2000h	40h
65	2001h	01h
..	..	..
254	2003h	3Fh
255	2003h	40h

Table 21: Mapping of Output Data (in case of unchanged object lengths and netX 52 applied)

2.4.4 Extended Mode

The *Extended Mode* is characterized by the following features:

- The *Extended Mode* has been designed for maximum flexibility, for instance, if you want to create an object directory of your own or you want to create your own device profile.
- The extended bus parameter structure CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T is applied, see *Table 22: Extended bus parameter structure CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T* on page 37.
- The AUTOSTART option is not available.
- Configuration can only be done by *"Set Configuration"* packet. Currently, there are no suitable configuration tools available.
- The DPM contains the PDOs (DPM Byte 0…7: PDO1, DPM Byte 8…15: PDO2, and so on.).
- The object dictionary must completely be set up manually. For each of the entries made in the EDS file that you want to use, the according entries must be made in the object dictionary (via ODV3/object create and write). For a precise list of entries that can be set up, see subsection 2.4.5 "*Object Dictionary with Firmware Functionality"* on page 38 below.
- If further objects are added later, these may be mapped in *Extended Mode*.

2.4.4.1 Configuring the Extended Mode

The extended bus parameter structure CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T is used.

It consists of the following items:

Parameter	Type	Meaning	Range of Values
usNumOfRxPdo	UINT16	Number of Receive PDOs	0.. 256
usNumOfTxPdo	UINT16	Number of Transmit PDOs	0.. 256
aulReserved[9]	UINT32[9]	Reserved, set to zero	

Table 22: Extended bus parameter structure CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T

2.4.4.2 Handling of Process Data in Extended Mode

The 4 objects (0x2000…0x2003) for send data and 4 objects of receive data (0x2200…0x2203) are not used in *Extended Mode*. In the *Extended Mode*, the DPM contains PDOs instead of objects.

2.4.4.3 Start Sequence of Extended Mode

The following sequence of steps is necessary to get the *Extended Mode* running after the bus parameters have been set:

1. Create all necessary objects within the object dictionary. Only objects mentioned in Table 23: Supported Object Dictionary Entries (Communication Profile, present in the EDS file) below make sense in this context.
2. Register all required indications (however, a later registration is also possible).
3. Start communication. This causes a verification whether sufficient information is available to start up the protocol stack. For example, if 3 PDOs are configured, the objects 0x1400, 0x1600, 0x1401, 0x1601, 0x1402 and 0x1602 must exist and will be checked for existence.
4. Then, no communication objects may be deleted anymore.

Note: In *Extended Mode*, configuration can only be done by *"Set Configuration"* packet. However, the auto-start option is not available.

2.4.5 Object Dictionary with Firmware Functionality

The following contains a list of object dictionary entries with Firmware Functionality. These should be exactly the ones mentioned within the EDS file. Only these object dictionary entries can be managed by the CANopen Slave V3 Protocol Stack.

However, this is done in a quite different manner in *Standard Mode* and in *Extended Mode.*

- In *Standard Mode;* all these entries are created by the CANopen Slave by default and are available. They may not be deleted.
- In *Extended Mode,* these entries will be supported by the firmware if you create them manually using the ODV3 capabilities (object create/write). It does not make sense to create other entries of the communication profile unless you completely implement the functionality of such an entry on your own.

Now, this is the list of object dictionary entries with firmware functionality:

Index of Object	Name of Object	Comment
0x1000	Device type	This object is mandatory
0x1001	Error register	This object is mandatory
0x1005	COB-ID SYNC	Consumer only
0x100C	Guard Time	
0x100D	Life Time Factor	
0x1012	COB-ID Time Stamp	Producer and consumer
0x1014	COB-ID EMCY	Producer
0x1015	Inhibit time Emergency	
0x1016	Heartbeat Consumer Entries	Contains up to 64 entries
0x1017	Producer Heartbeat Time	
0x1018	Identity Object	This object is mandatory
0x1029	Error Behavior	The following error behaviors have between implemented in the CANopen Slave V3 protocol stack: *Remain* (no change of NMT state) *Stopped* *Pre-operational* (if operational)
0x1200	Server SDO Parameter 1	
0x1400…143F	RxPDO Communication Parameters	Up to 8 mappable objects per PDO
0x1600…163F	RxPDO Mapping	Up to 8 mappable objects per PDO
0x1800…183F	TxPDO Communication Parameters	Up to 8 mappable objects per PDO
0x1A00.. 1A3F	TxPDO Mapping	Up to 8 mappable objects per PDO

Table 23: Supported Object Dictionary Entries (Communication Profile, present in the EDS file)

Additionally, only for the *Standard Mode*, the following objects of the manufacturer-specific profile are available:

Index of Object	Name of Object	Default (netX 50/51/100/500)	Default (netX 52)	Minimum	Maximum
0x2000	Bytes in (1)	128 Bytes	64 Bytes	0 Bytes	255 Bytes
0x2001	Bytes in (2)	128 Bytes	64 Bytes	0 Bytes	255 Bytes
0x2002	Bytes in (3)	128 Bytes	64 Bytes	0 Bytes	255 Bytes
0x2003	Bytes in (4)	128 Bytes	64 Bytes	0 Bytes	255 Bytes
0x2200	Bytes out (1)	128 Bytes	64 Bytes	0 Bytes	255 Bytes
0x2201	Bytes out (2)	128 Bytes	64 Bytes	0 Bytes	255 Bytes
0x2202	Bytes out (3)	128 Bytes	64 Bytes	0 Bytes	255 Bytes
0x2203	Bytes out (4)	128 Bytes	64 Bytes	0 Bytes	255 Bytes

Table 24: Supported Object Dictionary Entries (Manufacturer-specific Profile, present in the EDS file)

3 The Application Interface

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

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

3.1 Configuration

The user application can use this service in order to configure the AP task with parameters. After the stack receives this request, the AP task will configure the CANopen Slave.

Behavior of the stack when receiving a Set Configuration Command

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

- The configuration packets name is `CANOPEN_APS_SET_CONFIGURATION_REQ` for the request and `CANOPEN_APS_SET_CONFIGURATION_CNF` for the confirmation.
- The configuration data are checked for consistency and integrity.
- In case of failure no data are accepted.
- In case of success the configuration parameters are stored internally (within the RAM).
- The parameterized data will be activated only after a channel init has been performed.
- No automatic registration of the application at the stack happens.
- The confirmation packet `CANOPEN_APS_SET_CONFIGURATION_CNF` only transfers simple status information, but does not repeat the whole parameter set.
- A "Set Configuration Command" will not be processed in case of the protocol stack has already been configured via database prior to receiving this command.

3.1.1 CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration

The Set Configuration Packet contains the parameter system flag parameter, watchdog time parameter and the structure with bus parameters.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag
  CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_T;

#define CANOPEN_APS_SYS_FLAG_COM_CONTROLLED_RELEASE 0x00000001L /* Automatic start    */
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_ENABLED      0x00000002L /* Not supported      */
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_32_BIT       0x00000004L /* Not supported      */

#define CANOPEN_APS_SYS_FLAG_ADDRESS_SWITCH         0x00000010L /* Switch for address */
#define CANOPEN_APS_SYS_FLAG_BAUD_SWITCH            0x00000020L /* Switch for baud    */

#define CANOPEN_APS_WD_OFF                          0x00000000L /* Watchdog disabled */
#define CANOPEN_APS_WD_MIN_TIMEOUT                  0x00000014L /* Minimum watchdog  */
#define CANOPEN_APS_WD_MAX_TIMEOUT                  0x0000FFFFL /* Maximum watchdog  */

__PACKED_PRE struct CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag
{
  TLR_UINT32 ulSystemFlags;               /* System flags  */
  TLR_UINT32 ulWdgTime;                   /* Watchdog time */
  CANOPEN_SLAVE_BUSPARAM_DATA_T tBusParam; /* Bus parameter */

