



CANopen Manager

Software Manual



NOTE

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd gmbh**.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd gmbh** nor the rights of others.

esd electronic system design gmbh

Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd-electronics.com
Internet: www.esd-electronics.com

USA / Canada:

esd electronics Inc.

525 Bernardston Road
Suite 1
Greenfield, MA 01301
USA

Phone: +1-800-732-8006
Fax: +1-800-732-8093
E-mail: us-sales@esd-electronics.com
Internet: www.esd-electronics.us

Manual File:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\CAL-COPN\CANOPEN\Comst221.ma9
Date of Print:	2008-06-05

Described Software:	CANopen Manager
Revision:	2.2.x

Order no.:	CANopen manager: P.1405.80
-------------------	----------------------------

Changes in this manual

With software revision 2.x the CANopen manager is completely rewritten to comply to CiA DS-302. The API is the same as for the CAL-/CANopen master for backward compatibility but the syntax of the initialization file has changed from a proprietary ASCII format to XML.

The changes in this version of the user's manual listed below affect changes in the software, as well as changes in the description of the facts only compared to the previous revision.

Alterations in the appendix versus previous revisions	Alterations in software	Alterations in documentation
Revised introduction and reference		x
New configuration file keywords NMT_Reset_All, NMT_Reset_Node, Sync_Interval, Sync_Cob_Id, Debug_Mask, Manager_Thread_Priorities, Slave_Thread_Priorities and Native_Thread_Priorities.	x	x
Default of keyword NMT_AUTOSTART_SLAVE is TRUE and not FALSE.		x
New chapter to document the additional command line tools which come with a CANopen release.		x

Content	Page
1. Reference	5
2. Overview	6
2.1 Notes to this Manual	6
2.2 Introduction	6
2.3 Features of the CANopen manager library	7
2.4 Receiving process data	7
3. Initializing the CANopen Manager	8
3.1 Starting and Terminating the CANopen Manager	8
3.2 Configuration of the CANopen Manager	8
3.2.1 Configuration file element relationship	8
3.2.2 Element description	9
3.2.3 Elements of <Manager_Instance>	10
3.2.4 Elements of <Node_Instance>	15
3.2.5 Example configuration file	20
4. Manager API	21
4.1 Overview	21
4.2 Basic manager initialization functions	21
4.3 Optional node control functions	25
4.4 Extended Functions	28
5. Additional Tools	34
5.1 dcftest	34
5.2 dcf2ccf	34
5.3 ccftread	34
6. Appendix	35
6.1 Error Codes	35

1. Reference

- /1/: CiA DS-301 V 4.1, CANopen-Application layer and communication profile, February 2006
- /2/: CiA DSP-302 V4.0, Framework for CANopen Managers and Programmable Devices, August 2006
- /3/: CiA DS-306 V1.3.0, Electronic Data Sheet Specification for CANopen, January 2005
- /4/: ESD NTCAN-API Manual, V2.8, July 2007

2. Overview

The CANopen manager library allows an easy integration of the complex CANopen manager functionality in an application with the need for a sophisticated CANopen network and process control.

The library implements a complete CANopen manager according to CiA DS-301 and DS-302 with support for the standardized network bootup, error control via cyclic node guarding or heartbeat monitoring, EMCY object monitoring, SYNC generation, error handling and support for SDO services.

Some highlights of the library are:

- ▶ Comprehensive set of services based on the CANopen specifications CiA DS-301 and CiA DS-302 to easily integrate CANopen manager functionality into an application.
- ▶ Support for several (real-time) operating systems and CAN adapter available with the same OS and hardware independent proven CANopen manager core.
- ▶ Comes as fully multi-threaded shared or static library. All CANopen manager related tasks like the check of the network integrity, control of the parallel system boot-up, automatic download of device configuration files and the detection of CANopen slave device failures are handled in background.
- ▶ Supports CANopen slave devices which comply to DS-301 V 3.x and 4.x.
- ▶ Supports control of multiple CAN networks.
- ▶ Can be configured by the application with an XML based configuration file or with the help of additional tools via CAN and the standardized object dictionary.
- ▶ For device configuration DCF files which comply to DS-306 as well as CCF files which comply to DS-302 are supported.

2.1 Notes to this Manual

This manual describes the CANopen manager software package by esd. The CANopen manager is available for various (real-time) operating systems and CAN adapter.

The first chapters of this manual describe the general functionality of the CANopen manager. The operating system specific features of the manager are listed in the appendix.

Knowledge of CANopen definitions and protocol descriptions is required.. References to CiA publications are accordingly cross-referenced (/1/.../3/).

2.2 Introduction

The CANopen protocol started as CAL based protocol using its communication and network management services and defined application and device specific objects which are stored in an object dictionary. Nowadays the CANopen protocol is improved without any reference to CAL.

Each remote node in a CANopen network is assigned a module identifier (1-127) which has to be unique in the network. CANopen nodes are configured by the configuration manager part of the CANopen manager via access to the object dictionary based on Device Configuration Files (DCF). The CANopen nodes are managed and monitored by the NMT master via CAN after the boot-up.

2.3 Features of the CANopen manager library

The CANopen manager library provides all features to configure, start and monitor CANopen slaves. The library can be used easily as stand-alone application to setup a CANopen network based on a manager configuration file in XML format and DCF files for the CANopen nodes but provides also a Application Programmer Interface (API) to control remote slaves by a custom application. All functionality specified by CANopen for a master to configure and manage remote nodes is covered by the library:

- Detection of CANopen nodes in the CAN network.
- Configuration of CANopen nodes with or without DCF files (Configuration manager).
- Management and monitoring of remote node states (NMT master).
- Monitoring of CANopen Emergency Objects.
- SYNC object generation (Hardware dependent).