}__PACKED_POST;
```

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENAPS	Destination Queue-Handle of CANopen slave-task Process Queue
ulLen	UINT32	68	Packet Data Length (In Bytes)
ulId	UINT32	0 ... 2^{32}-1	Packet Identification As Unique Number
ulCmd	UINT32	0x00002E04	CANOPEN_APS_SET_CONFIGURATION_REQ - Command
Data			
ulSystemFlags	UINT32		System Flags BIT 0: AUTOSTART / APPLICATION CONTROLLED The start of the device can be performed either application controlled or automatically: **Automatic**: (not available in *Extended Mode*) Network connections are opened automatically without taking care of the state of the host application. **Application controlled**: The channel firmware is forced to wait for the host application to set the Application Ready flag. communication with a controller after a device start is allowed without BUS_ON flag, but the communication will be stopped if the BUS_ON flag changes state to 0 communication with controller is allowed only with the BUS_ON flag. BIT 1: I/O STATUS DISABLED/ENABLED Not supported yet BIT 2: IO STATUS 32 BIT Not supported yet BIT 3: Reserved for further use, set to zero BIT 4: ADDRESS_SWITCH Should be set when hardware address switch is used and there is no TAG present. BIT 5: BAUD_SWITCH Should be set when hardware baudrate switch is used and there is no TAG present. BIT 6 - 31: Reserved for further use, set to zero
ulWdgTime	UINT32	0 20 .. 65535	Watchdog supervision Watchdog supervision deactivated Watchdog time in milliseconds Watchdog time within which the device watchdog must be retriggered from the application program while the application program monitoring is activated. When the watchdog time value is equal to 0 respectively the application program monitoring is deactivated.
tBusParam	CANOPEN_SLAVE_BUSPARAM_DATA_T		Bus parameter structure

Table 25: CANOPEN_APS_PCK_SET_CONFIGURATION_REQ – Set Configuration Parameter Request

3.1.2 Bus parameter

The bus parameter structure `CANOPEN_SLAVE_BUSPARAM_DATA_T` related to parameter `tDeviceCfg` consists of the following items:

Parameter	Type	Meaning	Range of Values
ulSlaveNodeId	UINT32	Node ID of CANopen Slave	1...127
ulBaudrate	UINT32	Baudrate for CANopen network	See Table 27 on page 45.
ulCanOpenFlags	UINT32	CANopen Flags BIT 0: 29-BIT IDENTIFIER DISABLED/ENABLED Not supported yet, set to zero BIT 1-3: Reserved for further use, set to zero (BIT4 up to BIT10 are only applicable in Standard Mode) BIT 4: Evaluate Vendor ID DISABLED/ENABLED BIT 5: Evaluate Product Code DISABLED/ENABLED BIT 6: Evaluate Serial Number DISABLED/ENABLED BIT 7: Evaluate Revision Number DISABLED/ENABLED BIT 8: Evaluate Device Type DISABLED/ENABLED BIT 9: Evaluate Object Size DISABLED/ENABLED BIT 10: Evaluate PDO Count DISABLED/ENABLED BIT 11-15: Reserved for further use, set to zero BIT 16: Disable Send COS Synchronous Acyclic BIT 17: Disable Send COS Manufacturer Spec. BIT 18: Disable Send COS Profile Spec. BIT 19: Reserved for further use, set to zero BIT 20: Enable reject if restricted CAN ID is confugured. BIT 21 - 23: Reserved for further use, set to zero BIT 24: Hold last state BIT 25-27: Reserved for further use, set to zero BIT 28: Extended Mode Flag BIT 29-31: Reserved for further use, set to zero	Flags
tStdBusParam or tExtBusParam	union	This item either contains the standard parameters or the extended parameters depending on bit 29. Standard parameters (bit 29 = 0), see Table 17 on page 34. Extended parameters (bit 29 = 1), see Table 22 on page 37.	-
ul29BitCode	UINT32	29 bit code for acceptance filtering	
ul29BitMask	UINT32	29 bit mask for acceptance filtering	

Table 26: Bus parameter structure CANOPEN_SLAVE_BUSPARAM_DATA_T

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

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

Value	Corresponding Baud Rate of CANopen Network
0	1 MBaud
1	800 KBaud
2	500 KBaud
3	250 KBaud
4	125 KBaud
5	100 KBaud (this value is officially not defined in the CAN specifications!)
6	50 KBaud
7	20 KBaud
8	10 KBaud
255	Auto-Detection

Table 27: Codes and Corresponding Baud Rates of CANopen Network

The standard bus parameter structure `CANOPEN_SLAVE_STD_BUSPARAM_DATA_T` and the extended bus parameter structure `CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T` are described in *Table 17: Standard bus parameter structure CANOPEN_SLAVE_STD_BUSPARAM_DATA_T* on page 34 and in *Table 22: Extended bus parameter structure CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T* on page 37.

The following table on page 46 explains the meaning of the CANopen Flags in a more detailed manner:

Explanation of Parameter `ulCanOpenFlags`

Bit	Name	Comment	Applicable for Standard Mode	Applicable for Extended Mode
0	29 bit identifier DISABLED/ENABLED	Currently, 29 bit identifiers are not supported. Therefore set this value to 0.	x	x
4	Evaluate Vendor ID DISABLED/ENABLED	The `ulVendorId` parameter provides the value for the Vendor ID entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 1) if bit 4 in parameter `ulCanopenFlags` is set. Otherwise, the vendor ID is set to zero.	x	-
5	Evaluate Product Code DISABLED/ENABLED	The `ulProductCode` parameter provides the value for the product code entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 2) if bit 5 in parameter `ulCanopenFlags` is set. Otherwise, the product code is set to zero.	x	-
6	Evaluate Serial Number DISABLED/ENABLED	The `ulSerialNumber` parameter provides the value for the serial number entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 4) if bit 6 in parameter `ulCanopenFlags` is set. Otherwise, the serial number is set to zero.	x	-
7	Evaluate Revision Number DISABLED/ENABLED	The `ulRevisionNumber` parameter provides the value for the revision number entry in the object dictionary of the CANopen slave (Object 0x1018, sub-index 3) if bit 7 in parameter `ulCanopenFlags` is set. Otherwise, the revision number is set to default.	x	-
8	Evaluate Device Type DISABLED/ENABLED		x	-
9	Evaluate Object Size DISABLED/ENABLED		x	-
10	Evaluate PDO Count DISABLED/ENABLED		x	-
16	Send COS Synchronous Acyclic DISABLED/ENABLED	No automatic transmission of TxPDOs in transmission mode 0, else PDO sent automatically with start node and on data change with next SYNC.	x	x
17	Send COS Manufacturer Spec. DISABLED/ENABLED	No automatic transmission of TxPDOs in transmission mode 254, else PDO sent automatically with start node and on data change.	x	x
18	Send COS Profile Spec. DISABLED/ENABLED	No automatic transmission of TxPDOs in transmission mode 255, else PDO sent automatically with start node and on data change.	x	x
20	Enable reject if restricted CAN ID is confugured.	If enabled the CANopen Slave stack will reject restricted CAN IDs.	x	
24	Hold last state	Hold last data in case of not operational, else data is cleared.	x	x
28	Extended mode flag	The extended mode flag switches on and off the Extended Mode.	x	x

Table 28: Explanation of Parameter ulCanOpenFlags

The following applies for filtering of 29 bit CANopen-IDs (only supported in Data Link Layer/CAN DL Task):

A receive filter for the COB-IDs can be defined.

ul29BitMask

Here it is possible to define the bits, the filter uses. In other words: All bits currently not set will not be filtered out.

ul29BitCode

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.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_STD_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_STD_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_STD_BUSPARAM_DATA_T;
/** type of <code>CANOPEN_SLAVE_EXT_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_EXT_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T;
/** type of <code>CANOPEN_SLAVE_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_BUSPARAM_DATA_T;

struct CANOPEN_SLAVE_STD_BUSPARAM_DATA_Ttag
{
  TLR_UINT32 ulVendorId;       /* Vendor ID               */
  TLR_UINT32 ulProductCode;    /* Product code            */
  TLR_UINT32 ulSerialNumber;   /* Serial number           */
  TLR_UINT32 ulRevisionNumber; /* Revision number         */
  TLR_UINT32 ulDeviceType;     /* Device Type             */

  TLR_UINT8  bObject2000Size;  /* Size of object 2000     */
  TLR_UINT8  bObject2001Size;  /* Size of object 2001     */
  TLR_UINT8  bObject2002Size;  /* Size of object 2002     */
  TLR_UINT8  bObject2003Size;  /* Size of object 2003     */

  TLR_UINT8  bObject2200Size;  /* Size of object 2200     */
  TLR_UINT8  bObject2201Size;  /* Size of object 2201     */
  TLR_UINT8  bObject2202Size;  /* Size of object 2202     */
  TLR_UINT8  bObject2203Size;  /* Size of object 2203     */

  TLR_UINT16 usNumOfRxPdo;     /* Number of receive PDOs  */
  TLR_UINT16 usNumOfTxPdo;     /* Number of transmit PDOs */

  TLR_UINT32 aulReserved[2];   /* Reserved, set to zero   */

};

struct CANOPEN_SLAVE_EXT_BUSPARAM_DATA_Ttag
{
  TLR_UINT16 usNumOfRxPdo;     /* Number of receive PDOs  */
  TLR_UINT16 usNumOfTxPdo;     /* Number of transmit PDOs */
  TLR_UINT32 aulReserved[9];   /* Reserved, set to zero   */
};

/*****************************************************************************/
/** type of <code>CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag
  CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_T;

#define CANOPEN_APS_SYS_FLAG_COM_CONTROLLED_RELEASE 0x00000001L /* Automatic start    */
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_ENABLED      0x00000002L /* Not supported      */
```

```
#define CANOPEN_APS_SYS_FLAG_IO_STATUS_32_BIT       0x00000004L /* Not supported      */

#define CANOPEN_APS_SYS_FLAG_ADDRESS_SWITCH         0x00000010L /* Switch for address */
#define CANOPEN_APS_SYS_FLAG_BAUD_SWITCH            0x00000020L /* Switch for baud    */

#define CANOPEN_APS_WD_OFF                          0x00000000L /* Watchdog disabled */
#define CANOPEN_APS_WD_MIN_TIMEOUT                  0x00000014L /* Minimum watchdog  */
#define CANOPEN_APS_WD_MAX_TIMEOUT                  0x0000FFFFL /* Maximum watchdog  */

#define CANOPEN_SLAVE_MIN_SLAVE_NODE_ID  1
#define CANOPEN_SLAVE_MAX_SLAVE_NODE_ID  127

#define CANOPEN_SLAVE_CFG_BAUD_1000    0x00000000L   /* 1MBaud   */
#define CANOPEN_SLAVE_CFG_BAUD_800     0x00000001L   /* 800kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_500     0x00000002L   /* 500kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_250     0x00000003L   /* 250kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_125     0x00000004L   /* 125kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_100     0x00000005L   /* 100kBaud */
#define CANOPEN_SLAVE_CFG_BAUD_50      0x00000006L   /* 50kBaud  */
#define CANOPEN_SLAVE_CFG_BAUD_20      0x00000007L   /* 20kBaud  */
#define CANOPEN_SLAVE_CFG_BAUD_10      0x00000008L   /* 10kBaud  */
#define CANOPEN_SLAVE_CFG_BAUD_AUTO_DETECTION 0x000000FFL /* Auto-Baud detection */

#define CANOPEN_SLAVE_STD_CFG_DEF_OBJECT_SIZE  128 /* Default object size in std mode */

#define CANOPEN_SLAVE_CFG_MAX_RXPDO            256 /* Maximum number of RxPDO    */
#define CANOPEN_SLAVE_CFG_MAX_TXPDO            256 /* Maximum number of TxPDO    */

struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
{
  TLR_UINT32 ulSlaveNodeId;  /* Node ID       */
  TLR_UINT32 ulBaudrate;     /* Baud-rate     */
  TLR_UINT32 ulCanOpenFlags; /* CANopen flags */

   union
  {
    CANOPEN_SLAVE_STD_BUSPARAM_DATA_T tStdBusParam; /* Parameter for standard mode*/
    CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T tExtBusParam; /* Parameter for extended mode*/

  } uMode;

  TLR_UINT32 ul29BitCode; /* 29Bit Code */
  TLR_UINT32 ul29BitMask; /* 29Bit Mask */

};

struct CANOPEN_APS_SET_CONFIGURATION_REQ_DATA_Ttag
{
  TLR_UINT32 ulSystemFlags;               /* System flags  */
  TLR_UINT32 ulWdgTime;                   /* Watchdog time */
  CANOPEN_SLAVE_BUSPARAM_DATA_T tBusParam; /* Bus parameter */

};

/*****************************************************************************/
/** type of <code>CANOPEN_APS_SET_CONFIGURATION_REQ_Ttag</code> */
typedef struct CANOPEN_APS_SET_CONFIGURATION_REQ_Ttag
  CANOPEN_APS_SET_CONFIGURATION_REQ_T;

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

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

3.2 CANopen Slave Services

This section describes the packages of the CANopen Slave services.

The CANopen Slave Protocol Stack V3 supports two different application situations:

- Loadable Firmware (*LFW*)
- Linkable Object Modules (*LOM*)

This section describes the packages if you use LFW or LOM. Section *Packages for LOM* on page 97 describes the packages which can be used with Linkable Object Modules (LOM) only.

The following table lists all packets that can be uses with LFW and LOM. Additionaly, the table lists whether a package will be denied if the '*Configuration Lock'* flag is set.

Section	Name	LFW	LOM	Denied on Cfg. Lock
3.2.1	CANOPEN_SLAVE_STARTSTOP_REQ/CNF – Start/Stop CANopen Network	x	x	x
3.2.2	CANOPEN_SLAVE_EXCHANGE_DATA_REQ/CNF – Exchange Data	x	x	x
3.2.3	CANOPEN_SLAVE_SEND_EMCY_REQ/CNF – Send Emergency Message	x	x	x
3.2.4	CANOPEN_SLAVE_SEND_EMCY_IND/RES – Emergency Message Indication	x	x	x
3.2.5	CANOPEN_SLAVE_SET_NMT_STATE_REQ/CNF – Set NMT State	x	x	x
3.2.6	CANOPEN_SLAVE_SEND_TIME_STAMP_REQ/CNF – Send Time Stamp	x	x	x
3.2.7	CANOPEN_SLAVE_RECV_TIME_STAMP_IND/RES – Receive Time Stamp Indication	x	x	x
3.2.8	CANOPEN_SLAVE_SEND_TXPDO_REQ – Send TxPDO Request	x	x	x
3.2.9	CANOPEN_SLAVE_RECV_RXPDO_REQ/CNF – Receive RxPDO Request	x	x	x
3.2.10	CANOPEN_SLAVE_RECV_RXPDO_IND/RES – Receive RxPDO Indication	x	x	x
3.2.11	CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ/CNF – Set Events Indicated Request	x	x	0
3.2.12	CANOPEN_SLAVE_GET_IO_INFO_REQ/CNF – Get I/O Info	x	x	x
3.2.13	CANOPEN_SLAVE_NMT_STATE_CHANGE_IND/RES – NMT State Change Indication	x	x	x
3.2.14	CANOPEN_SLAVE_ERR_CTRL_EVENT_IND/RES – Error Control Event Indication	x	x	x
3.2.15	CANOPEN_SLAVE_NMT_COMMAND_IND/RES – NMT Command Indication	x	x	x
3.2.16	CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ/CNF – Setup PDO Indication	x	x	x
3.2.17	CANOPEN_SLAVE_RECEIVE_PDO_IND/RES – Receive PDO Indication	x	x	x

Table 29: Packets of CANopen Slave Protocol Stack V3 and Restrictions of Usage

The following packet of the CANopen Slave-Task will be denied if the 'Configuration Lock' flag is set:

CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ/CNF – Set Events Indicated Request

3.2.1 CANOPEN_SLAVE_STARTSTOP_REQ/CNF – Start/Stop CANopen Network

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

- If the `ulMode` parameter has the value 0 and the current NMT State is either *Pre-operational* state or *Operational* state, the NMT State will switch to *Stopped* state.
- If the `ulMode` parameter has the value 1 and the current NMT State is either *Pre-operational* state or *Stopped* state, the NMT State will switch to *Operational* state.

The BUS-ON/BUS-OFF flag is set accordingly (`ulMode=0`: BUS-OFF `ulMode=1`: BUS-ON)

For more information about NMT States, see section 2.3.1 *NMT Slave State Machine*.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_STARTSTOP_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_STARTSTOP_REQ_DATA_Ttag
  CANOPEN_SLAVE_STARTSTOP_REQ_DATA_T;

#define CANOPEN_SLAVE_STOP_CANOPEN     0x00000000L  /* Stop CANopen  */
#define CANOPEN_SLAVE_START_CANOPEN    0x00000001L  /* Start CANopen */

/** Structure of task command start/stop CANopen request data */
struct CANOPEN_SLAVE_STARTSTOP_REQ_DATA_Ttag
{
  TLR_UINT32 ulMode; /* Start or stop CANopen */
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_Ttag
  CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulLen	UINT32	4	Packet Data Length in bytes
ulCmd	UINT32	0x00002902	`CANOPEN_SLAVE_STARTSTOP_REQ`
Data			
ulMode	UINT32	 0 1	Depending on this assignment, communication is either started or stopped: Stop CANopen Start CANopen

Table 30: CANOPEN_SLAVE_PACKET_STARTSTOP_REQ_T – Start/Stop Communication Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_STARTSTOP_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_STARTSTOP_CNF_DATA_Ttag
  CANOPEN_SLAVE_STARTSTOP_CNF_DATA_T;

#define CANOPEN_SLAVE_STOP_CANOPEN     0x00000000L  /* Stop CANopen  */
#define CANOPEN_SLAVE_START_CANOPEN    0x00000001L  /* Start CANopen */

/** Structure of task command start/stop CANopen confirmation data */
struct CANOPEN_SLAVE_STARTSTOP_CNF_DATA_Ttag
{
  TLR_UINT32 ulMode; /* Start or stop CANopen */
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_Ttag
  CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	4	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *"Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x00002903	CANOPEN_SLAVE_STARTSTOP_CNF
Data			
ulMode	UINT32		Depending on this assignment, communication is either started or stopped:
		0	Stop CANopen
		1	Start CANopen

Table 31: CANOPEN_SLAVE_PACKET_STARTSTOP_CNF_T – Start/Stop Communication Confirmation

3.2.2 CANOPEN_SLAVE_EXCHANGE_DATA_REQ/CNF – Exchange Data

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

This packet is only applicable in Standard Mode.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_Ttag
  CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_T;

/** Structure of task command start/stop CANopen request data */
struct CANOPEN_SLAVE_EXCHANGE_DATA_REQ_DATA_Ttag
{
  TLR_UINT32 ulRecvIndex;     /* Object index for recv data     */
  TLR_UINT32 ulRecvSubIndex;  /* Object sub-index for recv data */
  TLR_UINT32 ulRecvDataCnt;   /* Recv data count                */

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

  TLR_UINT8  abSendData[CANOPEN_SLAVE_MAX_SEND_SUB_IDX];
};

#define CANOPEN_SLAVE_EXCHANGE_DATA_HEAD_SIZE (6 * sizeof(TLR_UINT32))

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_Ttag
  CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulLen	UINT32	24.. 279	Packet Data Length in bytes
ulCmd	UINT32	0x290A	CANOPEN_SLAVE_EXCHANGE_DATA_REQ - Command
Data			
ulRecvIndex	UINT32	0x2200...0x2203	Receive object index
ulRecvSubIndex	UINT32	1.. 255	Receive object sub-index
ulRecvDataCnt	UINT32	0 ...255	Number of data bytes to be read
ulSendIndex	UINT32	0x2000...0x2003	Send object index
ulSendSubIndex	UINT32	1.. 255	Send object sub-index
ulSendDataCnt	UINT32	0..255	Number of data bytes to be sent
abSendData[255]	UINT8[]		Data to be sent

Table 32: CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_REQ_T – Exchange Data Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_EXCHANGE_DATA_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_EXCHANGE_DATA_CNF_DATA_Ttag
  CANOPEN_SLAVE_EXCHANGE_DATA_CNF_DATA_T;

/** Structure of task command start/stop CANopen confirmation data */
struct CANOPEN_SLAVE_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_SLAVE_MAX_RECV_SUB_IDX];
};

#define CANOPEN_SLAVE_EXCHANGE_DATA_HEAD_SIZE (6 * sizeof(TLR_UINT32))

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_Ttag
  CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	24 .. 279	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *"Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x290B	CANOPEN_SLAVE_EXCHANGE_DATA_CNF
Data			
ulRecvIndex	UINT32	0x2200...0x2203	Receive object index
ulRecvSubIndex	UINT32	1.. 255	Receive object sub-index
ulRecvDataCnt	UINT32	0 ..255	Number data byte to be read
ulSendIndex	UINT32	0x2000..0x2003	Send object index
ulSendSubIndex	UINT32	1.. 255	Send object sub-index
ulSendDataCnt	UINT32	0 ..255	Number data byte to be sent
abRecvData[255]	UINT8[]		Receive data

Table 33: CANOPEN_SLAVE_PACKET_EXCHANGE_DATA_CNF_T –Exchange Data Confirmation

3.2.3 CANOPEN_SLAVE_SEND_EMCY_REQ/CNF – Send Emergency Message

This packet sends an emergency telegram to the CANopen network. The emergency error codes are listed in section *Emergency Error Codes* on page 125. The content of the Error Register is listed in section *Error Register* on page 126.

The field `abManErrorCode[5]` additionally provides the possibility to supply 5 bytes of manufacturer-specific error information.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_Ttag
  CANOPEN_SLAVE_SEND_EMCY_REQ_DATA_T;

#define CANOPEN_SLAVE_EMCY_DATA_SIZE 5

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

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

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulLen	UINT32	8	Packet Data Length in bytes
ulCmd	UINT32	0x2918	`CANOPEN_SLAVE_SEND_EMCY_REQ`
Data			
usErrorCode	UINT16		Emergency Error Code, see Table 96.
abManError Code[5]	UINT8[]		Area for manufacturer-specific error codes
bErrorRegister	UINT8	Bit mask	See Table 97.

Table 34: `CANOPEN_SLAVE_PACKET_SEND_EMCY_REQ_T` – Send Emergency Message Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *"Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2919	CANOPEN_SALVE_SEND_EMCY_CNF - Command

Table 35: CANOPEN_SLAVE_PACKET_SEND_EMCY_CNF_T – Send Emergency Message Confirmation

3.2.4 CANOPEN_SLAVE_SEND_EMCY_IND/RES – Emergency Message Indication

This indication informs the application about the sending of an emergency telegram by the CANopen protocol stack. The emergency error codes are listed in section *Emergency Error Codes* on page 125. The content of the Error Register is listed in section *Error Register* on page 126.

The field `abManErrorCode[5]` additionally provides the possibility to supply 5 bytes of manufacturer-specific error codes.

Packet Structure Reference

```
/** type of <code>CANOPEN_SLAVE_SEND_EMCY_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SEND_EMCY_IND_DATA_Ttag
  CANOPEN_SLAVE_SEND_EMCY_IND_DATA_T;

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

/** type of <code>CANOPEN_SLAVE_PACKET_SEND_EMCY_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_EMCY_IND_Ttag
  CANOPEN_SLAVE_PACKET_SEND_EMCY_IND_T;

struct CANOPEN_SLAVE_PACKET_SEND_EMCY_IND_Ttag
{
  TLR_PACKET_HEADER_T                tHead; /** packet header.            */
  CANOPEN_SLAVE_SEND_EMCY_IND_DATA_T  tData; /** packet induest data.      */
};
```

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	8	Packet Data Length in bytes
ulCmd	UINT32	0x2938	`CANOPEN_SLAVE_SEND_EMCY_IND`
Data			
usErrorCode	UINT16		Emergency Error Code, see Table 96
abManError Code[5]	UINT8[]		Area for manufacturer-specific error codes, see Table 98 and Table 99.
bErrorRegister	UINT8	Bit mask	See Table 97.

Table 36: `CANOPEN_SLAVE_PACKET_SEND_EMCY_IND_T` – Send Emergency Message Indication

Packet Structure Reference

```
/** type of <code>CANOPEN_SLAVE_PACKET_SEND_EMCY_RES_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_EMCY_RES_Ttag
  CANOPEN_SLAVE_PACKET_SEND_EMCY_RES_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	8	Packet Data Length in bytes
ulSta	UINT32	0	See section 6.2 *"Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2939	`CANOPEN_SLAVE_SEND_EMCY_RES` - Command

Table 37: `CANOPEN_SLAVE_PACKET_SEND_EMCY_RES_T` – Response to Send Emergency Message Indication

3.2.5 CANOPEN_SLAVE_SET_NMT_STATE_REQ/CNF – Set NMT State

This packet allows the host application to set the NMT state of the CANopen slave.

Normally the state is set by the NMT Master using the CANOPEN_SLAVE_NMT_COMMAND_IND indication, but sometimes it may be necessary to control the state manually from the host application.

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

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

Table 38: NMT States

The value 1 for Operational is only supported for compatibility reasons. However, according to the CANopen specification, only the NMT Master is allowed to set the NMT state to *Operational*.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_Ttag
  CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_T;

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

struct CANOPEN_SLAVE_SET_NMT_STATE_REQ_DATA_Ttag
{
  TLR_UINT32 ulNmtState;  /* NMT state */
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_NMT_STATE_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_NMT_STATE_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SET_NMT_STATE_REQ_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulLen	UINT32	4	Packet Data Length in bytes
ulCmd	UINT32	0x0000291A	*CANOPEN_SLAVE_SET_NMT_STATE_REQ*
Data			
ulNmtState	UINT32	1,2,128..130	State requested to be set *See Table 38: NMT States*

Table 39: CANOPEN_SLAVE_PACKET_SET_NMT_STATE_REQ_T – Set NMT State Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_Ttag
  CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_T;

struct CANOPEN_SLAVE_SET_NMT_STATE_CNF_DATA_Ttag
{
  TLR_UINT32 ulNmtState; /* NMT state */
};
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_NMT_STATE_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_NMT_STATE_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SET_NMT_STATE_CNF_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	4	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *"Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x0000291B	CANOPEN_SLAVE_SET_NMT_STATE_CNF
Data			
ulNmtState	UINT32	1,2,128..130	NMT State having really been set *See Table 38: NMT States*

Table 40: CANOPEN_SLAVE_PACKET_SET_NMT_STATE_CNF_T – Set NMT State Confirmation

3.2.6 CANOPEN_SLAVE_SEND_TIME_STAMP_REQ/CNF – Send Time Stamp

This packet allows to send a time stamp according to the CANopen time stamp protocol, if the CANopen Slave is a valid time stamp producer.