Communication between the library and the application is realized with an event mechanism. In addition the libraries API gives the application the possibility to:

- Control the node state of remote slaves.
- Monitor the node state of remote slaves.
- Read and write object dictionary entries of remote slaves.

2.4 Receiving process data

The CANopen manager library doesn't support receiving process data in PDOs directly. If you want to build an application which is intended to manage a CANopen network and process the data of the CANopen slave devices in this network you have to create a further CANopen slave with the CANopen slave library, assign it another node-ID than the one which is assigned to the CANopen manager and can use the slave API to process the data.

3. Initializing the CANopen Manager

3.1 Starting and Terminating the CANopen Manager

The CANopen manager operates in the background and if it is running faultlessly it does not require further intervention by the user after its start

A configuration file has to be created to describe the CANopen network which represents the project specific CANopen net.

3.2 Configuration of the CANopen Manager

The manager configuration file and the CANopen DCF files can be created with a standard text editor for ASCII files. The syntax of the CANopen DCF files is described in /3/. The manager configuration file follows the XML syntax. The keywords are described in following abstract.

3.2.1 Configuration file element relationship

Each XML configuration file for the CANopen manager has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<CANopen_Manager>
  <Manager_Instance>
    .
    .
    <Node_Instance>
      .
      .
    </Node_Instance>
    <Node_Instance>
      .
      .
    </Node_Instance>
  </Manager_Instance>
</CANopen_Manager>
```

CANopen_Manager is the root element of the configuration file. It contains child elements of Manager_Instance for every physical CAN network that should be controlled by this manager. Each of these Manager_Instance child elements contains child elements of Node_Instance for every instance of a remote slave that should be controlled in this network. Each of the elements mentioned above contain only element content. These elements are described in the following abstracts.

3.2.2 Element description

The description of all XML elements that are supported in the configuration file are made in a table with the following structure:

Keyword	Category	Data Type	Dictionary Entry
Value Range	Default	Obsolete syntax	
Description			

- **Keyword** is a XML element name.
- **Category** is either Mandatory or Optional.
- **Data Type** is one of the following:
 - ▶ UCHAR - 0..0xFF
 - ▶ USHORT - 0..0xFFFF
 - ▶ ULONG - 0..0xFFFFFFFF
 - ▶ INT16 - -32768..32767
 - ▶ BOOL - TRUE/FALSE
 - ▶ ASCII - ASCII string
 - ▶ ENUM - Element specific enumeration
- **Dictionary Entry** describes index, subindex and bit position for a keyword that is directly related to a DS-302 dictionary entry. If there is no relation this box is empty.
- **Value Range** is the valid range for this keyword. If a range isn't applicable this box is empty.
- **Default** is the value that is used if an optional keyword is missing.
- **Obsolete Syntax** is the keyword representation in the old style configuration syntax. If there is no such relation this box is empty. You can use this as a reference to port previous configuration files.
- **Description** is the purpose or effect of this keyword.

3.2.3 Elements of <Manager_Instance>

Net_No	Mandatory	UCHAR	
0 - 255		Net	
The element defines the logical net number the device driver has assigned to the CAN port that should be used by this instance of the manager.			

Baudrate	Mandatory	USHORT	
Valid CANopen baudrate		Baud	
The baudrate used for this CAN network in Kbit/s.			

Manager_Node_Id	Optional	UCHAR	
1 - 127	127		
According to DS-302 the manager itself is an active instance of a CANopen slave to offer configuration mechanisms via an object dictionary. For this purpose a Node-ID has to be assigned to the manager. This Node-ID must not conflict with the Node-ID of any remote CANopen slave in the network that is controlled by this manager.			

Config_File_Repository	Optional	ASCII	
OS dependent valid path		DCF_PATH	
Additional search path for the CANopen node's configuration files (DCF file or CCF file) to ease the logical organization of configuration files in the host file system. If opening a given configuration file in the working directory fails, this string is concatenated with the operating system specific path separator is prepended to the filename to open the configuration file from here.			

Heartbeat_Producer_Time	Optional	USHORT	0x1017
100 - 65535	0		
This parameter defines the cycle time of the manager heartbeat. The producer heartbeat time is 0 if not defined.			

Boot_Time	Optional	USHORT	0x1F89
100 - 65535	0		
Maximum time in ms the NMT master will wait for all mandatory slaves before signalling an error. If the time is 0 the master will wait endlessly. Attempt to boot optional slaves will always be repeated endlessly.			

NMT_Master	Optional	BOOL	0x1F80, Bit 0
TRUE, FALSE	TRUE		
If set to TRUE this is the NMT master. If set to FALSE this isn't the NMT master, the application is started in the slave only mode and no attempt is made to configure and start the CANopen network.			

NMT_Start_All	Optional	BOOL	0x1F80, Bit 1
TRUE, FALSE	FALSE	AUTOSTART	
This parameter defines the way the NMT master will start the remote slaves If set to TRUE the master will perform the service 'NMT Start Remote Nodes All Nodes'. If set to FALSE the master will start each remote node individually.			

NMT_Auto_Operational	Optional	BOOL	0x1F80, Bit 2
TRUE, FALSE	TRUE	AUTOSTART	
This parameter defines the behavior of the NMT master after all mandatory slaves are booted successfully and all optional slaves are tried to be booted at least once. If set to TRUE the master will automatically enter the node state 'Operational' and continue to start the remote slaves. If set to FALSE the NMT master will wait to start the remote slaves until the application decides the master should enter the state 'Operational'.			

NMT_Auto_Start	Optional	BOOL	0x1F80, Bit 3
TRUE, FALSE	TRUE	AUTOSTART	
This parameter defines the behavior of the NMT master after it has entered the state 'Operational'. If set to TRUE the master is allowed to start the remote slaves. If set to FALSE the application will start the remote slaves.			

NMT_Action_On_Error	Optional	ENUM	0x1F80, Bit 4/6
	RESET_SLAVE		
<p>This parameter defines the way the NMT master handles error control events.</p> <p>If this parameter is set to RESET_SLAVE on Error Control Event of a mandatory the slave is treated individually.</p> <p>If this parameter is set to RESET_ALL on Error Control Event of a mandatory slave the NMT master performs a 'NMT Reset all Nodes' command (including self).</p> <p>If this parameter is set to STOP_ALL on Error Control Event of a mandatory slave the NMT master performs a 'Stop all Nodes' command (including self).</p>			

Boot_Delay	Optional	USHORT	
0 - 65535	200		
<p>This parameter defines the delay in ms the manager waits after the initial 'NMT Reset Communication' before continuing with the 'Start Boot Slave Process'. You might adjust this parameter to be sure that all slaves finished their initialisation. See also the node specific parameter Boot_Delay in the next chapter.</p>			

Wait_For_Startup	Optional	BOOL	
TRUE, FALSE	FALSE		
<p>This parameter defines the behaviour of mcanOpenStart() that initiates the 'Start Boot Slave' process, which is performed in parallel for each managed slave node.</p> <p>If this parameter is set to FALSE the call will return immediately after the parallel processes are started.</p> <p>If this parameter is set to TRUE the call won't return before every process hasn't created at least the signal for the Boot Slave try.</p>			

NMT_Reset_All	Optional	BOOL	
TRUE, FALSE	FALSE		
<p>This parameter defines, if the initial NMT command to reset the CANopen network at the beginning of the bootup procedure is the NMT Reset Communication (as defined as only mechanism in the DS-302) or the NMT Reset Node. This parameter is only evaluated if no node is configured with NMT_Check_Nodestate set to TRUE. This possible change from the standard behaviour should only be enabled if a bootup with NMT Reset Communication causes problems with any of the CANopen slaves connected to this network.</p>			

Preset_Emergency_Consumer	Optional	BOOL	
TRUE, FALSE	FALSE		
<p>This parameter defines the behaviour of the CANopen manager as emergency consumer. If this parameter is set to FALSE only emergency messages of remote slaves that are defined in this configuration file are received. If this parameter is set to TRUE the manager is configured to receive emergency messages of all remote nodes according to the default connection set even if they are not defined in this configuration file.</p>			

Sync_Interval	Optional	ULONG	0x1005, 0x1006
N/A	N/A		
<p>This parameter configures the CANopen manager as SYNC producer. If this entry is present, the object dictionary entries COB-ID SYNC message (0x1005) and Communication Cycle Period (0x1006) are created. The parameter defines the SYNC interval in us. If set to 0 the object dictionary entries are created but the SYNC object isn't generated. If set to another value SYNC generation starts immediately with the configured value.</p> <p>Note: To archive minimal jitter the SYNC object is generated with the help of the scheduling mode /4/ of the NTCAN-API. For this reason this service is not available on all CAN hardware and/or operating systems which are supported by the CANopen stack.</p>			

Sync_Cob_Id	Optional	ULONG	0x1005
N/A	0x80		
<p>COB-ID of the SYNC object. To configure SYNC generation you also have to define the <code>Sync_Interval</code> described above.</p>			

Manager_Thread_Priorities	Optional	INT16	
N/A	N/A		
<p>Priorities of the manager threads as three INT16 values separated by colons. The 1st priority value is used for the manager main control daemon, the 2nd priority value for the NMT error control daemon and the 3rd priority for the slave node bootup daemons. The priority is either a PSYS priority or a OS specific native priority if <code>Native_Thread_Priorities</code> described below is set to TRUE.</p>			

Slave_Thread_Priorities	Optional	INT16	
N/A	N/A		
<p>Priorities of the slave threads as four INT16 values separated by colons. The 1st priority value is used for the slave NMT thread, the 2nd priority value for the slave PDO thread, the 3rd priority for the slave SDO thread and the 4th priority for the slave EMCY consumer thread. The priority is either a PSYS priority or a OS specific native priority if <code>Native_Thread_Priorities</code> described below is set to TRUE.</p>			

Native_Thread_Priorities	Optional	BOOL	
TRUE, FALSE	FALSE		
<p>If TRUE the priorities of <code>Mananger_Thread_Priorities</code> and <code>Slave_Thread_Priorities</code> are native OS specific priorities instead of PSYS priorities.</p>			

Debug_Mask	Optional	ULONG	
N/A	N/A		
<p>For special debug builds of the mananger and slave CANopen stack it's possible to define a mask to control the amount of debug trace messages. The LSW of this value is the debug mask for the mananger and the MSW for the slave. This parameter is ignored by a release build of the library.</p>			

3.2.4 Elements of <Node_Instance>

Node_Id	Mandatory	UCHAR	
1-127		NODEID	
Node-ID of this instance of the remote node that should be controlled by the master. The same Node-ID must not be assigned twice within one Manager_Instance. The Node-ID of the manager itself has to be unique, too.			

NMT_Slave	Optional	BOOL	0x1F81, Sub Node_Id, Bit 0
TRUE, FALSE	TRUE		
If set to TRUE the node with this Node-ID is a slave. After configuration (with Configuration Manager) the Node will be set to the state Operational by the NMT master. If set to FALSE the node with this Node-ID is not started by the NMT master			