See *Figure 10: CANopen Time Stamp Protocol* below:

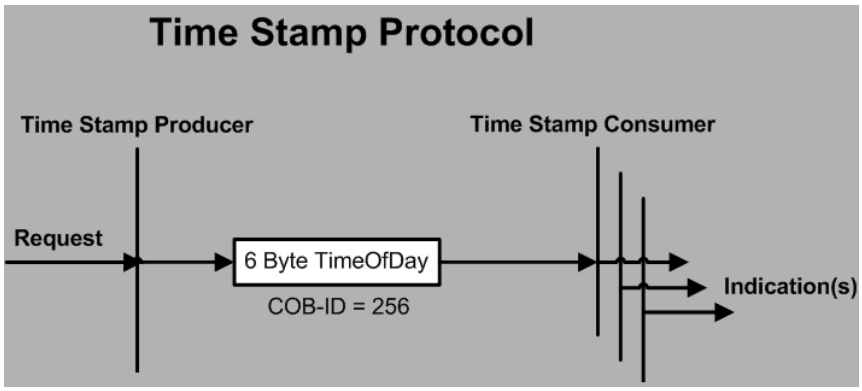


Figure 10: CANopen Time Stamp Protocol

The contents of the time stamp is divided into a milliseconds part and a days part:

- The milliseconds part (variable `ulMilliseconds`) contains the number of milliseconds that have occurred since midnight.
- The days part (variable `usDays`) contains the number of days that have occurred since January 1, 1984.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SEND_TIME_STAMP_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SEND_TIME_STAMP_REQ_DATA_Ttag
  CANOPEN_SLAVE_SEND_TIME_STAMP_REQ_DATA_T;

struct CANOPEN_SLAVE_SEND_TIME_STAMP_REQ_DATA_Ttag
{
  TLR_UINT32 ulMilliseconds;
  TLR_UINT16 usDays;
};

/*****************************************************************************/

/** type of <code>CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_REQ_T;

 struct CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_REQ_Ttag
{
  TLR_PACKET_HEADER_T                      tHead; /** packet header.            */
  CANOPEN_SLAVE_SEND_TIME_STAMP_REQ_DATA_T  tData; /** packet request data.      */
};

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle
ulLen	UINT32	6	Packet Data Length in bytes
ulCmd	UINT32	0x291E	CANOPEN_SLAVE_SEND_TIME_STAMP_REQ
Data			
ulMilliseconds	UINT32	0 ... 86.399.999	Milliseconds part of time stamp (number of milliseconds that have occurred since last midnight)
usDays	UINT16	0..65535	Days part of time stamp (number of days that have occurred since January 1, 1984)

Table 41: CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_REQ_T - Send Time Stamp Request

Packet Structure Reference

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

/** type of <code>CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_CNF_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x291F	CANOPEN_SLAVE_SEND_TIME_STAMP_CNF

Table 42: CANOPEN_SLAVE_PACKET_SEND_TIME_STAMP_CNF_T – Confirmation to Send Time Stamp Request

3.2.7 CANOPEN_SLAVE_RECV_TIME_STAMP_IND/RES – Receive Time Stamp Indication

This indication packet is received when the Time Stamp Producer sends a time stamp by issuing a time service (TIME). The time service and the time protocol are described in section 9.2.4 of CANopen document DR301. Also, see *Figure 10: CANopen Time Stamp Protocol* in the preceding subsection:

The content of the time stamp is divided into a milliseconds part and a days part:

- The milliseconds part (variable `ulMilliseconds`) contains the number of milliseconds that have occurred since midnight.
- The days part (variable `usDays`) contains the number of days that have occurred since January 1, 1984.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_RECV_TIME_STAMP_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_RECV_TIME_STAMP_IND_DATA_Ttag
  CANOPEN_SLAVE_RECV_TIME_STAMP_IND_DATA_T;

 struct CANOPEN_SLAVE_RECV_TIME_STAMP_IND_DATA_Ttag
{
  TLR_UINT32 ulMilliseconds;
  TLR_UINT16 usDays;
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_IND_Ttag
  CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_IND_T;

 struct CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_IND_Ttag
{
  TLR_PACKET_HEADER_T                      tHead; /** packet header.            */
  CANOPEN_SLAVE_RECV_TIME_STAMP_IND_DATA_T  tData; /** packet request data.      */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	6	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2920	CANOPEN_SLAVE_RECV_TIME_STAMP_IND
Data			
ulMilliseconds	UINT32	0 ... 86.399.999	Milliseconds part of time stamp (number of milliseconds that have occurred since last midnight)
usDays	UINT16	0..65535	Days part of time stamp (number of days that have occurred since January 1, 1984)

Table 43: `CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_IND_T` - Receive Time Stamp Indication

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_RES_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_RES_Ttag
  CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_RES_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2921	CANOPEN_SLAVE_RECV_TIME_STAMP_RES

Table 44: CANOPEN_SLAVE_PACKET_RECV_TIME_STAMP_RES- Response to Receive Time Stamp Indication

3.2.8 CANOPEN_SLAVE_SEND_TXPDO_REQ – Send TxPDO Request

This packet allows to send one or more TxPDOs to a communication partner (CANopen Master or Slave) on the CANopen network using the Write PDO.Protocol, see upper part of *Figure 11: CANopen PDO.Protocol*.

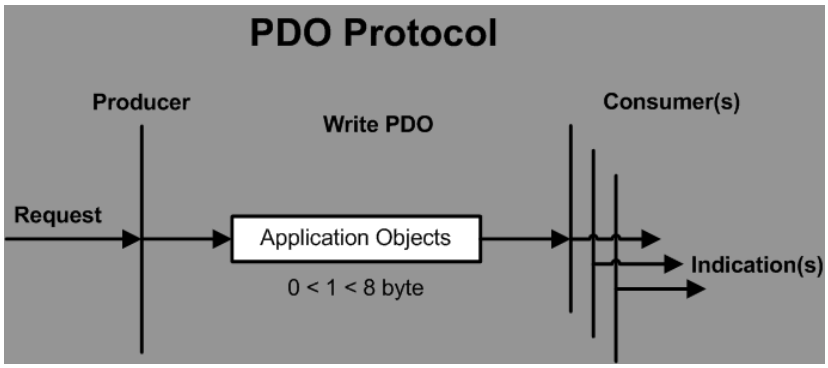


Figure 11: CANopen PDO.Protocol

This can be done simultaneously for up to 16 TxPDOs, whose numbers must be assigned to the members of array `aulRecvRxPdoNumber[]` of the request packet.

The 16 TxPDOs will be processed separately and for each the resulting status code (success/error) will be stored in array `aulRecvRxPdoResult[]` of the confirmation packet

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SEND_TXPDO_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SEND_TXPDO_REQ_DATA_Ttag
  CANOPEN_SLAVE_SEND_TXPDO_REQ_DATA_T;

#define CANOPEN_SLAVE_SEND_TXPDO_REQ_MAX    16

 struct CANOPEN_SLAVE_SEND_TXPDO_REQ_DATA_Ttag
{
  TLR_UINT32 aulSendTxPdoNumber[CANOPEN_SLAVE_SEND_TXPDO_REQ_MAX];
};

/** type of <code>CANOPEN_SLAVE_PACKET_SEND_TXPDO_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_TXPDO_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SEND_TXPDO_REQ_T;

 struct CANOPEN_SLAVE_PACKET_SEND_TXPDO_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead; /** packet header. */
  CANOPEN_SLAVE_SEND_TXPDO_REQ_DATA_T tData; /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle
ulLen	UINT32	64	Packet Data Length in bytes
ulCmd	UINT32	0x2922	CANOPEN_SLAVE_SEND_TXPDO_REQ
Data			
aulSendTxPdoNumber[]	UINT32[16]	1..255 for each number	Array of numbers of TxPDOs for sending

Table 45: CANOPEN_SLAVE_PACKET_SEND_TXPDO_REQ_T – Send TxPDO Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SEND_TXPDO_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SEND_TXPDO_CNF_DATA_Ttag
  CANOPEN_SLAVE_SEND_TXPDO_CNF_DATA_T;

#define CANOPEN_SLAVE_SEND_TXPDO_REQ_MAX    16

 struct CANOPEN_SLAVE_SEND_TXPDO_CNF_DATA_Ttag
{
  TLR_UINT32 aulSendTxPdoResult[CANOPEN_SLAVE_SEND_TXPDO_REQ_MAX];
};

/** type of <code>CANOPEN_SLAVE_PACKET_SEND_TXPDO_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SEND_TXPDO_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SEND_TXPDO_CNF_T;

 struct CANOPEN_SLAVE_PACKET_SEND_TXPDO_CNF_Ttag
{
  TLR_PACKET_HEADER_T                 tHead; /** packet header. */
  CANOPEN_SLAVE_SEND_TXPDO_CNF_DATA_T tData; /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	64	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2923	CANOPEN_SLAVE_SEND_TXPDO_CNF
Data			
aulSendTxPdoResult[]	UINT32[16]		Array of Results of sending TxPDOs

Table 46: CANOPEN_SLAVE_PACKET_SEND_TXPDO_CNF_T – Confirmation to Send TxPDO Request

3.2.9 CANOPEN_SLAVE_RECV_RXPDO_REQ/CNF – Receive RxPDO Request

This packet sends one or more RTR Telegrams (CAN Remote Frames) to the CANopen network in order to request a RxPDO from a communication partner (CANopen Master or Slave). Issuing the request packet `CANOPEN_SLAVE_RECV_RXPDO_REQ` causes a CAN Remote Frame to be sent causing an indication at the communication partner. It is up to the communication partner to decide, whether it wants to react to this RTR Telegram, or not. In case of a positive decision, the communication partner will send a response with the requested application to the CANopen Slave.

Requesting can be done simultaneously for up to 16 RxPDOs, whose numbers must be assigned to the members of array `aulRecvRxPdoNumber[]` of the request packet.

The 16 RxPDOs will be processed separately and for each the resulting status code (success/error) will be stored in array `aulRecvRxPdoResult[]` of the confirmation packet

For details of the protocol, see the lower part of *Figure 12: CANopen PDO Protocol.*

In *Figure 12: CANopen PDO Protocol*, request(s) there relates to the `CANOPEN_SLAVE_RECV_RXPDO_REQ` packet and confirmation(s) to the `CANOPEN_SLAVE_RECV_RXPDO_CNF` packet.

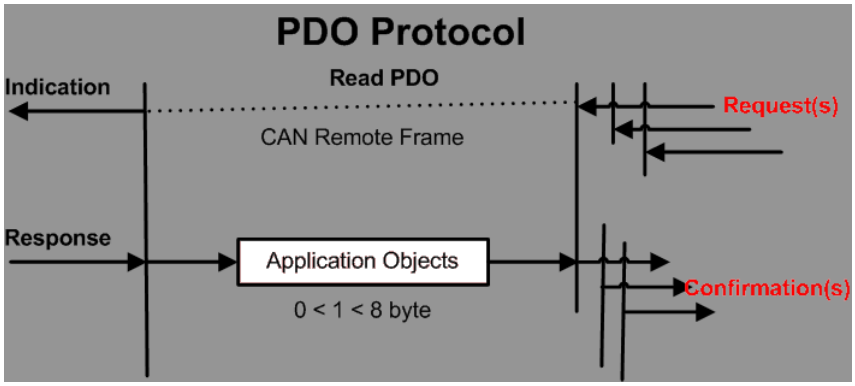


Figure 12: CANopen PDO Protocol

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_RECV_RXPDO_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_RECV_RXPDO_REQ_DATA_Ttag
  CANOPEN_SLAVE_RECV_RXPDO_REQ_DATA_T;

#define CANOPEN_SLAVE_RECV_RXPDO_REQ_MAX    16

struct CANOPEN_SLAVE_RECV_RXPDO_REQ_DATA_Ttag
{
  TLR_UINT32 aulRecvRxPdoNumber[CANOPEN_SLAVE_RECV_RXPDO_REQ_MAX];
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_RECV_RXPDO_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_RECV_RXPDO_REQ_Ttag
  CANOPEN_SLAVE_PACKET_RECV_RXPDO_REQ_T;

struct CANOPEN_SLAVE_PACKET_RECV_RXPDO_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead; /** packet header. */
  CANOPEN_SLAVE_RECV_RXPDO_REQ_DATA_T tData; /** packet data.   */
};

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32		Destination Queue-Handle
ulLen	UINT32	64	Packet Data Length in bytes
ulCmd	UINT32	0x2924	CANOPEN_SLAVE_RECV_RXPDO_REQ
Data			
aulRecvRxPdoNumber[]	UINT32[16]	1..255 for each number	Array of numbers of RxPDOs to be received

Table 47: CANOPEN_SLAVE_PACKET_RECV_RXPDO_REQ_T - Receive RxPDO Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_RECV_RXPDO_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_RECV_RXPDO_CNF_DATA_Ttag
  CANOPEN_SLAVE_RECV_RXPDO_CNF_DATA_T;

#define CANOPEN_SLAVE_RECV_RXPDO_REQ_MAX    16

struct CANOPEN_SLAVE_RECV_RXPDO_CNF_DATA_Ttag
{
  TLR_UINT32 aulRecvRxPdoResult[CANOPEN_SLAVE_RECV_RXPDO_REQ_MAX];
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_RECV_RXPDO_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_RECV_RXPDO_CNF_Ttag
  CANOPEN_SLAVE_PACKET_RECV_RXPDO_CNF_T;

struct CANOPEN_SLAVE_PACKET_RECV_RXPDO_CNF_Ttag
{
  TLR_PACKET_HEADER_T                 tHead; /** packet header. */
  CANOPEN_SLAVE_RECV_RXPDO_CNF_DATA_T tData; /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	64	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2925	CANOPEN_SLAVE_RECV_RXPDO_CNF
Data			
aulRecvRxPdoResult[]	UINT32[16]		Array of Results of received RxPDOs

Table 48: CANOPEN_SLAVE_PACKET_RECV_RXPDO_CNF_T – Confirmation to Receive RxPDO Request

3.2.10 CANOPEN_SLAVE_RECV_RXPDO_IND/RES – Receive RxPDO Indication

This packet indicates the arrival of new process data from a communication partner (CANopen Master or a Slave), i.e. the reception of one or more PDOs. Sending a response does not send new telegrams or execute other reactions.

The packet can simultaneously indicate requests of the communication partner for up to 16 RxPDOs, whose numbers are transmitted by the members of array `aulRecvRxPdoNumber[]` of the request packet.

The 16 RxPDOs will be processed separately and for each the resulting status code (success/error) will be stored in the parameter array `aulRecvRxPdoResult[]` of the indication packet.

The response packet does not have any parameters

PDO communication works based on the producer/consumer model:

- the CANopen Slave acts as a consumer here,
- the communication partner acts as producer.

For details of the protocol, see the lower part of *Figure 13: CANopen PDO Protocol.*

Indication there relates to the `CANOPEN_SLAVE_RECV_RXPDO_IND` packet and Response to the `CANOPEN_SLAVE_RECV_RXPDO_RES` packet.

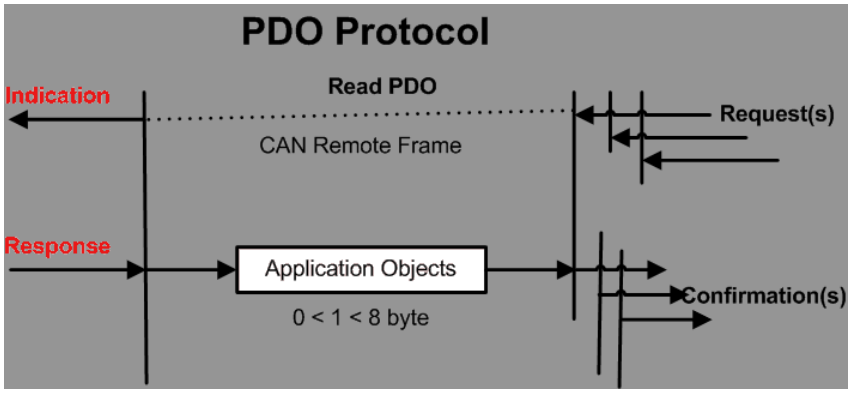


Figure 13: CANopen PDO Protocol

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_RECV_RXPDO_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_RECV_RXPDO_IND_DATA_Ttag
  CANOPEN_SLAVE_RECV_RXPDO_IND_DATA_T;

#define CANOPEN_SLAVE_RECV_RXPDO_IND_MAX    16

 struct CANOPEN_SLAVE_RECV_RXPDO_IND_DATA_Ttag
{
  TLR_UINT32 aulRecvRxPdoNumber[CANOPEN_SLAVE_RECV_RXPDO_IND_MAX];

};
 /*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_RECV_RXPDO_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_RECV_RXPDO_IND_Ttag
  CANOPEN_SLAVE_PACKET_RECV_RXPDO_IND_T;

struct CANOPEN_SLAVE_PACKET_RECV_RXPDO_IND_Ttag
{
  TLR_PACKET_HEADER_T                 tHead; /** packet header. */
  CANOPEN_SLAVE_RECV_RXPDO_IND_DATA_T tData; /** packet data.   */

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	64	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2926	CANOPEN_SLAVE_RECV_RXPDO_IND
Data			
aulRecvRxPdoNumber []	UINT32[16]	1..255 for each number	Array of numbers of received RxPDOs

Table 49: CANOPEN_SLAVE_PACKET_RECV_RXPDO_IND_T –Receive RxPDO Indication

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_RECV_RXPDO_RES_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_RECV_RXPDO_RES_Ttag
  CANOPEN_SLAVE_PACKET_RECV_RXPDO_RES_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2927	CANOPEN_SLAVE_RECV_RXPDO_RES

Table 50: CANOPEN_SLAVE_PACKET_RECV_RXPDO_RES_T – Response to Receive RxPDO Indication

3.2.11 CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ/CNF – Set Events Indicated Request

Note: This packet will be denied in case of being called as long as the 'Configuration lock' flag of the netX is set!