NMT_Auto_Start_Slave	Optional	BOOL	0x1F81, Sub Node_Id, Bit 2
TRUE, FALSE	TRUE		
If set to FALSE on Error Control Event or detection of a booting slave the application is informed but the slave isn't automatically configure and started. If set to TRUE on Error Control Event or detection of a booting slave inform the application and do start the process "Start Boot Slave".			

NMT_Mandatory_Slave	Optional	BOOL	0x1F81, Sub Node_Id, Bit 3
TRUE, FALSE	FALSE		
If set to TRUE the node with this Node-ID is a mandatory slave and the network must not be started if this slave node could not be contacted during the boot slave procedure. If set to FALSE the node with this Node-ID is an optional slave network may be started even if this node could not be contacted			

Guard_Time	Optional	USHORT	0x1F81, Sub Node_Id, Byte 2-3
0-65535	0	GUARD	
This parameter determines the interval in ms a guard RTR is send. If the answer is missing (Retry_Factor-1) the master will create a guard error event. Guarding will be performed only if Guard_Time and Retry_Factor are both different from 0. The guard time in ms configured for the NMT master should be smaller than the guard time configured for the remote slave by the configuration manager to prevent frequent timeout errors.			

Life_Time_Factor	Optional	UCHAR	0x1F81, Sub Node_Id, Byte1
0-255	0	LIFE	
<p>This parameter determines the factor the guard RTR is resend before the NMT master will create a guard error event. Guarding will be performed only if <code>Guard_Time</code> and <code>Retry_Factor</code> are both different from 0.</p>			

NMT_Check_Nodestate	Optional	BOOL	0x1F81, Sub Node_Id, Bit 4
TRUE, FALSE	FALSE		
<p>If set to FALSE the slave node may be reset with 'NMT Reset Communication' command independent of its state. Hence, no checking of its state has to be executed prior to 'NMT Reset Communication' command. If set to TRUE the NMT Master must not send 'NMT Reset Communication' for this node if it notices the slave to be in 'Operational' state. This is controlled either waiting for the heartbeat message or waiting for the reply of a RTR requested node guard message.</p>			

NMT_Verify_Software	Optional	BOOL	0x1F81, Sub Node_Id, Bit 5
TRUE, FALSE	FALSE		
<p>If set to FALSE the application software version verification for this node is not required. If set to TRUE application software version verification for this node is required.</p>			

NMT_Allow_Software_Update	Optional	BOOL	0x1F81, Sub Node_Id, Bit 6
TRUE, FALSE	FALSE		
<p>If set to FALSE the automatic application software update (download) is not allowed. If set to TRUE the automatic application software update (download) is allowed.</p>			

Consumer_Heartbeat_Time	Optional	USHORT	0x1016, Sub Node_Id, Byte 0-1
65535	0		
<p>Defines the consumer heartbeat time in ms. If within this interval no heartbeat message is received from the remote slave the NMT master will create a error monitoring event. The consumer heartbeat time in ms configured for the NMT master should be configured to a higher value than the producer heartbeat time configured for the remote slave by the configuration manager to prevent frequent error monitoring events. If heartbeat monitoring is configured for this remote node all configuration for guarding is ignored.</p>			

Device_Type_Identification	Optional	ULONG	0x1F84, Sub Node_Id
0x0..0xFFFFFFFF	0		
<p>This element allows to enter values for expected device types. On Boot-Up the Master reads object 0x1000 of each assigned slave. If this value is 0, this read access only gives information about the principle existence of a device with this Node-ID. If the value is not 0, it is compared against the value read from the device and the boot-up for that device is only continued on exact equality.</p>			

Vendor_Identification	Optional	ULONG	0x1F85, Sub Node_Id
0x0..0xFFFFFFFF	0		
<p>This element allows to enter values for expected vendor id. If <code>Vendor_Identification</code> is not 0 the master will read object 0x1018, subindex 1 of the remote node and will compare this value against the value read from the device. The boot-up for that device is only continued on exact equality.</p>			

Product_Code	Optional	ULONG	0x1F86, Sub Node_Id
0x0..0xFFFFFFFF	0		
<p>This element allows to enter values for expected vendor id. If <code>Product_Code</code> is not 0 the master will read object 0x1018, subindex 2 of the remote node and will compare this value against the value read from the device. The boot-up for that device is only continued on exact equality.</p>			

Revision_Number	Optional	ULONG	0x1F87, Sub Node_Id
0x0..0xFFFFFFFF	0		
<p>This element allows to enter values for expected vendor id. If <code>Revision_Number</code> is not 0 the master will read object 0x1018, subindex 3 of the remote node and will compare this value against the value read from the device. The boot-up for that device is only continued on exact equality.</p>			

Serial_Number	Optional	ULONG	0x1F88, Sub Node_Id
0x0..0xFFFFFFFF	0		
<p>This element allows to enter values for expected vendor id. If <code>Serial_Number</code> is not 0 the master will read object 0x1018, subindex 4 of the remote node and will compare this value against the value read from the device. The boot-up for that device is only continued on exact equality.</p>			

DCF_File	Optional	ASCII	
		DCF	
<p>DCF file according to DS-306 that is used by the Configuration Manager to configure the remote node. This file is translated internally into the 'Concise Configuration Format' (CCF) defined in DS-302. For one Node_Instance either a DCF file or a CCF file can be defined. If no DCF file or CCF file is assigned to the node the Configuration Manager can't configure the remote slave. Error control is only possible if guarding is configured and the remote slave supports passive guarding or if the remote slave has a pre-configured heartbeat time that matches the consumer heartbeat time configured for the NMT master.</p>			