The following types of indications may occur in the CANopen Protocol Stack V3 and indicate that a specific event has happened:

- NMT State Change Event (CANOPEN_SLAVE_NMT_STATE_CHANGE_IND/RES – NMT State Change Indication, for more information see page 76)
- Time Stamp Event (CANOPEN_SLAVE_RECV_TIME_STAMP_IND/RES – Receive Time Stamp Indication, for more information see page 63)
- Error Control Event (CANOPEN_SLAVE_ERR_CTRL_EVENT_IND/RES – Error Control Event Indication, for more information see page 78)
- Receive PDO Event (CANOPEN_SLAVE_RECV_RXPDO_IND/RES – Receive RxPDO Indication, for more information see page 69)
- NMT Command Event (CANOPEN_SLAVE_NMT_COMMAND_IND/RES – NMT Command Indication, for more information see page 81)
- Send EMCY Event (CANOPEN_SLAVE_SEND_EMCY_IND/RES – Emergency Message Indication, for more information see page 57)

Each type of indication is associated to a bit of the bit mask contained in variable `ulEventsIndicated` in the following manner:

Type	Bit No.	Mask (numeric value)	Mask (symbolic name)
NMT State Change Event	0	0x01	`CANOPEN_SLAVE_EVENT_NMT_STATE_CHANGE_MSK`
Time Stamp Event	1	0x02	`CANOPEN_SLAVE_EVENT_TIME_STAMP_MSK`
Error Control Event	2	0x04	`CANOPEN_SLAVE_EVENT_ERR_CTRL_MSK`
Receive PDO Event	3	0x08	`CANOPEN_SLAVE_EVENT_RECV_RXPDO_MSK`
NMT Command Event	4	0x10	`CANOPEN_SLAVE_EVENT_NMT_COMMAND_MSK`
Send EMCY Event	5	0x20	`CANOPEN_SLAVE_EVENT_SEND_EMCY_MSK`

Table 51: Bit Mask `ulEventsIndicated`

Each of these bits within the bit mask of `ulEventsIndicated` allows switching on (when set to 1) and off (when set to 0) the associated type of event indication.

The higher bits of `ulEventsIndicated` should always be set to 0.

The NMT State Change Events include the Module Control Services described in the CANopen Specification CiA Draft Standard 301, subsection 9.2.6.1.1.

These are:

- Start Remote Node
- Stop Remote Node
- Enter Pre-Operational
- Reset Node
- Reset Communication

The Error Control Events allow detecting errors within the CAN network. They include the following types of events described in the CANopen Specification CiA Draft Standard 301, subsection 9.2.6.1.2:

- Node Guarding Event
- Life Guarding Event
- Heartbeat Event

There is one single type of Time Stamp Event defined in CANopen which can be handled by the CANOPEN_SLAVE_RECV_TIME_STAMP_IND/RES – Receive Time Stamp Indication in the CANopen Slave V3 protocol stack.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ_DATA_Ttag
  CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ_DATA_T;

#define CANOPEN_SLAVE_EVENT_NMT_STATE_CHANGE_MSK 0x00000001L
#define CANOPEN_SLAVE_EVENT_TIME_STAMP_MSK       0x00000002L
#define CANOPEN_SLAVE_EVENT_ERR_CTRL_MSK         0x00000004L
#define CANOPEN_SLAVE_EVENT_RECV_RXPDO_MSK       0x00000008L
#define CANOPEN_SLAVE_EVENT_NMT_COMMAND_MSK      0x00000010L
#define CANOPEN_SLAVE_EVENT_SEND_EMCY_MSK        0x00000020L
#define CANOPEN_SLAVE_EVENT_RESERVED_MSK         0xFFFFFFC0L

struct CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ_DATA_Ttag
{
  TLR_UINT32 ulEventsIndicated;

};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_REQ_T;

struct CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_REQ_Ttag
{
  TLR_PACKET_HEADER_T                          tHead; /** packet header. */
  CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ_DATA_T tData; /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle
ulLen	UINT32	4	Packet Data Length in bytes
ulCmd	UINT32	0x2928	CANOPEN_SLAVE_SET_EVENTS_INDICATED_REQ
Data			
ulEventsIndicated	UINT32	Bit mask	Indicated events

Table 52: CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_REQ_T - Set Events Indicated Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_CNF_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2929	CANOPEN_SLAVE_SET_EVENTS_INDICATED_CNF

Table 53: CANOPEN_SLAVE_PACKET_SET_EVENTS_INDICATED_CNF_T – Confirmation to Set Events Indicated Request

3.2.12 CANOPEN_SLAVE_GET_IO_INFO_REQ/CNF – Get I/O Info

This packet can be used to retrieve the counts of received data (Variable `ulRecvDataCnt` of confirmation packet) and send data (Variable `ulSendDataCnt` of confirmation packet) in CANopen data communication.

Note: This information is also part of the slave state, see section 5.2.4 .CANOPEN_SLAVE_STATE_CHANGE_IND/RES – Change of Task State Indication.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_GET_IO_INFO_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_GET_IO_INFO_REQ_Ttag
  CANOPEN_SLAVE_PACKET_GET_IO_INFO_REQ_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle
ulLen	UINT32	0	Packet Data Length in bytes
ulCmd	UINT32	0x292A	CANOPEN_SLAVE_GET_IO_INFO_REQ

Table 54: CANOPEN_SLAVE_PACKET_GET_IO_INFO_REQ_T - Get I/O Info Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_GET_IO_INFO_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_GET_IO_INFO_CNF_DATA_Ttag
  CANOPEN_SLAVE_GET_IO_INFO_CNF_DATA_T;

struct CANOPEN_SLAVE_GET_IO_INFO_CNF_DATA_Ttag
{
  TLR_UINT32 ulRecvDataCnt;
  TLR_UINT32 ulSendDataCnt;
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_GET_IO_INFO_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_GET_IO_INFO_CNF_Ttag
  CANOPEN_SLAVE_PACKET_GET_IO_INFO_CNF_T;

struct CANOPEN_SLAVE_PACKET_GET_IO_INFO_CNF_Ttag
{
  TLR_PACKET_HEADER_T                  tHead; /** packet header.            */
  CANOPEN_SLAVE_GET_IO_INFO_CNF_DATA_T tData; /** packet confirmation data. */
};

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	8	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x292B	CANOPEN_SLAVE_GET_IO_INFO_CNF
Data			
ulRecvDataCnt	UINT32	0 ... 2^{32}-1	Receive Data Count
ulSendDataCnt	UINT32	0 ... 2^{32}-1	Send Data Count

Table 55: `CANOPEN_SLAVE_PACKET_GET_IO_INFO_CNF_T` – Confirmation to Get I/O Info Request

3.2.13 CANOPEN_SLAVE_NMT_STATE_CHANGE_IND/RES – NMT State Change Indication

This packet indicates a change in the CANopen Slave's own NMT State Machine. The new state is delivered with the indication packets `ulNmtState` variable.

The relation between the values and the states is as follows:

Value	Symbolic Name	State	Sub-state
1	CANOPEN_SLAVE_NMT_STATE_OPERATIONAL	*Operational*	-
2	CANOPEN_SLAVE_NMT_STATE_STOP	*Stop*	-
128	CANOPEN_SLAVE_NMT_STATE_PRE_OPERATIONAL	*Pre-operational*	-
129	CANOPEN_SLAVE_NMT_STATE_RESET_NODE	*Initialization*	*Reset Node*
130	CANOPEN_SLAVE_NMT_STATE_RESET_COMM	*Initialization*	*Reset Communication*

Table 56: NMT States

This indication packet allows the application to react and perform all necessary application-internal changes after the change of NMT state. After having performed these, the application should send the CANOPEN_SLAVE_NMT_STATE_CHANGE_RES response packet to the CANopen Master.

For possible reasons of changes of NMT Slave states see section "*NMT State Change Events"* on page 17.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_NMT_STATE_CHANGE_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_NMT_STATE_CHANGE_IND_DATA_Ttag
  CANOPEN_SLAVE_NMT_STATE_CHANGE_IND_DATA_T;

 struct CANOPEN_SLAVE_NMT_STATE_CHANGE_IND_DATA_Ttag
{
  TLR_UINT32 ulNmtState;
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_IND_Ttag
  CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_IND_T;

struct CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_IND_Ttag
{
  TLR_PACKET_HEADER_T                       tHead; /** packet header. */
  CANOPEN_SLAVE_NMT_STATE_CHANGE_IND_DATA_T tData; /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	4	Packet Data Length in bytes
ulCmd	UINT32	0x292E	CANOPEN_SLAVE_NMT_STATE_CHANGE_IND
Data			
ulNmtState	UINT32	1,2,128..130	New NMT State, see *Table 38: NMT States*

Table 57: CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_IND_T - NMT State Change Indication

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_RES_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_RES_Ttag
  CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_RES_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x292F	CANOPEN_SLAVE_NMT_STATE_CHANGE_RES

Table 58: CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_RES_T – Response to NMT State Change Indication

3.2.14 CANOPEN_SLAVE_ERR_CTRL_EVENT_IND/RES – Error Control Event Indication

In a CANopen network, the Master supervises the error control of the entire network and is aware of all changes of the error control state. Additionally, it even shares this knowledge with the slaves by informing these on the most current changes.

This is exactly done with the indication packet `CANOPEN_SLAVE_ERR_CTRL_EVENT_IND` described here, which indicates the occurrence of one or more Error Control Event(s) i.e. changes of the error control state within the entire CANopen network, and informs the CANopen Slave.

Error Control Events allow detecting errors within the CAN network. They include the following types of events occurring at the slave described in the CANopen Specification CiA Draft Standard 301, subsection 9.2.6.1.2:

- Life Guarding Event
- Heartbeat Event

For each change of a node's error control state there is an entry in the array of structures `atErrCtrlEvent[16]`. Each array element contains the following structure:

Name	Meaning	Data type	Range of Values
`ulEvent`	Type of event that has happened	UINT32	1…6
`ulNodeId`	Node-ID	UINT32	1…127

Table 59: Array Elements of `atErrCtrlEvent[16]`

The meaning of the values of `ulEvent` is defined as follows:

Value	Symbolic Name	Meaning
1	`CANOPEN_SLAVE_HEARTBEAT_STARTED`	Heartbeat Start for affected node with ID `ulNodeId`
2	`CANOPEN_SLAVE_HEARTBEAT_ERROR`	Heartbeat Error for affected node with ID `ulNodeId`
3	`CANOPEN_SLAVE_HEARTBEAT_STOPPED`	Heartbeat Stop for affected node with ID `ulNodeId`
4	`CANOPEN_SLAVE_LIFE_GUARD_STARTED`	Lifeguard Start for affected node with ID `0`
5	`CANOPEN_SLAVE_LIFE_GUARD_ERROR`	Lifeguard Error for affected node with ID `0`
6	`CANOPEN_SLAVE_LIFE_GUARD_STOPPED`	Lifeguard Stop for affected node with ID `0`

Table 60: Possible Values of ulEvent and their Meanings

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_ERR_CTRL_EVENT_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_ERR_CTRL_EVENT_IND_DATA_Ttag
  CANOPEN_SLAVE_ERR_CTRL_EVENT_IND_DATA_T;
/** type of <code>CANOPEN_SLAVE_ERR_CTRL_EVENT_Ttag</code> */
typedef struct CANOPEN_SLAVE_ERR_CTRL_EVENT_Ttag
  CANOPEN_SLAVE_ERR_CTRL_EVENT_T;

#define CANOPEN_SLAVE_MAX_ERR_CTRL_EVENT  16

#define CANOPEN_SLAVE_HEARTBEAT_STARTED   0x00000001L
#define CANOPEN_SLAVE_HEARTBEAT_ERROR     0x00000002L
#define CANOPEN_SLAVE_HEARTBEAT_STOPPED   0x00000003L

#define CANOPEN_SLAVE_LIFE_GUARD_STARTED  0x00000004L
#define CANOPEN_SLAVE_LIFE_GUARD_ERROR    0x00000005L
#define CANOPEN_SLAVE_LIFE_GUARD_STOPPED  0x00000006L

 struct CANOPEN_SLAVE_ERR_CTRL_EVENT_Ttag
{
  TLR_UINT32 ulEvent;
  TLR_UINT32 ulNodeId;

};

 struct CANOPEN_SLAVE_ERR_CTRL_EVENT_IND_DATA_Ttag
{
  CANOPEN_SLAVE_ERR_CTRL_EVENT_T atErrCtrlEvent[CANOPEN_SLAVE_MAX_ERR_CTRL_EVENT];
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_IND_Ttag
  CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_IND_T;

struct CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_IND_Ttag
{
  TLR_PACKET_HEADER_T                     tHead; /** packet header. */
  CANOPEN_SLAVE_ERR_CTRL_EVENT_IND_DATA_T tData; /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32		Packet Data Length in bytes
ulCmd	UINT32	0x2930	CANOPEN_SLAVE_ERR_CTRL_EVENT_IND
Data			
atErrCtrlEvent[]	CANOPEN_SLAVE_ERR_CTRL_EVENT_T[16]		Structure containing up to 16 Error Control Events. Also see *Table 59: Array Elements of `atErrCtrlEvent[16]`* and *Table 60: Possible Values of ulEvent and their Meanings.*

Table 61: `CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_IND_T` - Error Control Event Indication

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_NMT_STATE_CHANGE_RES_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_RES_Ttag
  CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_RES_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2931	CANOPEN_SLAVE_ERR_CTRL_EVENT_RES

Table 62: `CANOPEN_SLAVE_PACKET_ERR_CTRL_EVENT_RES_T` – Response to Error Control Event Indication

3.2.15 CANOPEN_SLAVE_NMT_COMMAND_IND/RES – NMT Command Indication

The CANopen Network Management (NMT) within the CANopen Master provides services for initializing, starting, monitoring, resetting and stopping CANopen Slaves. These are also called the Module Control Services

Each time such a service is requested, a CANOPEN_SLAVE_NMT_COMMAND_IND indication is received. This enables the host application to react to the requests of the CANopen Master. Which action should be taken, depends on the value of variable `ulNmtCommand` in the following manner:

Value	Symbolic Name	Meaning/ Action to take
1	CANOPEN_SLAVE_NMT_STATE_OPERATIONAL	Set state to *Operational*
2	CANOPEN_SLAVE_NMT_STATE_STOP	Set state to *Stopped*
128	CANOPEN_SLAVE_NMT_STATE_PRE_OPERATIONAL	Set state to *Pre-operational*
129	CANOPEN_SLAVE_NMT_STATE_RESET_NODE	Reset node
130	CANOPEN_SLAVE_NMT_STATE_RESET_COMM	Reset communication

Table 63: NMT States

Reset communication should set the state to *Initialization*, sub-state *Reset Communication*.

This means first restoring the communication profile area of the Object Dictionary with the power-on values and then run the initialization process like after power-on.

Reset node should set the state to *Initialization*, sub-state *Reset Application*.

This means first restoring the manufacturer-specific area and the device profile area of the slave's Object Dictionary with the default values, then restoring the communication profile area of the Object Dictionary with the power-on values and finally run the initialization process like after power-on.

The NMT Protocol associated with the NMT Command Event is illustrated in *Figure 14: NMT Protocol.*

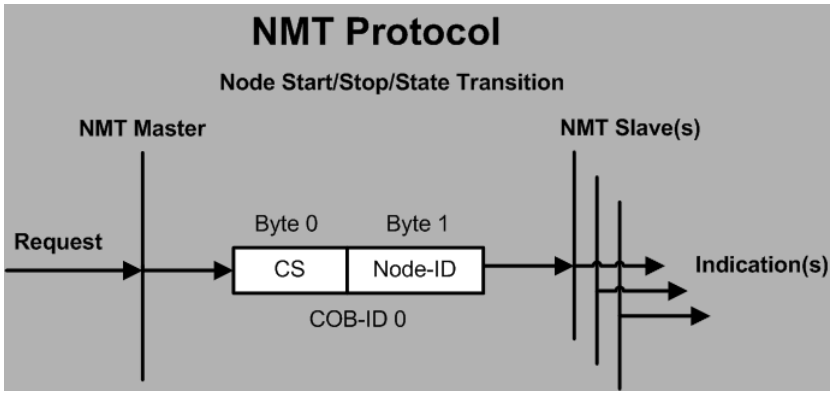


Figure 14: NMT Protocol

In this figure, CS (Command Specifier) means the value of variable `ulNmtCommand`.

When the application receives the CANOPEN_SLAVE_NMT_COMMAND_IND indication, it should try to do everything required to set the slave into the state having been requested. However, this may not be possible in all situations. So there is the possibility to leave the slave in another state than the requested one.

In any case, the application needs to send a response packet. The current state after processing the CANOPEN_SLAVE_NMT_COMMAND_IND indication needs to be supplied in the response packets `ulNmtState` variable. As NMT is an unconfirmed service, the response is not sent to the CANopen Master, but required for internal use at the CANopen Slave.