CCF_File	Optional	ASCII	
<p>CCF file according to DS-302 that is used by the Configuration Manager to configure the remote node. A CCF file is already in a (binary) format that describes the data stream the Configuration Manager sends to the remote node for configuration. CCF files have to be derived from DCF files in an offline configuration step. For one Node_Instance either a DCF file or a CCF file can be defined. If no DCF file or CCF file is assigned to the node the Configuration Manager can't configure the remote slave. Error control is only possible if guarding is configured and the remote slave supports passive guarding or if the remote slave has a pre-configured heartbeat time that matches the consumer heartbeat time configured for the NMT master.</p>			

Old_Config_Style	Optional	BOOL	
TRUE, FALSE	FALSE		
<p>In DS-301 Rev. 4.x the configuration process for PDOs was described in detail the first time. To configure the PDO mapping the Configuration Manager has to set the number of mapping entries to 0 before changing the PDO mapping. Some older CANopen devices may have problems with this behaviour as they expect the number of mapping entries is configured to the correct value before PDO Mapping entries are changed.</p> <p>If this parameter is set to FALSE the the number of mapping entries is set to 0 before the mapping entries itself are changed. The number of mapping entries is set to the correct value afterwards to validate the configuration.</p> <p>If this parameter is set to TRUE the correct value is written before the mapping entries itself are changed.</p> <p>This entry only has an effect to DCF files. As CCF files are already in a format that describes the data stream the Configuration Manager sends, changing configuration style has to be indicated during offline conversion from DCF files into CCF files.</p>			

NMT_Reset_Node	Optional	BOOL	
TRUE, FALSE	FALSE		
<p>This parameter defines, if the initial NMT command to reset this node at the beginning of the bootup procedure is the NMT Reset Communication (as defined as only mechanism in the DS-302) or the NMT Reset Node. This parameter is only evaluated if the node isn't configured with NMT_Check_Nodestate set to TRUE. This possible change from the standard behaviour should only be enabled if a bootup with NMT Reset Communication causes problems with any of the CANopen slaves connected to this network.</p>			

Boot_Delay	Optional	USHORT	
0 - 65535	200		
<p>This parameter defines the delay in ms the manager waits after a 'NMT Reset Communication' or 'NMT Reset Application' for this node before continuing with the 'Start Boot Slave Process'. It is applied if the manager tries to restart the node as result of an error monitoring event, if the application forces the reset or the node is configured with the 'Keeping alive' option (see parameter NMT_Check_Nodestate). In comparison to the network specific parameter Boot_Delay this time is part of the Boot_Time configured for this network.</p>			

SDO_Timeout	Optional	USHORT	
0 - 65535	200	SDO_RX_TIMEOUT	
<p>If the object dictionary of the remote slave is read or written by the CANopen manager during bootup and configuration the operation fails after a timeout without an reply of the slave. This parameter defines the timeout value in ms.</p>			

SDO_Segmented_Size_Indication	Optional	BOOL	
TRUE, FALSE	TRUE		
<p>According to DS-301 a segmented SDO download request may indicate the data set size or not in the SDO Download Initiate. If this parameter is set to TRUE, the download size is indicated to the remote CANopen node, which is the sensible default for most CANopen devices on the market. If this parameter is set to FALSE the size isn't indicated to the remote CANopen node during SDO Download Initiate, instead the last downloaded segment is marked to indicate the end of transmission.</p>			

3.2.5 Example configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<CANopen_Manager>
  <Manager_Instance>
    <Net_No>0</Net_No>
    <Baudrate>500</Baudrate>
    <Manager_Node_Id>127</Manager_Node_Id>
    <Config_File_Path>./dcf</Config_File_Path>
    <Boot_Time>20000</Boot_Time>
    <Heartbeat_Producer_Time>5000</Heartbeat_Producer_Time>
    <NMT_Master>TRUE</NMT_Master>
    <NMT_Start_All>TRUE</NMT_Start_All>
    <NMT_Auto_Operational>TRUE</NMT_Auto_Operational>
    <NMT_Action_On_Error>STOP_ALL</NMT_Action_On_Error>
    <Node_Instance>
      <Node_Id>1</Node_Id>
      <NMT_Slave>TRUE</NMT_Slave>
      <NMT_Mandatory_Slave>FALSE</NMT_Mandatory_Slave>
      <Guard_Time>0</Guard_Time>
      <Life_Time_Factor>0</Life_Time_Factor>
      <Consumer_Heartbeat_Time>3000</Consumer_Heartbeat_Time>
      <Device_Type_Identification>0x00000191</Device_Type_Identification>
      <Vendor_Identification>0x17</Vendor_Identification>
      <NMT_Check_Nodestate>FALSE</NMT_Check_Nodestate>
      <DCF_File>asyncio.dcf</DCF_File>
    </Node_Instance>
    <Node_Instance>
      <Node_Id>2</Node_Id>
      <NMT_Slave>TRUE</NMT_Slave>
      <NMT_Mandatory_Slave>FALSE</NMT_Mandatory_Slave>
      <Guard_Time>0</Guard_Time>
      <Life_Time_Factor>0</Life_Time_Factor>
      <Consumer_Heartbeat_Time>3000</Consumer_Heartbeat_Time>
      <Device_Type_Identification>0x00000191</Device_Type_Identification>
      <Vendor_Identification>0x17</Vendor_Identification>
      <NMT_Check_Nodestate>FALSE</NMT_Check_Nodestate>
      <CCF_File>asyncio.ccf</CCF_File>
    </Node_Instance>
    <Node_Instance>
      <Node_Id>3</Node_Id>
      <NMT_Slave>TRUE</NMT_Slave>
      <NMT_Mandatory_Slave>FALSE</NMT_Mandatory_Slave>
      <Guard_Time>2000</Guard_Time>
      <Life_Time_Factor>2</Life_Time_Factor>
      <Consumer_Heartbeat_Time>0</Consumer_Heartbeat_Time>
      <Device_Type_Identification>0x00000191</Device_Type_Identification>
      <Vendor_Identification>0x17</Vendor_Identification>
      <NMT_Check_Nodestate>FALSE</NMT_Check_Nodestate>
      <DCF_File>asyncio2.dcf</DCF_File>
    </Node_Instance>
  </Manager_Instance>
</CANopen_Manager>
```