Packet Structure Reference

```
/*****************************************************************************/
 * NMT states for CANOPEN SLAVE SET NMT STATE REQ,
 * CANOPEN SLAVE NMT STATE CHANGE IND and
 * CANOPEN SLAVE NMT COMMAND IND
/*****************************************************************************/
#define CANOPEN_SLAVE_NMT_STATE_OPERATIONAL       0x01
#define CANOPEN_SLAVE_NMT_STATE_STOP              0x02
#define CANOPEN_SLAVE_NMT_STATE_PRE_OPERATIONAL   0x80
#define CANOPEN_SLAVE_NMT_STATE_RESET_NODE        0x81
#define CANOPEN_SLAVE_NMT_STATE_RESET_COMM        0x82
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_NMT_COMMAND_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_NMT_COMMAND_IND_DATA_Ttag
  CANOPEN_SLAVE_NMT_COMMAND_IND_DATA_T;

 struct CANOPEN_SLAVE_NMT_COMMAND_IND_DATA_Ttag
{
  TLR_UINT32 ulNmtCommand;  /* Request NMT command from NMT master */

};/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_NMT_COMMAND_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_NMT_COMMAND_IND_Ttag
 CANOPEN_SLAVE_PACKET_NMT_COMMAND_IND_T;

struct CANOPEN_SLAVE_PACKET_NMT_COMMAND_IND_Ttag
{
  TLR_PACKET_HEADER_T                  tHead; /** packet header. */
  CANOPEN_SLAVE_NMT_COMMAND_IND_DATA_T tData; /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	4	Packet Data Length in bytes
ulCmd	UINT32	0x2932	CANOPEN_SLAVE_NMT_COMMAND_IND
Data			
ulNmtCommand	UINT32	Valid NMT command	Requested NMT command from NMT master

Table 64: CANOPEN_SLAVE_PACKET_NMT_COMMAND_IND_T - NMT Command Indication

Packet Structure Reference

```
/*****************************************************************************/
typedef struct CANOPEN_SLAVE_NMT_COMMAND_RES_DATA_Ttag
  CANOPEN_SLAVE_NMT_COMMAND_RES_DATA_T;

 struct CANOPEN_SLAVE_NMT_COMMAND_RES_DATA_Ttag
{
  TLR_UINT32 ulNmtState;  /* New local NMT state */

};

/*****************************************************************************//**
type of <code>CANOPEN_SLAVE_PACKET_NMT_COMMAND_RES_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_NMT_COMMAND_RES_Ttag
 CANOPEN_SLAVE_PACKET_NMT_COMMAND_RES_T;

struct CANOPEN_SLAVE_PACKET_NMT_COMMAND_RES_Ttag
{
  TLR_PACKET_HEADER_T                  tHead; /** packet header. */
  CANOPEN_SLAVE_NMT_COMMAND_RES_DATA_T tData; /** packet data.   */

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32		Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2933	CANOPEN_SLAVE_NMT_COMMAND_RES
Data			
ulNmtState	UINT32	Valid NMT State	New local NMT state

Table 65: CANOPEN_SLAVE_PACKET_NMT_COMMAND_RES_T – Response to NMT Command Indication

3.2.16 CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ/CNF – Setup PDO Indication

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

The parameter `ulSetupPdoIndication` can be used to decide whether you want to disable (1) or enable PDO indications (2 or 3) and whether you want to send multiple PDOs within a single packet

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_Ttag
  CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_T;

#define CANOPEN_SLAVE_SETUP_PDO_INDICATION_DISABLE    0x000000001L
#define CANOPEN_SLAVE_SETUP_PDO_INDICATION_ENABLE     0x000000002L
#define CANOPEN_SLAVE_SETUP_PDO_INDICATION_ENABLE_EXT 0x000000003L

struct CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_Ttag
{
  TLR_UINT32 ulSetupPdoIndication; /* Parameter for PDO indications*/

};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_T;

struct CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_Ttag
{
  TLR_PACKET_HEADER_T                          tHead; /** packet header. */
  CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ_DATA_T tData; /** packet data.   */

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulLen	UINT32	4	Packet Data Length in bytes
ulCmd	UINT32	0x29BA	`CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ`
Data			
ulSetupPdoIndication	UINT32	0x00000001 0x00000002 0x00000003	*Disable PDO Indications* *Enable PDO Indication* *Enable PDO Indication (for multiple PDO transfer)*

Table 66: `CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_REQ_T` – Setup PDO Indication Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x29BB	CANOPEN_SLAVE_SETUP_PDO_INDICATION_CNF

Table 67: `CANOPEN_SLAVE_PACKET_SETUP_PDO_INDICATION_CNF_T` – Setup PDO Indication Confirmation

3.2.17 CANOPEN_SLAVE_RECEIVE_PDO_IND/RES – Receive PDO Indication

The following indication is sent from the CANopen slave to the application each time a PDO is received. The indication includes

- the PDO number,
- the identifier (COB-ID),
- the length and
- the data.

Note: No PDO indications are sent to the application until this functionality is explicitly enabled. Enabling PDO indications is described in this manual in section CANOPEN_SLAVE_SETUP_PDO_INDICATION_REQ/CNF – Setup PDO Indication. Enabling can be done for single or multiple PDO transfer with one packet.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_Ttag
  CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_T;
/** type of <code>CANOPEN_SLAVE_PDO_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_PDO_DATA_Ttag
  CANOPEN_SLAVE_PDO_DATA_T;

#define CANOPEN_SLAVE_RECEIVE_PDO_IND_MAX       16
#define CANOPEN_SLAVE_RECEIVE_PDO_IND_MAX_DATA  8

struct CANOPEN_SLAVE_PDO_DATA_Ttag
{
  TLR_UINT32 ulPdoNumber;                                    /* PDO number     */
  TLR_UINT32 ulIdentifier;                                   /* CAN identifier */
  TLR_UINT32 ulLength;                                       /* Data length    */
  TLR_UINT8  abPdoData[CANOPEN_SLAVE_RECEIVE_PDO_IND_MAX_DATA]; /* PDO data       */
};

struct CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_Ttag
{
  CANOPEN_SLAVE_PDO_DATA_T atPdoData[CANOPEN_SLAVE_RECEIVE_PDO_IND_MAX];
}; /*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_Ttag
  CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_T;

struct CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_Ttag
{
  TLR_PACKET_HEADER_T                 tHead; /** packet header. */
  CANOPEN_SLAVE_RECEIVE_PDO_IND_DATA_T tData; /** packet data.   */

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	12.. 20	Packet Data Length in bytes
ulCmd	UINT32	0x000029BC	CANOPEN_SLAVE_RECEIVE_PDO_IND
Data			
ulPdoNumber	UINT32	1..255	Number of received PDO
ulIdentifier	UINT32	0..2047	Identifier (COB-ID) of received PDO
ulLength	UINT32	0..8	Length of received PDO
abPdoData[8]	UINT8[8]		Data of received PDO

Table 68: CANOPEN_SLAVE_PACKET_RECEIVE_PDO_IND_T – Receive PDO Indication

Packet Structure Reference

```
typedef struct CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_Ttag
  CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x000029BD	CANOPEN_SLAVE_RECEIVE_PDO_RE

Table 69: CANOPEN_SLAVE_PACKET_RECEIVE_PDO_RES_T – Receive PDO Response

3.3 Hardware Switches for the Adjustment of Slave Address and Baudrate

For handling address and baud rate switches on a netX device, the firmware must be enabled to evaluate the switch values from the hardware. This can be done by setting a special TAG via the “Tag List Editor” tool. In this case, the values which are configured via database or packet interface will be ignored.

If the hardware switch function is not enabled via TAG, then the firmware uses the values set either via NXD or IniBatch database or via packet interface (SET_CONFIGURATION_REQ or RCX_SET_FW_PARAMETER_REQ)

Enabling and disabling Address and Baudrate Switching

On database files and SET_CONFIGURATION_REQ evaluating the switches can be activated or deactivated. This information is located at the System Flags

- CANOPEN_APS_SYS_FLAG_ADDRESS_SWITCH = 0x10
- CANOPEN_APS_SYS_FLAG_BAUD_SWITCH = 0x20.

Also see section 3.1.1 “CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration”.

Behavior at Start-up

In general, the firmware stack can be configured in different ways. Only one type of configuration can be active at a certain time. These are evaluated at start-up in the following order:

- SYCON.net database
- iniBatch database (via netX Configuration Tool)
- Warmstart Request packet (compatibility)
- Set Configuration Request packet

After a Restart the stack will first search for the SYCON.net database files (config.nxd). If these are found all other configuration methods will not be accepted. If no SYCON.net database exists, but an iniBatch database exists, its configuration will be used and configuration packets will be not accepted.

If no database is found the stack is unconfigured until the receipt of the first configuration packet. In this case the firmware waits for the SET_CONFIGURATION_REQ or WARMSTART_REQ packet. Once one of these packets (i.e. SET_CONFIGURATION_REQ) was received, the other one (i.e. WARMSTART_REQ) will be rejected.

The host has the possibility to modify the configuration with the packet RCX_SET_FW_PARAMETER_REQ/CNF (Set the Value of the Firmware Parameter).

Using the hardware switches for adjusting of slave address and baudrate requires the option *Application_controlled* being active (either when configuring using SET_CONFIGURATION_REQ packet (see CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration on page 42) or in the SYCON.net database file config.nxd).

The stack will start the device with the received configuration as soon as the application ready flag is set by the host application.

On starting the stack the hardware switches are evaluated if hardware switches are enabled via the TAG. The values from the hardware switches will overwrite the values, which was set via database or packet previously. This can be avoided if the hardware switches are disabled via the “Tag List Editor” tool. A description of the “Tag List Editor” tool is given in reference [3].

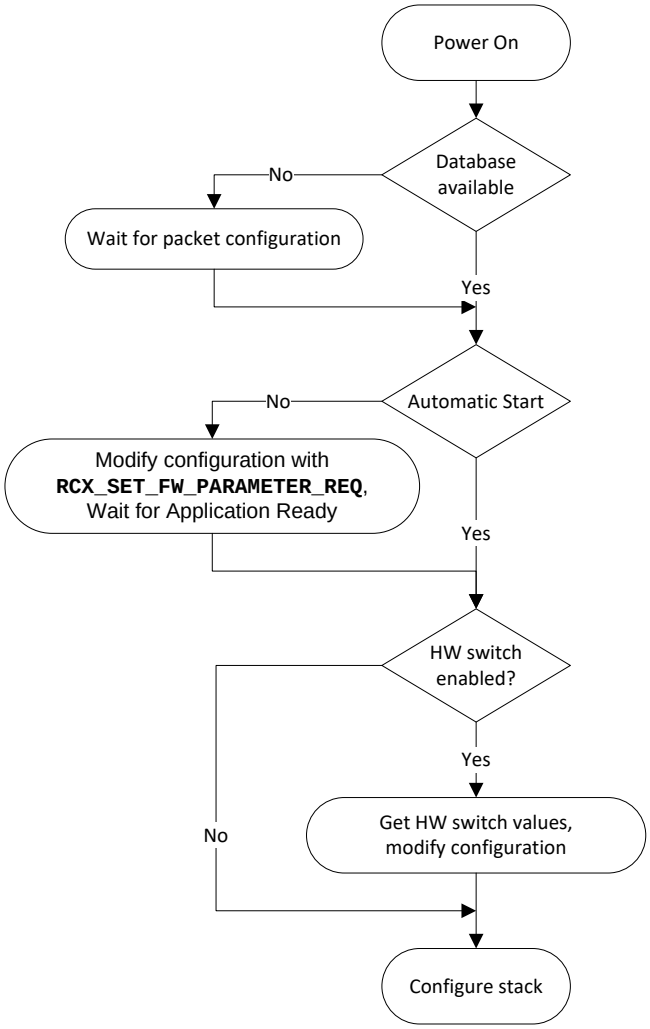


Figure 15: Start-up Process

3.4 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 [6].

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 Slave 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.

3.5 ODV3 Task

If working with Loadable Firmware, you can also use the SDO functionality of the ODV3 Task for accessing the object dictionary.

The packet interface of the Object Dictionary V3 is described within a separate manual, the Object Dictionary V3 Protocol API Manual. See reference # 4.

The following packets of Object Dictionary V3 can be used without any restrictions:

- ODV3_READ_OBJECT_REQ
- ODV3_WRITE_OBJECT_REQ
- ODV3_GET_OBJECT_LIST_REQ
- ODV3_GET_OBJECT_INFO_REQ
- ODV3_GET_SUBOBJECT_INFO_REQ
- ODV3_GET_OBJECT_ACCESS_INFO_REQ
- ODV3_GET_OBJECT_SIZE_REQ
- ODV3_READ_OBJECT_NO_IND_REQ
- ODV3_GET_OBJECT_COUNT_REQ
- ODV3_WRITE_ALL_BY_INDEX_REQ
- ODV3_READ_ALL_BY_INDEX_REQ
- ODV3_WRITE_MULTIPLE_PARAMETER_BY_INDEX_REQ
- ODV3_READ_MULTIPLE_PARAMETER_BY_INDEX_REQ
- ODV3_GET_TIMEOUT_REQ

The following packets of Object Dictionary V3 will be denied as long as the ‘Configuration Lock’ flag is set:

- ODV3_CREATE_OBJECT_REQ
- ODV3_CREATE_SUBOBJECT_REQ
- ODV3_DELETE_OBJECT_REQ
- ODV3_DELETE_SUBOBJECT_REQ
- ODV3_REGISTER_OBJECT_NOTIFY_REQ
- ODV3_UNREGISTER_OBJECT_NOTIFY_REQ
- ODV3_REGISTER_SUBOBJECT_NOTIFY_REQ
- ODV3_UNREGISTER_SUBOBJECT_NOTIFY_REQ
- ODV3_REGISTER_UNDEFINED_NOTIFY_REQ
- ODV3_UNREGISTER_UNDEFINED_NOTIFY_REQ
- ODV3_REGISTER_OBJINFO_NOTIFY_REQ
- ODV3_UNREGISTER_OBJINFO_NOTIFY_REQ
- ODV3_CREATE_DATATYPE_REQ
- ODV3_DELETE_DATATYPE_REQ
- ODV3_SET_TIMEOUT_REQ
- ODV3_SET_OBJECT_NAME_REQ
- ODV3_SET_SUBOBJECT_NAME_REQ
- ODV3_LOCK_OBJECT_DELETION_REQ
- ODV3_UNLOCK_OBJECT_DELETION_REQ

Whether or not indications are sent to your application, depends on which ODV3 events have been notified!

4 Status information

4.1 Extended Status

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

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

Extended Status Block

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

Table 70: Extended Status Block (General Structure)

Extended Status Block Structure

```
typedef struct CANOPEN_SLAVE_EXTENDED_STATE_Ttag
  CANOPEN_SLAVE_EXTENDED_STATE_T;

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

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

#define CANOPEN_SLAVE_ADD_DETAIL_SIZE             0x00000003L

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

  TLR_UINT32 aulReserved[4];

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

4.2 Extended Status Block

Additional Info Block for CANopen Slave

Offset	Type	Name	Description
0x50	unsigned long	ulFlags	Bit field for Flags
0x54	unsigned long	ulNodeState	Current Node State
0x58	unsigned long	ulBusOffEveCnt	Counter for bus off events
0x5C	unsigned long	ulErrorPassiveEveCnt	Counter for passive event errors
0x60	unsigned long	ulRxOverflowCnt	Counter for receive overflows
0x64	unsigned long	ulTxOverflowCnt	Counter for transmit overflows
0x68	unsigned long	ulErrorWarningCnt	Count of errors and Warnings
0x6C	unsigned long	ulTimeoutCnt	Number of timeouts
0x70	unsigned long[]	aulReserved[3]	Reserved for further use
0x7C	unsigned long	ulIndLostCnt	Count of Lost Indications
0x80	unsigned long	ulDiagInfoCount	Number of diagnostic entries
0x84	unsigned long	ulLastDiagInfo	Last diagnostic entry
0x88	unsigned long	ulMaxRecvIdx	Maximum Object Index Value for Receive Data
0x8C	unsigned long	ulMaxSendIdx	Maximum Object Index Value for Send Data
0x90	unsigned long[]	aulAddDetail[3]	Additional detail for diagnostic entry

Table 71: Additional Info Block

ulFlags

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

Bit	Name	Description
D31.. D17	Reserved	Reserved for further use
D16	CANOPEN_SLAVE_EXT_STATE_ FLAG_WARNING	A warning has been issued
D15	Reserved	Reserved for further use
D14	CANOPEN_SLAVE_EXT_STATE_ TIMEOUT	The DEVICE has detected an overstepped timeout supervision time of at least one CAN message to be sent. The transmission of this message was aborted. The data is lost. Its an indication that no other CAN device was connected or responsive at this time to acknowledge the sent message requests. The bit will be set when the first timeout was detected and will not be deleted any more.
D13	CANOPEN_SLAVE_EXT_STATE_ NRDY	Indicates if the HOST program has set its state to operative or not. If the bit is set the HOST program is not ready to communicate.
D12	CANOPEN_SLAVE_EXT_STATE_ CTRL	Parameterization error or severe run time error
D11.. D10	Reserved	Reserved for further use
D9	CANOPEN_SLAVE_EXT_STATE_ FLAG_SLAVE_ERROR	An error has been issued from the slave
D8	CANOPEN_SLAVE_EXT_STATE_ FLAG_WDG	Watchdog error detected
D7	Reserved	Reserved for further use
D6	CANOPEN_SLAVE_EXT_STATE_ FLAG_IND_LOST	Indication has been lost
D5	CANOPEN_SLAVE_EXT_STATE_ FLAG_TX_OVERFLOW	Transmit overflow detected
D4	CANOPEN_SLAVE_EXT_STATE_ FLAG_RX_OVERFLOW	Receive overflow detected
D3	CANOPEN_SLAVE_EXT_STATE_ FLAG_BUS_OFF	CAN is in Bus-off state
D2	CANOPEN_SLAVE_EXT_STATE_ FLAG_PASSIVE	CAN is in error passive state
D1	CANOPEN_SLAVE_EXT_STATE_ FLAG_CAN_ACTIVE	CAN is activated
D0	CANOPEN_SLAVE_EXT_STATE_ FLAG_CAN_INIT	CAN is initialized

Table 72: Additional Info Flags

ulNodeState

Internal node state of node:

0: Unknown state

1: Operational state

2: Stop

128: Pre-operational state

255: Initializing

ulBusOffEveCnt

Number of Bus-off events

ulErrorPassiveEveCnt

Number of error passive events

ulRxOverflowCnt

Number of receive overrun events

ulTxOverflowCnt

Number of transmit overrun events

ulErrorWarningCnt

Total count of errors and warnings

ulTimeoutCnt

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

aulReserved[ ]

Reserved for further use

ulIndLostCnt

Count of lost indications

ulDiagInfoCount

Number of diagnostic entries

ulLastDiagInfo

Last diagnostic entry

ulMaxRecvIdx

Number of highest PDO mapped receive object index

ulMaxSendIdx

Number of highest PDO mapped send object index

aulAddDetail[ ]

Additional detail for diagnostic entry

5 Special Topics

5.1 Using LOM

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

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

Table 73: APM-Task Process Queue

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

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

ASCII queue name	Description
"QUE_CANOPENSLV"	Name of the CANopen slave-task process queue
"QUE_COS_ODV3"	Name of the CANopen slave-task ODV3 process queue

Table 74 CANopen Slave-Task Process Queue

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

5.2 Packages for LOM

5.2.1 Overview

The following table lists the packages which can be used with Linkable Object Modules (LOM) only. Additionaly, the table lists whether a package will be denied if the '*Configuration Lock'* flag is set.