4. Manager API

4.1 Overview

The API consists of three groups of functions:

- Basic functions that have to be called to setup the manager.
- Optional functions that can be called by the application to get more detailed information about the managed nodes..
- Functions to access the remote nodes object dictionary or change their node state.

4.2 Basic manager initialization functions

The following procedure is necessary to setup and start the manager.

1. Initializing the manager library with *mcanOpenInitialize()*.
2. Reading the configuration by calling *mcanOpenConfig()*.
3. Starting the manager by calling *mcanOpenStart()*.

The following procedure is necessary to stop the manager.

1. Terminating the master by calling *mcanOpenShutDown()*.
2. Cleanup the library with *mcanOpenCleanup()*.

mcanOpenInitialize()

- Name:** **mcanOpenInitialize()** - Initialize the manager library
- Call:** `int mcanOpenInitialize(void);`
- Description:** This function initializes the manager library. The function has to be called once before any other API call.
- Return:** Success or error code described in the appendix.
-

mcanOpenCleanup()

- Name:** **mcanOpenCleanup()** - Cleanup the manager library
- Call:** `int mcanOpenCleanup(void);`
- Description:** This function returns all resources that are allocated by the manager library to the operating system. The function has to be called once at the end of the library usage.
- Return:** Success or error code described in the appendix.

mcanOpenConfig()

Name: **mcanOpenConfig()** - Configuring the manager database.

Call:

```
int mcanOpenConfig
(
    char *cfgFile
);
```

Description: This function reads the manager configuration file *cfgFile* and all associated DCF or CCF files for the project specific CANopen network setup. The CAN controller will be initialized and all necessary resources will be allocated.

Return: Success or error code described in the appendix.

mcanOpenStart()

Name: **mcanOpenStart()** - Start the manager.

Call: `void mcanOpenStart(void);`

Description: The function starts the manager daemon tasks which setup and start all nodes in the CANopen network according to the configuration file given with *mcanOpenConfig()*.

Return: N/A.

mcanOpenShutDown()

Name: **mcanOpenShutDown()** - Terminate the manager.

Call: `int mcanOpenShutDown(void);`

Description: This function terminates the CANopen manager and frees all dynamically allocated resources.

Return: Success or error code described in the appendix.

mcanOpenGetVersion()

Name: **mcanOpenGetVersion()** - Return the software-version number

Call:

```
int mcanOpenGetVersion
(
    unsigned short *version
);
```

Description: In parameter *version the current software-version number is returned.

Value range:

Bit 12...15 : level
Bit 8..11 : revision
Bit 0..7 : change

Return: Success or error code described in the appendix.

4.3 Optional node control functions

The following functions are optional node control functions that can be called by an application to monitor the state of the remote nodes.

mcanOpenEvent()

Name: **mcanOpenEvent()** - Receive events from the manager

Call:

```
int mcanOpenEvent
(
    EVENT_T *event,
    int wait
);
```

Description: This function has to be called by the application to receive errors, warnings and indications from the manager daemons. The application can either block until the manager indicates an event to the application by setting parameter *wait* to 1 which is typically done with a separate task or can poll for events by setting parameter *wait* to 0. In the structure `EVENT_T` which is initialized by the manager the event type and additional parameter is stored:

```
typedef struct {
    long event;
    int net;
    int node;
    union {
        char c;
        unsigned char uc;
        short s;
        unsigned short us;
        int i;
        unsigned int ui;
        long l;
        unsigned long ul;
        struct {
            unsigned short index;
            unsigned short subindex;
            ERROR_T err;
        } write;
        char str[64];
    } para;
} EVENT_T;
```

This structure has to be interpreted in the following way:

The MSB of EVENT_T.event is set, if EVENT_T.net is invalid, the next is then accordingly set, if EVENT_T.node is invalid.

The following 22 bits mark the valid parameter of the Union EVENT_T.para. The least significant byte contains the message number of the message field corresponding to the parameter type. Flags and messages needed to evaluate 'event' are to be taken from 'mcanopen.h' and 'paraInfo.h'.

Example:

```
EVENT_T.event = 0x00001022
EVENT_T.net   = 1
EVENT_T.node  = 5
EVENT_T.para.i = 2
```

```
NET_NOT_VAL = 0x80000000
NOD_NOT_VAL = 0x40000000
PARA        = 0x3FFFFFF0
```

```
0x00001022 & NET_NOT_VAL == 0    ==> EVENT_T.net is valid
0x00001022 & NOD_NOT_VAL == 0    ==> EVENT_T.node is valid
0x00001022 & PARA == 0x00001000 ==> EVENT_T.para is valid
0x00001000 == PARA_I              ==> EVENT_T.para.i is valid
0x00001022 & 0xFF == 0x22        ==> lsb is 0x22, i.e. the error is
                                     the 34th error with integer
                                     parameter
```

Result: EVENT_T.net, Event_T.node, iParaInfo[0x22], EVENT_T.para.i

```

      |   |   |   |
Meaning:  "Net: 1 Node: 5 guarding toggle error no  2"
```

Return:

Success or error code described in the appendix.

mcanOpenGetNodeState()

Name: mcanOpenGetNodeState() - Return the current CANopen node state.

Call:

```
int mcanOpenGetNodeState
(
    int net,
    int nodeID,
    int *state
);
```

Description: This function returns the current CANopen node state of the node *nodeId* on logical CANopen net *net*. The state is returned in the parameter *state* and can have the following numerical values:

Numerical value	Node state
0x00	Offline
0x04	Stopped
0x05	Operational
0x7F	Pre-Operational

Return: Success or error code described in the appendix.

4.4 Extended Functions

The extended functions can be used to read/write the object dictionary of CANopen nodes and to force a change of node state.

mcanOpenReadSDO()

Name: **mcanOpenReadSDO()** - Reading an object-directory entry

Call:

```
int mcanOpenReadSDO
(
    unsigned short net,
    unsigned short modID,
    int index,
    int subindex,
    void *buffer,
    int *len,
    ERROR_T *error
);
```

Description: As SDO client this function transfers the data of the object dictionary entry defined by *index* and *subindex* from the remote module *modId* into the memory of the caller given by the parameter *buffer*. The available buffer size has to be indicated by the caller in the parameter *len*. If the operation was successfully completed *buffer* contains the data and *len* contains the number of bytes received.

If the remote node given by *modId* is managed by the master the internal SDO handle is used for SDO transfer otherwise a new handle is opened for this operation and closed afterwards. The receive and transmit timeout for the internal handle can be configured individually for each node with the parameters `SDO_TX_TIMEOUT` and `SDO_RX_TIMEOUT` in the manager configuration file. The default RX/TX timeout for nodes not managed by the master is 200 ms.

If the function returned with the error code `MCAL_ABORT_DOMAIN` the following structure that has to be provided by the caller with the parameter *error* which is used to return the abort codes defined in DS-301.

```
typedef struct{
    unsigned char eclass;
    unsigned char code;
    unsigned char spec;
    unsigned char global;
} ERROR_T;
```

Return: Success or an error code listed in the appendix.

mcanOpenWriteSDO()

Name: mcanOpenWriteSDO() - Writing an object-directory entry

Call:

```
int mcanOpenWriteSDO
(
    unsigned short net,
    unsigned short modID,
    int index,
    int subindex,
    int type,
    void *buffer
    int len,
    ERROR_T *error
);
```

Description: As SDO client this function writes the content of the data buffer *buffer* into the object directory of *modID* into the entry defined by *index* and *subindex*.

The parameter *type* specifies the data type. The following table contains the list of supported data types and their default transfer size.

Type descriptor	Transfer size in bytes
TYP_CHAR	1
TYP_INT8	1
TYP_INT16	2
TYP_INT32	4
TYP_UINT8	1
TYP_UINT16	2
TYP_UINT32	4
TYP_FLOAT	4
TYP_VIS_STRING	N/A
TYP_OCT_STRING	N/A
TYP_DATE	6
TYP_TIME_OF_DAY	6
TYP_TIME_DIFFERENCE	6
TYP_DOMAIN	N/A

If you want to transmit data with an arbitrary size you have to provide the length of the buffer with the parameter *len*.

If the transfer size of the buffer is less than 5 bytes the expedited SDO transfer mode instead of the segmented SDO transfer mode is used. In addition the parameter *type* can be logically or'd with `TYP_INDICATE_SIZE` or `TYP_INDICATE_NO_SIZE` to determine that the data transfer size is indicated to the SDO server during initialization of the SDO transfer or not. The default behaviour is *length indication* for expedited transfer and *no length indication* for segmented transfer.

If the remote node given by *modId* is managed by the master the internal SDO handle is used for SDO transfer otherwise a new handle is opened for this operation and closed afterwards. The receive and transmit timeout for the internal handle can be configured individually for each node with the parameters `SDO_TX_TIMEOUT` and `SDO_RX_TIMEOUT` in the master configuration file (see page ?). The default RX/TX timeout for nodes not managed by the master is 2000 ms.

If the function returned with the error code `MCAL_ABORT_DOMAIN` the following structure that has to be provided by the caller with the parameter *error* which is used to return the abort codes defined in DS-301.

```
typedef struct{
    unsigned char eclass;
    unsigned char code;
    unsigned char spec;
    unsigned char global;
} ERROR_T;
```

Return: Success or an error code listed in the appendix.

mcanOpenNodeControl()

Name: **mcanOpenNodeControl()** - Requesting the control over a remote node

Call:

```
int mcanOpenNodeControl
(
    int net,
    int nodeID,
    long flags
);
```

Description: Obsolete function, present for API backward compatibility.

Return: Always success.

mcanOpenStartNode()

Name: **mcanOpenStartNode()** - Start a remote node

Call:

```
int mcanOpenStartNode
(
    int net,
    int nodeID
);
```

Description: This functions starts the remote node *nodeId* on CANopen network *net*.

Return: Success or an error code listed in the appendix.

mcanOpenResetNode()

Name: **mcanOpenResetNode()** - Reset a remote node

Call:

```
int mcanOpenResetNode
(
    int net,
    int nodeID
);
```

Description: This functions resets the remote node *nodeId* on CAN network *net*.

Return: Success or an error code listed in the appendix.

mcanOpenEnterPreop()

Name: **mcanOpenEnterPreop()** - Make remote node enter Pre-Operational

Call:

```
int mcanOpenEnterPreop
(
    int net,
    int nodeID
);
```

Description: This functions makes the remote node *nodeId* on CAN network *net* enter the node state Pre-Operational.

Return: Success or an error code listed in the appendix.

mcanOpenStopNode()

Name: **mcanOpenStopNode()** - Stop the remote node

Call:

```
int mcanOpenEnterPreop
(
    int net,
    int nodeID
);
```

Description: This functions makes the remote node *nodeId* on CAN network *net* enter the node state Stopped.