Section	Name	LFW	LOM	Denied on Cfg. Lock
5.2.2	CANOPEN_SLAVE_REGISTER_REQ/CNF – Register Application	0	x	x
5.2.3	CANOPEN_SLAVE_INITIALIZE_REQ/CNF – Initialization of CANopen Slave	0	x	x
5.2.4	CANOPEN_SLAVE_STATE_CHANGE_IND/RES – Change of Task State Indication	0	x	x
5.2.5	CANOPEN_SLAVE_SET_BUSPARAM_REQ/CNF – Set Bus Parameters	0	x	0
5.2.6	CANOPEN_SLAVE_SET_API_PARAM_REQ/CNF – Set API Parameter	0	x	x
5.2.7	CANOPEN_SLAVE_GET_BUSPARAM_REQ/CNF – Get Bus Parameters	0	x	x
5.2.8	CANOPEN_SLAVE_SET_WATCHDOG_FAIL_REQ/CNF – Set Watchdog Fail	0	x	x

Table 75: Packets of CANopen Slave Protocol Stack V3 and Restrictions of Usage

5.2.2 CANOPEN_SLAVE_REGISTER_REQ/CNF – Register Application

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

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

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

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_REGISTER_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_REGISTER_REQ_DATA_Ttag
  CANOPEN_SLAVE_REGISTER_REQ_DATA_T;

struct CANOPEN_SLAVE_REGISTER_REQ_DATA_Ttag
{
  TLR_UINT8 bReserved; /* Reserved for further use, set to zero*/

};
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_REGISTER_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_REGISTER_REQ_Ttag
  CANOPEN_SLAVE_PACKET_REGISTER_REQ_T;

/** Structure of task command application register request*/
struct CANOPEN_SLAVE_PACKET_REGISTER_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;  /** packet header.       */
  CANOPEN_SLAVE_REGISTER_REQ_DATA_T tData;  /** packet request data. */

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	QUE_CANOPENSLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulDestId	UINT32	ulCANOPENSLV0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulCmd	UINT32	0x2900	`CANOPEN_SLAVE_REGISTER_REQ`
Data			
bReserved	UINT32	0	Reserved for further use, set to zero

Table 76: `CANOPEN_SLAVE_PACKET_APP_REGISTER_REQ_T` – Register Application Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_REGISTER_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_REGISTER_CNF_DATA_Ttag
  CANOPEN_SLAVE_REGISTER_CNF_DATA_T;

struct CANOPEN_SLAVE_REGISTER_CNF_DATA_Ttag
{
  TLR_UINT8 bReserved; /* Reserved for further use */
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_REGISTER_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_REGISTER_CNF_Ttag
  CANOPEN_SLAVE_PACKET_REGISTER_CNF_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
ulLen	UINT32	4	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *"Codes of the CANopen Slave-Task"*
ulCmd	UINT32	0x2901	CANOPEN_SLAVE_REGISTER_CNF
Data			
bReserved	UINT32	0	Reserved for further use, set to zero

Table 77: CANOPEN_SLAVE_PACKET_APP_REGISTER_CNF_T – Register Application Confirmation

5.2.3 CANOPEN_SLAVE_INITIALIZE_REQ/CNF – Initialization of CANopen Slave

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

You can read in section 2.3.1 “*NMT Slave State Machine*” what happens in detail during the initialization process of the CANopen Slave protocol stack.

If you are interested to know in detail what happens during the initialization process of the CANopen Slave protocol stack, you should read section 2.3.1 “*NMT Slave State Machine*”.

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

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

Packet Structure Reference

```
/** type of <code>CANOPEN_SLAVE_INITIALIZE_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_INITIALIZE_REQ_DATA_Ttag
  CANOPEN_SLAVE_INITIALIZE_REQ_DATA_T;

/** Structure of task command delete configuration CANopenrequest data */
struct CANOPEN_SLAVE_INITIALIZE_REQ_DATA_Ttag
{
  TLR_UINT32 ulReserved; /* Reserved fur further use, set to zero */
};

/** type of <code>CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_Ttag
  CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_T;

/** Structure of task command delete configuration request */
struct CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;  /** packet header.       */
  CANOPEN_SLAVE_INITIALIZE_REQ_DATA_T tData;  /** packet request data. */
};
```

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPEN SLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulSta	UINT32	0	See section 6.2 *“Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2904	CANOPEN_SLAVE_SLAVE_REQ
Data			
ulReserved	UINT32	0	Reserved for further use, set to zero

Table 78: CANOPEN_SLAVE_PACKET_INITIALIZE_REQ_T – Initialization of CANopen Slave Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_INITIALIZE_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_INITIALIZE_CNF_DATA_Ttag
  CANOPEN_SLAVE_INITIALIZE_CNF_DATA_T;

/** Structure of task command delete configuration confirmation data */
struct CANOPEN_SLAVE_INITIALIZE_CNF_DATA_Ttag
{
  TLR_UINT32 ulReserved; /* Reserved fur further use */
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_Ttag
  CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_T;

/** Structure of task command delete configuration confirmation */
struct CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;  /** packet header.            */
  CANOPEN_SLAVE_INITIALIZE_CNF_DATA_T tData;  /** packet confirmation data. */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section 6.2 *"Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2905	CANOPEN_SLAVE_INITIALIZE_CNF
Data			
ulReserved	UINT32	0	Reserved for further use, set to zero

Table 79: CANOPEN_SLAVE_PACKET_INITIALIZE_CNF_T – Initialization of CANopen Slave Confirmation

5.2.4 CANOPEN_SLAVE_STATE_CHANGE_IND/RES – Change of Task State Indication

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

- The slave state
- The extended slave state

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

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

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

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

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_Ttag
  CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_T;

struct CANOPEN_SLAVE_STATE_CHANGE_IND_DATA_Ttag
{
  CANOPEN_SLAVE_SLAVE_STATE_T    tSlaveState;     /* Slave state    */
  CANOPEN_SLAVE_EXTENDED_STATE_T tExtendedState;  /* Extended state */
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_Ttag
  CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	112	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulCmd	UINT32	0x2912	CANOPEN_SLAVE_STATE_CHANGE_IND
Data			
tSlaveState	CANOPEN_SLAVE_SLAVE_STATE_T		Structure for slave state, see explanation below.
tExtendedState	CANOPEN_SLAVE_EXTENDED_STATE_T		Structure for extended slave state, see explanation below.

Table 80: CANOPEN_SLAVE_PACKET_STATE_CHANGE_IND_T – Change of State Indication

CANopen Slave State Structure Reference

```
typedef struct CANOPEN_SLAVE_SLAVE_STATE_Ttag
  CANOPEN_SLAVE_SLAVE_STATE_T;

#define CANOPEN_SLAVE_CAN_STATE_UNKNOWN          0x00000000L
#define CANOPEN_SLAVE_CAN_STATE_NOT_CONFIGURED   0x00000001L
#define CANOPEN_SLAVE_CAN_STATE_STOPPED          0x00000002L
#define CANOPEN_SLAVE_CAN_STATE_STARTED          0x00000003L
#define CANOPEN_SLAVE_CAN_STATE_RUNNING          0x00000004L

#define CANOPEN_SLAVE_STATE_FLAG_RDY             0x00000001L
#define CANOPEN_SLAVE_STATE_FLAG_RUN             0x00000002L
#define CANOPEN_SLAVE_STATE_FLAG_COM             0x00000004L
#define CANOPEN_SLAVE_STATE_FLAG_BUS_ON          0x00000008L
#define CANOPEN_SLAVE_STATE_FLAG_COMM_ERROR      0x00000010L
#define CANOPEN_SLAVE_STATE_FLAG_CAN_STARTED     0x00000100L

struct CANOPEN_SLAVE_SLAVE_STATE_Ttag
{
  TLR_UINT32  ulCanState;

  TLR_UINT32  aulUnused[2];

  TLR_UINT32  ulFlags;

  TLR_UINT32  ulErrorCount;
  TLR_UINT32  ulCommError;

  TLR_UINT32  ulRunLedState;
  TLR_UINT32  ulErrLedState;

  TLR_UINT32  ulRecvDataCnt;
  TLR_UINT32  ulSendDataCnt;

  TLR_UINT32  ulReserved;

};
```

ulCanState

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

Bit	Name	Description
D4	CANOPEN_SLAVE_CAN_STATE_RUNNING	CAN State is *Running*
D3	CANOPEN_SLAVE_CAN_STATE_STARTED	CAN State is *Started*
D2	CANOPEN_SLAVE_CAN_STATE_STOPPED	CAN State is *Stopped*
D1	CANOPEN_SLAVE_CAN_STATE_NOT_CONFIGURED	CAN State is *Not configured*
D0	CANOPEN_SLAVE_CAN_STATE_UNKNOWN	CAN State is unknown

Table 81: Flags of `ulCanState`

aulUnused[2]

This array has been introduced only for compatibility reasons. It is not used.

ulFlags

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

Bit	Name	Description
D32...D9	-	Unused
D8	CANOPEN_SLAVE_STATE_FLAG_CAN_STARTED	CAN network has been started
D7…D5	-	Unused
D4	CANOPEN_SLAVE_STATE_FLAG_COMM_ERROR	Communication Error
D3	CANOPEN_SLAVE_STATE_FLAG_BUS_ON	Bus on
D2	CANOPEN_SLAVE_STATE_FLAG_COM	Communication running
D1	CANOPEN_SLAVE_STATE_FLAG_RUN	Run
D0	CANOPEN_SLAVE_STATE_FLAG_RDY	Ready

Table 82: Bit field ulFlags

ulErrorCount

This field contains 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.

ulCommError

This field contains the error code of the last communication error that occurred.

ulRunLedState

This field contains the current state of the RUN LED.

ulErrLedState

This field contains the current state of the ERR LED.

ulRecvDataCnt

This field contains the Received Data Count

ulSendDataCnt

This field contains the Send Data Count.

ulReserved

This field is reserved for future use.

Extended Slave State Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_EXTENDED_STATE_Ttag</code> */
typedef struct CANOPEN_SLAVE_EXTENDED_STATE_Ttag
  CANOPEN_SLAVE_EXTENDED_STATE_T;

#define CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_INIT     0x00000001L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_CAN_ACTIVE   0x00000002L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_PASSIVE      0x00000004L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_BUS_OFF      0x00000008L

#define CANOPEN_SLAVE_EXT_STATE_FLAG_RX_OVERFLOW  0x00000010L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_TX_OVERFLOW  0x00000020L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_IND_LOST     0x00000040L

#define CANOPEN_SLAVE_EXT_STATE_FLAG_WDG          0x00000100L
#define CANOPEN_SLAVE_EXT_STATE_FLAG_SLAVE_ERROR  0x00000200L

#define CANOPEN_SLAVE_EXT_STATE_CTRL              0x00001000L
#define CANOPEN_SLAVE_EXT_STATE_NRDY              0x00002000L
#define CANOPEN_SLAVE_EXT_STATE_TIMEOUT           0x00004000L

#define CANOPEN_SLAVE_EXT_STATE_FLAG_WARNING      0x00010000L

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

#define CANOPEN_SLAVE_ADD_DETAIL_SIZE             0x00000003L

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

  TLR_UINT32 aulReserved[3];
  TLR_UINT32 ulIndLostCnt;

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

};

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

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_Ttag
  CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulCANOPENSLV0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *"Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2913	`CANOPEN_SLAVE_STATE_CHANGE_RES`

Table 83: CANOPEN_SLAVE_PACKET_STATE_CHANGE_RES_T – Change of State Response

5.2.5 CANOPEN_SLAVE_SET_BUSPARAM_REQ/CNF – Set Bus Parameters

This packet can be applied for setting the bus parameters for the CANopen Slave.

Note: Use this packet only when working with linkable object modules. It is not designed for usage in the context of loadable firmware. In the context of loadable firmware we recommend to use ‘set configuration’ instead.

Note: This request will be denied if the configuration lock flag is set.

All parameters used by this packet are also used by the packet CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration with the same meaning and in the same way. For more information about these parameters, see Table 26 on page 44.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_STD_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_STD_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_STD_BUSPARAM_DATA_T;
/** type of <code>CANOPEN_SLAVE_EXT_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_EXT_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T;
/** type of <code>CANOPEN_SLAVE_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_BUSPARAM_DATA_T;

/*------ Common configuration flags ---------------------------------------*/
#define CANOPEN_SLAVE_COMMON_FLAG_CFG_EXT_MODE                  0x10000000L
#define CANOPEN_SLAVE_COMMON_FLAG_CFG_HOLD_LAST_STATE           0x01000000L
#define CANOPEN_SLAVE_COMMON_FLAG_CFG_REJECT_RESTRICTED_CAN_ID  0x00100000L
#define CANOPEN_SLAVE_COMMON_FLAG_CFG_DISABLE_SEND_COS_SYNC_ACYC 0x00010000L
#define CANOPEN_SLAVE_COMMON_FLAG_CFG_DISABLE_SEND_COS_MAN_SPEC  0x00020000L
#define CANOPEN_SLAVE_COMMON_FLAG_CFG_DISABLE_SEND_COS_PROF_SPEC 0x00040000L
/*-----------------------------------------------------------------------*/

/*------ Configuration flags and for standard mode only -------------------*/
#define CANOPEN_SLAVE_STD_FLAG_CFG_VENDOR_ID                    0x00000010L
#define CANOPEN_SLAVE_STD_FLAG_CFG_PRODUCT_CODE                 0x00000020L
#define CANOPEN_SLAVE_STD_FLAG_CFG_SERIAL_NUMBER                0x00000040L
#define CANOPEN_SLAVE_STD_FLAG_CFG_REVISION_NUMBER              0x00000080L
#define CANOPEN_SLAVE_STD_FLAG_CFG_DEVICE_TYPE                  0x00000100L
#define CANOPEN_SLAVE_STD_FLAG_CFG_OBJECT_SIZE                  0x00000200L
#define CANOPEN_SLAVE_STD_FLAG_CFG_PDO_CNT                      0x00000400L
/*-----------------------------------------------------------------------*/

/*------ Configuration flags for extended mode only -----------------------*/

/*-----------------------------------------------------------------------*/

#define CANOPEN_SLAVE_MIN_SLAVE_NODE_ID        1            /* Minimum node ID       */
#define CANOPEN_SLAVE_MAX_SLAVE_NODE_ID        127          /* Maximum node ID       */

#define CANOPEN_SLAVE_CFG_BAUD_1000            0x00000000L  /* 1MBaud                */
#define CANOPEN_SLAVE_CFG_BAUD_800             0x00000001L  /* 800kBaud              */
#define CANOPEN_SLAVE_CFG_BAUD_500             0x00000002L  /* 500kBaud              */
#define CANOPEN_SLAVE_CFG_BAUD_250             0x00000003L  /* 250kBaud              */
#define CANOPEN_SLAVE_CFG_BAUD_125             0x00000004L  /* 125kBaud              */
#define CANOPEN_SLAVE_CFG_BAUD_100             0x00000005L  /* 100kBaud              */
#define CANOPEN_SLAVE_CFG_BAUD_50              0x00000006L  /* 50kBaud               */
#define CANOPEN_SLAVE_CFG_BAUD_20              0x00000007L  /* 20kBaud               */
#define CANOPEN_SLAVE_CFG_BAUD_10              0x00000008L  /* 10kBaud               */
#define CANOPEN_SLAVE_CFG_BAUD_AUTO_DETECTION  0x000000FFL  /* Auto-Baud detection   */
```

```
#define CANOPEN_SLAVE_STD_CFG_DEF_OBJECT_SIZE  128 /* Default object size in std mode */

#define CANOPEN_SLAVE_CFG_MAX_RXPDO            256 /* Maximum number of RxPDO         */
#define CANOPEN_SLAVE_CFG_MAX_TXPDO            256 /* Maximum number of TxPDO         */

__PACKED_PRE struct CANOPEN_SLAVE_STD_BUSPARAM_DATA_Ttag
{
  TLR_UINT32 ulVendorId;       /* Vendor ID               */
  TLR_UINT32 ulProductCode;    /* Product code            */
  TLR_UINT32 ulSerialNumber;   /* Serial number           */
  TLR_UINT32 ulRevisionNumber; /* Revision number         */
  TLR_UINT32 ulDeviceType;     /* Device Type             */

  TLR_UINT8  bObject2000Size;  /* Size of object 2000     */
  TLR_UINT8  bObject2001Size;  /* Size of object 2001     */
  TLR_UINT8  bObject2002Size;  /* Size of object 2002     */
  TLR_UINT8  bObject2003Size;  /* Size of object 2003     */

  TLR_UINT8  bObject2200Size;  /* Size of object 2200     */
  TLR_UINT8  bObject2201Size;  /* Size of object 2201     */
  TLR_UINT8  bObject2202Size;  /* Size of object 2202     */
  TLR_UINT8  bObject2203Size;  /* Size of object 2203     */

  TLR_UINT16 usNumOfRxPdo;     /* Number of receive PDOs  */
  TLR_UINT16 usNumOfTxPdo;     /* Number of transmit PDOs */

  TLR_UINT32 aulReserved[2];   /* Reserved, set to zero   */

}__PACKED_POST;

__PACKED_PRE struct CANOPEN_SLAVE_EXT_BUSPARAM_DATA_Ttag
{
  TLR_UINT16 usNumOfRxPdo;     /* Number of receive PDOs  */
  TLR_UINT16 usNumOfTxPdo;     /* Number of transmit PDOs */
  TLR_UINT32 aulReserved[9];   /* Reserved, set to zero   */

}__PACKED_POST;

/** Structure of task command set bus param data */
__PACKED_PRE struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
{
  TLR_UINT32 ulSlaveNodeId;  /* Node ID       */
  TLR_UINT32 ulBaudrate;     /* Baud-rate     */
  TLR_UINT32 ulCanOpenFlags; /* CANopen flags */

  __PACKED_PRE union
  {
    CANOPEN_SLAVE_STD_BUSPARAM_DATA_T tStdBusParam; /* Parameter for standard mode*/
    CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T tExtBusParam; /* Parameter for extended mode*/

  }__PACKED_POST uMode;

  TLR_UINT32 ul29BitCode; /* 29Bit Code */
  TLR_UINT32 ul29BitMask; /* 29Bit Mask */

}__PACKED_POST;

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_SET_BUSPARAM_REQ_DATA_T;

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_GET_BUSPARAM_CNF_DATA_T;
```

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	QUE_CANOPENSLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulDestId	UINT32	ulCANOPENSLV0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	80	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulCmd	UINT32	0x2906	*CANOPEN_SLAVE_SET_BUSPARAM_REQ*
Data			
tBusParam	CANOPEN_SLAVE_BUSPARAM_DATA_T		Bus parameter structure, see Table 26 on page 44.

Table 84: CANOPEN_SLAVE_PACKET_SET_BUSPARAM_REQ_T – Set Bus Parameters Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_T;

/** Structure of task command set bus parameter confirmation */
struct CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;                    /** packet header.       */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2907	CANOPEN_SLAVE_SET_BUSPARAM_CNF

Table 85: CANOPEN_SLAVE_PACKET_SET_BUSPARAM_CNF_T –Set Bus Parameter Confirmation

5.2.6 CANOPEN_SLAVE_SET_API_PARAM_REQ/CNF – Set API Parameter

This packet can be used to register callbacks for data exchange between Slave and APS task at the CANopen Slave.

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

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_SET_API_PARAM_REQ_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_SET_API_PARAM_REQ_DATA_Ttag
  CANOPEN_SLAVE_SET_API_PARAM_REQ_DATA_T;

typedef TLR_VOID (CALLBACK FAR* PFN_CANOPEN_SLAVE_CALLBACK)
(
TLR_HANDLE hApplication
);

struct CANOPEN_SLAVE_SET_API_PARAM_REQ_DATA_Ttag
{
  TLR_HANDLE hApplication;

  PFN_CANOPEN_SLAVE_CALLBACK pFncSendDataUpdated;
  TLR_UINT8 FAR* pbSrcData;
  TLR_UINT32 ulSrcDataCnt;

  PFN_CANOPEN_SLAVE_CALLBACK pFncRecvDataUpdated;
  TLR_UINT8 FAR* pbDstData;
  TLR_UINT32 ulDstDataCnt;
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_API_PARAM_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_API_PARAM_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SET_API_PARAM_REQ_T;

struct CANOPEN_SLAVE_PACKET_SET_API_PARAM_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead; /** packet header.         */
  CANOPEN_SLAVE_SET_API_PARAM_REQ_DATA_T    tData; /** packet request data.   */

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle
ulDestId	UINT32	ulCANOPENSLV0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulCmd	UINT32	0x292C	CANOPEN_SLAVE_SET_API_PARAM_REQ - Command
Data			
hApplication	TLR_HANDLE		Application Handle of Slave (see TLR Manual)
pFncSendDataUpdated	PFN_CANOPEN_SLAVE_CALLBACK	Callback	Callback for Send Data
pbSrcData	UINT8 FAR*	Pointer	Source Data Pointer
ulSrcDataCnt	UINT32	0 ... 2^{32}-1	Source Data Count
pFncRecvDataUpdated	PFN_CANOPEN_SLAVE_CALLBACK	Callback	Callback for Receive Data
pbDstData	UINT8 FAR*	Pointer	Destination Data Pointer
ulDstDataCnt	UINT32	0 ... 2^{32}-1	Destination Data Count

Table 86: `CANOPEN_SLAVE_PACKET_SET_API_PARAM_REQ_T` - Set API Parameter Request

Packet Structure Reference

```
/*****************************************************************************/
typedef struct CANOPEN_SLAVE_SET_API_PARAM_CNF_DATA_Ttag
  CANOPEN_SLAVE_SET_API_PARAM_CNF_DATA_T;

typedef TLR_VOID (CALLBACK FAR* PFN_CANOPEN_SLAVE_CALLBACK)
(
TLR_HANDLE hApplication
);

struct CANOPEN_SLAVE_SET_API_PARAM_CNF_DATA_Ttag
{
  TLR_HANDLE hSlave;
};

/** type of <code>CANOPEN_SLAVE_PACKET_SET_API_PARAM_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_API_PARAM_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SET_API_PARAM_CNF_T;

struct CANOPEN_SLAVE_PACKET_SET_API_PARAM_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead; /** packet header.            */
  CANOPEN_SLAVE_SET_API_PARAM_CNF_DATA_T tData; /** packet confirmation data. */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x292D	CANOPEN_SLAVE_SET_API_PARAM_CNF
Data			
hSlave	TLR_HANDLE		Slave Handle (see TLR Manual)

Table 87: CANOPEN_SLAVE_PACKET_SET_API_PARAM_CNF_T - Confirmation to Set API Parameter Request

5.2.7 CANOPEN_SLAVE_GET_BUSPARAM_REQ/CNF – Get Bus Parameters

This packet can be used to retrieve the current bus parameters

All parameters used by the confirmation packet are also used by the request packet CANOPEN_APS_SET_CONFIGURATION_REQ/CNF – Set Configuration with the same meaning and in the same way.

For a precise description of these parameters,, see *Table 26: Bus parameter structure CANOPEN_SLAVE_BUSPARAM_DATA_T* on page 44. and the description given there.

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_GET_BUSPARAM_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_GET_BUSPARAM_REQ_Ttag
  CANOPEN_SLAVE_PACKET_GET_BUSPARAM_REQ_T;

/** Structure of task command get bus parameter request */
struct CANOPEN_SLAVE_PACKET_GET_BUSPARAM_REQ_Ttag
{
  TLR_PACKET_HEADER_T tHead;  /** packet header.       */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPENSLV	Destination Queue-Handle
ulDestId	UINT32	ulCANOPENSLV0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32	0	See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2934	CANOPEN_SLAVE_GET_BUSPARAM_REQ

Table 88: CANOPEN_SLAVE_PACKET_GET_BUSPARAM_REQ_T - Get Bus Parameters Request

Packet Structure Reference

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

 struct CANOPEN_SLAVE_STD_BUSPARAM_DATA_Ttag
{
  TLR_UINT32 ulVendorId;       /* Vendor ID               */
  TLR_UINT32 ulProductCode;    /* Product code            */
  TLR_UINT32 ulSerialNumber;   /* Serial number           */
  TLR_UINT32 ulRevisionNumber; /* Revision number         */
  TLR_UINT32 ulDeviceType;     /* Device Type             */

  TLR_UINT8  bObject2000Size;  /* Size of object 2000     */
  TLR_UINT8  bObject2001Size;  /* Size of object 2001     */
  TLR_UINT8  bObject2002Size;  /* Size of object 2002     */
  TLR_UINT8  bObject2003Size;  /* Size of object 2003     */

  TLR_UINT8  bObject2200Size;  /* Size of object 2200     */
  TLR_UINT8  bObject2201Size;  /* Size of object 2201     */
  TLR_UINT8  bObject2202Size;  /* Size of object 2202     */
  TLR_UINT8  bObject2203Size;  /* Size of object 2203     */

  TLR_UINT16 usNumOfRxPdo;     /* Number of receive PDOs  */
  TLR_UINT16 usNumOfTxPdo;     /* Number of transmit PDOs */

  TLR_UINT32 aulReserved[2];   /* Reserved, set to zero   */
};

 struct CANOPEN_SLAVE_EXT_BUSPARAM_DATA_Ttag
{
  TLR_UINT16 usNumOfRxPdo;     /* Number of receive PDOs  */
  TLR_UINT16 usNumOfTxPdo;     /* Number of transmit PDOs */
  TLR_UINT32 aulReserved[9];   /* Reserved, set to zero   */
};

/*****************************************************************************/
/** Structure of task command set bus param data */
 struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
{
  TLR_UINT32 ulSlaveNodeId;  /* Node ID       */
  TLR_UINT32 ulBaudrate;     /* Baud-rate     */
  TLR_UINT32 ulCanOpenFlags; /* CANopen flags */

   union {
    CANOPEN_SLAVE_STD_BUSPARAM_DATA_T tStdBusParam; /* Parameter for standard mode*/
    CANOPEN_SLAVE_EXT_BUSPARAM_DATA_T tExtBusParam; /* Parameter for extended mode*/
  } uMode;

  TLR_UINT32 ul29BitCode; /* 29Bit Code */
  TLR_UINT32 ul29BitMask; /* 29Bit Mask */
};

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_BUSPARAM_DATA_Ttag</code> */
typedef struct CANOPEN_SLAVE_BUSPARAM_DATA_Ttag
  CANOPEN_SLAVE_GET_BUSPARAM_CNF_DATA_T;

/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_GET_BUSPARAM_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_GET_BUSPARAM_CNF_Ttag
  CANOPEN_SLAVE_PACKET_GET_BUSPARAM_CNF_T;

/** Structure of task command get bus parameter confirmation */
struct CANOPEN_SLAVE_PACKET_GET_BUSPARAM_CNF_Ttag
{
  TLR_PACKET_HEADER_T                  tHead;  /** packet header. */
  CANOPEN_SLAVE_GET_BUSPARAM_CNF_DATA_T tData;  /** packet data.   */
};
/*****************************************************************************/
```

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	60	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x2935	CANOPEN_SLAVE_GET_BUSPARAM_CNF
Data			
tBusParam	CANOPEN_SLAVE_BUSPARAM_DATA_T	Structure	Bus parameter structure, see *Table 26: Bus parameter structure CANOPEN_SLAVE_BUSPARAM_DATA_T*

Table 89: CANOPEN_SLAVE_PACKET_GET_BUSPARAM_CNF_T - Confirmation to Get Bus Parameters Request

5.2.8 CANOPEN_SLAVE_SET_WATCHDOG_FAIL_REQ/CNF – Set Watchdog Fail

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

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

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

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_Ttag
  CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	QUE_CANOPENSLV	Destination Queue-Handle of CANopen slave-Task Process Queue
ulDestId	UINT32	ulCANOPENSLV0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32	0	See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x29AA	`CANOPEN_SLAVE_SET_WATCHDOG_FAIL_REQ`

Table 90: `CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_REQ_T` – Set Watchdog Fail Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_Ttag</code> */
typedef struct CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_Ttag
  CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_T;

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

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

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		See section 6.2 *Codes of the CANopen Slave-Task*
ulCmd	UINT32	0x29AB	CANOPEN_SLAVE_SET_WATCHDOG_FAIL_CNF

Table 91: CANOPEN_SLAVE_PACKET_SET_WATCHDOG_FAIL_CNF_T – Set Watchdog Fail Confirmation

5.3 Other Packages

5.3.1 CANOPEN_APS_GET_STATE_REQ/CNF – Get State of AP task

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

Packet Structure Reference

```
typedef struct CANOPEN_APS_PCK_GET_STATE_REQ_Ttag
  CANOPEN_APS_PCK_GET_STATE_REQ_T;

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

Packet Description

Variable	Type	Value / Range	Description
ulDest	UINT32	0x20/ QUE_CANOPEN APS	Destination Queue-Handle of CANopen slave-Task Process Queue
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle of AP task Process Queue
ulDestId	UINT32	ulCANOPENSL V0Id	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	ulAPSS0Id	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulSta	UINT32		*See section 6.1.1 Codes of the CANopen-APS-Task*
ulCmd	UINT32	0x2E02	CANOPEN_APS_GET_STATE_REQ

Table 92: CANOPEN_APS_PCK_GET_STATE_REQ_T – Get State of AP task Request

Packet Structure Reference

```
/*****************************************************************************/
/** type of <code>CANOPEN_APS_GET_STATE_CNF_DATA_Ttag</code> */
typedef struct CANOPEN_APS_GET_STATE_CNF_DATA_Ttag
  CANOPEN_APS_GET_STATE_CNF_DATA_T;

struct CANOPEN_APS_GET_STATE_CNF_DATA_Ttag
{
  TLR_UINT32 aulUnused[2];
};

/*****************************************************************************/
/** type of <code>CANOPEN_APS_PCK_GET_STATE_CNF_Ttag</code> */
typedef struct CANOPEN_APS_PCK_GET_STATE_CNF_Ttag
  CANOPEN_APS_PCK_GET_STATE_CNF_T;

struct CANOPEN_APS_PCK_GET_STATE_CNF_Ttag /* Get state confirmation */
{
  TLR_PACKET_HEADER_T               tHead;   /** packet header */
  CANOPEN_APS_GET_STATE_CNF_DATA_T  tData;   /** packet data   */

};

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

Packet Description

Variable	Type	Value / Range	Description
ulDestId	UINT32	ulAPSS0Id	Destination End Point Identifier, untouched
ulSrcId	UINT32	ulCANOPENSLV0Id	Source End Point Identifier, untouched
ulLen	UINT32	8	Packet Data Length in bytes
ulSta	UINT32		See section *6.1.1 Codes of the CANopen-APS-Task*
ulCmd	UINT32	0x00002E03	CANOPEN_APS_GET_STATE_CNF
Data			
aulUnused[]	UINT32 [2]		Unused, present only due to compatibility reasons

Table 93: CANOPEN_APS_PCK_GET_STATE_CNF_T – Get State of AP task Confirmation

6 Status/Error Codes Overview

6.1.1 Codes of the CANopen-APS-Task

6.1.2 Error Messages

Definition / (Value)	Definition / Description
0x00000000	TLR_S_OK Status ok
0xC0000001	TLR_E_FAIL Common error, detailed error information optionally present in the data area of packet
0xC04A0002	TLR_E_CANOPEN_APS_DATABASE_FOUND Configuration database found.
0xC04A0003	TLR_E_CANOPEN_APS_NODE_ID_PARAMETER Invalid parameter for node id.
0xC04A0004	TLR_E_CANOPEN_APS_BAUDRATE_PARAMETER Invalid parameter for baudrate.
0xC04A0005	TLR_E_CANOPEN_APS_STATE Request not possible in current state.
0x404A0007	TLR_I_CANOPEN_APS_OPEN_DBM_FILE Failed to open configuration database.
0xC04A0008	TLR_E_CANOPEN_APS_DATASET Failed to open configuration dataset.
0xC04A0009	TLR_E_CANOPEN_APS_TABLE_GLOBAL Failed to open GLOBAL configuration dataset.
0xC04A000A	TLR_E_CANOPEN_APS_TABLE_BUS_CAN Failed to open BUS_CAN configuration dataset.
0xC04A000B	TLR_E_CANOPEN_APS_SIZE_TABLE_BUS_CAN Invalid size of BUS_CAN configuration dataset.
0x404A000C	TLR_I_CANOPEN_APS_CONFIG_LOCK Configuration is locked.
0xC04A000D	TLR_E_CANOPEN_APS_PACKET_LENGTH Invalid packet length.
0xC04A000E	TLR_E_CANOPEN_APS_WATCHDOG_PARAMETER Invalid parameter for watchdog supervision.
0xC04A000FL	TLR_E_CANOPEN_APS_WATCHDOG_ACTIVATE Failed to activate watchdog supervision.
0xC04A0010	TLR_E_CANOPEN_APS_PARAM_QUEUE_ELEMENT Invalid parameter for number of queue elements.
0xC04A0011	TLR_E_CANOPEN_APS_PARAM_POOL_ELEMENT Invalid parameter for number of pool elements.
0xC04A0012	TLR_E_CANOPEN_APS_PARAM_CYCLETIME Invalid parameter for cycle time.
0xC04A0013	TLR_E_CANOPEN_APS_PARAM_CHN_INSTANCE Invalid parameter for channel instance.
0xC04A0014	TLR_E_CANOPEN_APS_NUM_OF_RX_PDO_PARAMETER Invalid parameter for number of receive PDO.
0xC04A0015	TLR_E_CANOPEN_APS_NUM_OF_TX_PDO_PARAMETER Invalid parameter for number of send PDO.
0xC04A0016	TLR_E_CANOPEN_APS_SIZE_TABLE_VERSION Invalid size of table 'Version'.

Definition / (Value)	Definition / Description
0xC04A0017	TLR_E_CANOPEN_APS_INVALID_DBM_VERSION Invalid version of table 'Version'.
0xC04A0018	TLR_E_CANOPEN_APS_SIZE_TABLE_BUS_CAN_STD Invalid size of table 'BUS_COS_STD'.
0xC04A0019	TLR_E_CANOPEN_APS_SIZE_TABLE_BUS_CAN_EXT Invalid size of table 'BUS_COS_EXT'.
0xC04A001A	TLR_E_CANOPEN_APS_AUTOSTART_WITH_EXTENDED_MODE Auto start not allowed in extended mode.
0xC04A001B	TLR_E_CANOPEN_APS_ADDRESS_SWITCH_CONFIGURATION_NOT_POSSIBLE Address switch configuration is not possible.
0xC04A001C	TLR_E_CANOPEN_APS_BAUD_SWITCH_CONFIGURATION_NOT_POSSIBLE Baud switch configuration is not possible.
0xC04A001D	TLR_E_CANOPEN_APS_PARAM_LED_MODE Invalid parameter for LED mode.

Table 94: Error Messages of the AP task

6.2 Codes of the CANopen Slave-Task

6.2.1 Error Messages

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

Definition / (Value)	Description
0x00000000	TLR_S_OK Status ok
0xC0000001	TLR_E_FAIL Common error, detailed error information optionally present in the data area of packet
0xC0430003	TLR_E_CANOPEN_SLAVE_DATA_COUNT Invalid data count.
0xC0430004	TLR_E_CANOPEN_SLAVE_DATA_OFFSET Invalid data offset.
0xC0430005	TLR_E_CANOPEN_SLAVE_DATA_COUNT_WITH_OFFSET Invalid data count in combination with offset.
0xC0430006	TLR_E_CANOPEN_SLAVE_MODE Invalid mode in command.
0xC0430007	TLR_E_CANOPEN_SLAVE_STATE Command is not allowed in current state.
0xC0430009	TLR_E_CANOPEN_SLAVE_BUS_RUNNING Command is not allowed because CANopen is running.
0xC043000A	TLR_E_CANOPEN_SLAVE_BUS_PARAM_ALREADY_SET Bus parameters are already configured.
0xC043000B	TLR_E_CANOPEN_SLAVE_LOCAL_NODE_ID Invalid Node ID for CANopen slave.
0xC043000C	TLR_E_CANOPEN_SLAVE_BAUDRATE Invalid Baudrate.
0xC043000D	TLR_E_CANOPEN_SLAVE_29BIT_SELECTOR Invalid parameter for 29 bit selector.
0x4043000F	TLR_I_CANOPEN_SLAVE_ALREADY_IN_STATE Slave is already in requested state.
0xC0430010	TLR_E_CANOPEN_SLAVE_SEND_EMCY Send emergency-telegram failed.
0xC0430011	TLR_E_CANOPEN_SLAVE_INIT_LIB Failed to initialize CANopen library.
0xC0430012	TLR_E_CANOPEN_SLAVE_ERROR_PASSIVE CANopen is in error-passive state.
0xC0430013	TLR_E_CANOPEN_SLAVE_BUS_OFF CANopen is in bus-off state.
0xC0430014	TLR_E_CANOPEN_SLAVE_PUT_OBJECT_DATA Failed to write object data.
0xC0430015	TLR_E_CANOPEN_SLAVE_SET_OBJECT_DATA_VALID Failed to set object data valid.
0xC0430016	TLR_E_CANOPEN_SLAVE_GET_OBJECT_DATA Failed to get object data.
0xC0430017	TLR_E_CANOPEN_SLAVE_WRITE_PDO_REQ Failed to transmit PDO.
0xC0430018	TLR_E_CANOPEN_SLAVE_GUARD_ERROR Guard error detected.
0xC0430019	TLR_E_CANOPEN_SLAVE_INIT_BUFFER Initialization of buffer failed.

Definition / (Value)	Description
0xC043001A	TLR_E_CANOPEN_SLAVE_DL_REQ_FAILED CAN-DL request failed.
0xC043001B	TLR_E_CANOPEN_SLAVE_INVALID_INDEX Invalid object index.
0xC043001C	TLR_E_CANOPEN_SLAVE_INVALID_SUB_INDEX Invalid sub-index.
0xC043001D	TLR_E_CANOPEN_SLAVE_INVALID_MAP_LENGTH Invalid mapping length.
0xC043001E	TLR_E_CANOPEN_SLAVE_INVALID_PDO_MODE Invalid transmission mode for PDO.
0xC043001F	TLR_E_CANOPEN_SLAVE_INVALID_PDO_LENGTH Invalid length for PDO.
0xC0430020	TLR_E_CANOPEN_SLAVE_NO_WRITE_PERM No write permission for object.
0xC0430021	TLR_E_CANOPEN_SLAVE_NO_READ_PERM No read permission for object.
0xC0430022	TLR_E_CANOPEN_SLAVE_VALUE_TOO_LOW Value for object too low.
0xC0430023	TLR_E_CANOPEN_SLAVE_VALUE_TOO_HIGH Value for object too high.
0xC0430024	TLR_E_CANOPEN_SLAVE_INVALID_PARAMETER Invalid parameter for object.
0xC0430025	TLR_E_CANOPEN_SLAVE_INVALID_PDO_STATE Invalid PDO state.
0x40430026	TLR_I_CANOPEN_SLAVE_INITIALIZE Slave is initializing.
0xC0430027L	TLR_E_CANOPEN_SLAVE_OBJECT_SIZE Invalid size for object.
0xC0430028	TLR_E_CANOPEN_SLAVE_ID_IN_USE Identifier already in use.
0xC0430029	TLR_E_CANOPEN_SLAVE_INHIBIT Service is inhibited.
0xC043002A	TLR_E_CANOPEN_SLAVE_TX_OVERRUN Transmit overrun.
0xC043002B	TLR_E_CANOPEN_SLAVE_RX_OVERRUN Receive overrun.
0xC043002C	TLR_E_CANOPEN_SLAVE_ERROR_WARNING CANopen is in error-warning state.
0xC043002D	TLR_E_CANOPEN_SLAVE_RECV_PDO_REQ Request receive PDO failed.
0xC043002E	TLR_E_CANOPEN_SLAVE_NUM_OF_RX_PDO_PARAMETER Invalid parameter for number of receive PDO.
0xC043002F	TLR_E_CANOPEN_SLAVE_NUM_OF_TX_PDO_PARAMETER Invalid parameter for number of send PDO.
0xC0430030	TLR_E_CANOPEN_SLAVE_HB_CONSUMER_PARAMETER Invalid parameter for number of heartbeat consumer.
0xC0430031	TLR_E_CANOPEN_SLAVE_SEND_TIME_STAMP Failed to send timestamp message.

Table 95: Error Messages of the CANopen Slave-Task

6.3 Codes of CAN-DL Task

See separate CAN-DL Task documentation, reference [6].

6.4 Codes of ODV3

See separate Object Dictionary V3 documentation, reference [5].

6.5 Emergency Telegram

The Emergency Telegram transfers the following information

1. Byte 1 and 2: Emergency Error Code
2. Byte 3: Error Register
3. Byte 4 to 8: Manufacturer-specific Error Codes

6.5.1 Emergency Error Codes

The emergency error codes are specified in [2] and have the following meaning:

Error Code	Meaning
0x0000	Error reset or no error
0x1000	Generic error
0x2000	Current error
0x2100	Current error, device input side
0x2200	Current error, inside the device
0x2300	Current error, device output side
0x3000	Voltage error
0x3100	Main voltage error
0x3200	Voltage error inside the device
0x3300	Output voltage error
0x4000	Temperature error
0x4100	Ambient temperature error
0x4200	Device temperature error
0x5000	Device hardware error
0x6000	Device software error
0x6100	Internal software error
0x6200	User software error
0x6300	Data set error
0x7000	Additional modules error
0x8000	Monitoring error
0x8100	Communication error
0x8110	CAN overrun (objects lost) error
0x8120	CAN in Error Passive Mode error
0x8130	Life Guard Error or Heartbeat error
0x8140	Recovered from bus-off
0x8150	Transmit CAN-ID collision error
0x8200	Protocol error
0x8210	PDO not processed due to length error
0x8220	PDO length exceeded
0x8230	DAM MPDO not processed, destination object not available
0x8240	Unexpected SYNC data length error
0x8250	RPDO timeout error
0x9000	External error
0xF000	Additional functions error
0xFF00	Device-specific error

Table 96: Emergency Error Codes (Variable `usErrorCode`)

6.5.2 Error Register

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

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

Table 97: Error Register

6.5.3 Manufacturer-specific Error Codes

The CANopen Slave stack generated the manufacturer-specific error codes listed in the following tables.

1. and 2. byte (Uint16)	3. byte	4. byte	5. byte
Index of element which causes the EMCY telegram. 0 = all elements 1, 2, 3, ... = this element	Event, see table below.	0	0

Table 98: Manufacturer-specific Error Codes generated by the CANopen Slave stack (Structure)

The following table lists the events of manufacturer-specific emergency errors:

Event	Definition
0x00	CANOPEN_SLAVE_EMCY_PROD_EVENT_INITIALIZED
0x01	CANOPEN_SLAVE_EMCY_PROD_EVENT_BUS_OFF_RECOVER
0x02	CANOPEN_SLAVE_EMCY_PROD_EVENT_ERROR_PASSIVE
0x03	CANOPEN_SLAVE_EMCY_PROD_EVENT_RXTX_OVERRUN
0x04	CANOPEN_SLAVE_EMCY_PROD_EVENT_NMT_PROTOCOL_ERROR
0x05	CANOPEN_SLAVE_EMCY_PROD_EVENT_SYNC_PROTOCOL_ERROR
0x06	CANOPEN_SLAVE_EMCY_PROD_EVENT_SYNC_DATA_LENGTH_ERROR
0x07	CANOPEN_SLAVE_EMCY_PROD_EVENT_TIME_STAMP_PROTOCOL_ERROR
0x08	CANOPEN_SLAVE_EMCY_PROD_EVENT_WATCHDOG_ERROR
0x09	CANOPEN_SLAVE_EMCY_PROD_EVENT_STOP_COMMUNICATION
0x0A	CANOPEN_SLAVE_EMCY_PROD_EVENT_NODE_GUARD_ERROR
0x0B	CANOPEN_SLAVE_EMCY_PROD_EVENT_HEARTBEAT_ERROR
0x0C	CANOPEN_SLAVE_EMCY_PROD_EVENT_SDO_SRV_PROTOCOL_ERROR
0x0D	CANOPEN_SLAVE_EMCY_PROD_EVENT_PDO_LENGTH_ERROR
0x0E	CANOPEN_SLAVE_EMCY_PROD_EVENT_PDO_LENGTH_EXCEEDED

Table 99: Manufacturer-specific Error Codes generated by the CANopen Slave stack (Event information)

Examples

The CANopen Slave receives a telegram for the 3. PDO which is to short:
10 82 01 03 00 0D 00 00

The CANopen Slave then receives a telegram for the 3. PDO which has the expected length:
00 00 00 03 00 0D 00 00

7 Appendix

7.1 List of Tables

7.2 List of Figures

7.3 Legal Notes

Copyright

Important notes

Utmost care was/is given in the preparation of the documentation at hand consisting of a user's manual, operating manual and any other document type and accompanying texts. However, errors cannot be ruled out. Therefore, we cannot assume any guarantee or legal responsibility for erroneous information or liability of any kind. You are hereby made aware that descriptions found in the user's manual, the accompanying texts and the documentation neither represent a guarantee nor any indication on proper use as stipulated in the agreement or a promised attribute. It cannot be ruled out that the user's manual, the accompanying texts and the documentation do not completely match the described attributes, standards or any other data for the delivered product. A warranty or guarantee with respect to the correctness or accuracy of the information is not assumed.

We reserve the right to modify our products and the specifications for such as well as the corresponding documentation in the form of a user's manual, operating manual and/or any other document types and accompanying texts at any time and without notice without being required to notify of said modification. Changes shall be taken into account in future manuals and do not represent an obligation of any kind, in particular there shall be no right to have delivered documents revised. The manual delivered with the product shall apply.

Under no circumstances shall Hilscher Gesellschaft für Systemautomation mbH be liable for direct, indirect, ancillary or subsequent damage, or for any loss of income, which may arise after use of the information contained herein.

Liability disclaimer

The hardware and/or software was created and tested by Hilscher Gesellschaft für Systemautomation mbH with utmost care and is made available as is. No warranty can be assumed for the performance or flawlessness of the hardware and/or software under all application conditions and scenarios and the work results achieved by the user when using the hardware and/or software. Liability for any damage that may have occurred as a result of using the hardware and/or software or the corresponding documents shall be limited to an event involving willful intent or a grossly negligent violation of a fundamental contractual obligation. However, the right to assert damages due to a violation of a fundamental contractual obligation shall be limited to contract-typical foreseeable damage.

It is hereby expressly agreed upon in particular that any use or utilization of the hardware and/or software in connection with

- Flight control systems in aviation and aerospace;
- Nuclear fission processes in nuclear power plants;
- Medical devices used for life support and
- Vehicle control systems used in passenger transport

shall be excluded. Use of the hardware and/or software in any of the following areas is strictly prohibited:

- For military purposes or in weaponry;
- For designing, engineering, maintaining or operating nuclear systems;
- In flight safety systems, aviation and flight telecommunications systems;
- In life-support systems;
- In systems in which any malfunction in the hardware and/or software may result in physical injuries or fatalities.

You are hereby made aware that the hardware and/or software was not created for use in hazardous environments, which require fail-safe control mechanisms. Use of the hardware and/or software in this kind of environment shall be at your own risk; any liability for damage or loss due to impermissible use shall be excluded.

Warranty

Hilscher Gesellschaft für Systemautomation mbH hereby guarantees that the software shall run without errors in accordance with the requirements listed in the specifications and that there were no defects on the date of acceptance. The warranty period shall be 12 months commencing as of the date of acceptance or purchase (with express declaration or implied, by customer's conclusive behavior, e.g. putting into operation permanently).

The warranty obligation for equipment (hardware) we produce is 36 months, calculated as of the date of delivery ex works. The aforementioned provisions shall not apply if longer warranty periods are mandatory by law pursuant to Section 438 (1.2) BGB, Section 479 (1) BGB and Section 634a (1) BGB [Bürgerliches Gesetzbuch; German Civil Code] If, despite of all due care taken, the delivered product should have a defect, which already existed at the time of the transfer of risk, it shall be at our discretion to either repair the product or to deliver a replacement product, subject to timely notification of defect.

The warranty obligation shall not apply if the notification of defect is not asserted promptly, if the purchaser or third party has tampered with the products, if the defect is the result of natural wear, was caused by unfavorable operating conditions or is due to violations against our operating regulations or against rules of good electrical engineering practice, or if our request to return the defective object is not promptly complied with.

Costs of support, maintenance, customization and product care

Please be advised that any subsequent improvement shall only be free of charge if a defect is found. Any form of technical support, maintenance and customization is not a warranty service, but instead shall be charged extra.

Additional guarantees

Although the hardware and software was developed and tested in-depth with greatest care, Hilscher Gesellschaft für Systemautomation mbH shall not assume any guarantee for the suitability thereof for any purpose that was not confirmed in writing. No guarantee can be granted whereby the hardware and software satisfies your requirements, or the use of the hardware and/or software is uninterruptable or the hardware and/or software is fault-free.

It cannot be guaranteed that patents and/or ownership privileges have not been infringed upon or violated or that the products are free from third-party influence. No additional guarantees or promises shall be made as to whether the product is market current, free from deficiency in title, or can be integrated or is usable for specific purposes, unless such guarantees or promises are required under existing law and cannot be restricted.

Confidentiality

The customer hereby expressly acknowledges that this document contains trade secrets, information protected by copyright and other patent and ownership privileges as well as any related rights of Hilscher Gesellschaft für Systemautomation mbH. The customer agrees to treat as confidential all of the information made available to customer by Hilscher Gesellschaft für Systemautomation mbH and rights, which were disclosed by Hilscher Gesellschaft für Systemautomation mbH and that were made accessible as well as the terms and conditions of this agreement itself.

The parties hereby agree to one another that the information that each party receives from the other party respectively is and shall remain the intellectual property of said other party, unless provided for otherwise in a contractual agreement.

The customer must not allow any third party to become knowledgeable of this expertise and shall only provide knowledge thereof to authorized users as appropriate and necessary. Companies associated with the customer shall not be deemed third parties. The customer must obligate authorized users to confidentiality. The customer should only use the confidential information in connection with the performances specified in this agreement.

The customer must not use this confidential information to his own advantage or for his own purposes or rather to the advantage or for the purpose of a third party, nor must it be used for commercial purposes and this confidential information must only be used to the extent provided for in this agreement or otherwise to the extent as expressly authorized by the disclosing party in written form. The customer has the right, subject to the obligation to confidentiality, to disclose the terms and conditions of this agreement directly to his legal and financial consultants as would be required for the customer's normal business operation.

Export provisions

The delivered product (including technical data) is subject to the legal export and/or import laws as well as any associated regulations of various countries, especially such laws applicable in Germany and in the United States. The products / hardware / software must not be exported into such countries for which export is prohibited under US American export control laws and its supplementary provisions. You hereby agree to strictly follow the regulations and to yourself be responsible for observing them. You are hereby made aware that you may be required to obtain governmental approval to export, reexport or import the product.

7.4 Registered Trademarks

CANopen® is a registered trademark of CAN in AUTOMATION - International Users and Manufacturers Group e.V. (CiA), Erlangen.

All other mentioned trademarks are property of their respective legal owners.

7.5 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.
69800 Saint Priest
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