Return: Success or an error code listed in the appendix.

5. Additional Tools

The CANopen libraries come with three command line tools to manage CANopen configuration files. If the target operating system doesn't support self-hosted development usually these distributions come with the Win32 versions of these tools.

For a list of arguments and options start the tool without parameters on the command line.

5.1 dcfstest

The dcfstest tool is intended to verify DCF or EDS file with respect to the related standard /3/. The CANopen manager library is using the identical parser so you can check beforehand if the network bootup fails because of errors in a DCF configuration file.

The manager ignores all problems which are indicated as a warning by the dcfstest tool.

5.2 dcf2ccf

The dcf2ccf tool is intended to convert a DCF file into a file in the Concise Configuration Format (CCF) which is specified in /2/. The advantage of this format is the very compact representation of the configuration data and the faster network bootup as the manager can skip the time consuming step of parsing the configuration file in DCF format.

5.3 ccfred

The ccfred tool is intended to show the contents of a CANopen device configuration file in the binary CCF format.

6. Appendix

6.1 Error Codes

The following tables list the possible error codes that can be returned by the manager library API calls. They are divided into three groups:

- Error codes of the driver.
- Error codes returned by reading the DCF files.
- General master error codes.

When evaluating return values you should never use the numerical values but should always use the constants defined for this error codes.

MCANOPEN_OK

Success (no warning or error).

Severity	Success
Description	The operation was executed without any errors.
Function	All functions

MCAL_CAN_RX_TIMEOUT

RX timeout for CAN receive operation

Severity	Application specific
Description	The application performed an operation to receive data from a remote node but the receive operation timed out. It is application specific if this is an expected behaviour or an error.
Solutions	If this behaviour is not expected: <ul style="list-style-type: none"> • Check remote node. • Check cables. • Check baudrate.
Function	<i>mcanopenReadSDO()</i> <i>mcanopenWriteSDO()</i>

MCAL_CAN_TX_TIMEOUT

TX timeout for CAN transmit operation

Severity	Error / Warning
Description	The application performed an operation to transmit data to a remote node but the transmit operation timed out. The reason for this behaviour could be a very high CAN bus load caused by CAN messages with a higher priority or a communication error.
Solutions	<ul style="list-style-type: none">• Check cables.• Check baudrate.
Function	<i>mcanopenReadSDO()</i> <i>mcanopenWriteSDO()</i> <i>mcanopenStopNode()</i> <i>mcanopenStartNode()</i> <i>mcanopenResetNode()</i>

MCAL_CAN_TX_ERROR

TX error in CAN transmit operation

Severity	Error
Description	The application performed an operation to transmit data to a remote node but the transmit operation timed could not be performed. The reason for this behaviour is a communication error.
Solutions	<ul style="list-style-type: none">• Check cables.• Check baudrate.
Function	<i>mcanopenReadSDO()</i> <i>mcanopenWriteSDO()</i> <i>mcanopenStopNode()</i> <i>mcanopenStartNode()</i> <i>mcanopenResetNode()</i>

MCAL_CAN_CONTR_OFF_BUS

Error in CAN transmit operation

Severity	Error
Description	The application performed an operation to transmit data to a remote node but the transmit operation could not be performed as the CAN controller is in BUS OFF state due to too many received error frames.
Solutions	Try again later as driver will try an automatic recovery. If an recovery is not possible you must check your cable and if all remote nodes using the same baudrate.
Function	<i>mcanopenReadSDO() mcanopenWriteSDO() mcanopenStopNode() mcanopenStartNode() mcanopenResetNode()</i>

MCAL_CAN_CONTR_WARN

Error in CAN transmit operation

Severity	Error
Description	The application performed an operation to transmit data to a remote node but the transmit operation could not be performed as the controller is in warning state due to too many received error frames.
Solutions	Try again later as driver will try an automatic recovery. If an recovery is not possible you must check your cable and if all remote nodes using the same baudrate.
Function	<i>mcanopenReadSDO() mcanopenWriteSDO() mcanopenStopNode() mcanopenStartNode() mcanopenResetNode()</i>

MCAL_CAN_CONTR_BUSY

Error in CAN transmit operation

Severity	Error / Warning
Description	The application performed an operation to transmit data to a remote node but this transmit operation could not be performed as the last transmit operation for this Id was not completed.
Solutions	<ul style="list-style-type: none">• Decrease update frequency in your application.• Use flag TX_DONE_PDO in PDO definition.
Function	<i>mcanOpenWriteSDO()</i> <i>mcanOpenReadSDO()</i>

CANOPEN_DCF_INIT_ERROR

Error during initialization

Severity	Error
Description	A DCF file could not be opened.
Solutions	<ul style="list-style-type: none">• Check paths and filename in master configuration file.• Check access rights on host.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_OUT_OF_MEMORY

Error during initialization

Severity	Error
Description	Not enough memory to process a DCF file.
Solutions	Increase memory size that can be utilized by the CANopen master.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_WRONG_NODE_ID

Error during initialization

Severity	Error
Description	The node id given in the master configuration file and given in the DCF file in section [Device Commissioning] differ.
Solutions	Use same module id's.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_WRONG_BAUDRATE

Error during initialization

Severity	Error
Description	The baudrate given in the master configuration file and given in the DCF file in section [Device Commissioning] differ or the baudrate is not a valid CANopen baudrate.
Solutions	<ul style="list-style-type: none"> • Use same baudrates in configuration and DCF file. • Use standard baudrate.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_MISSING_SECTION

Error during initialization.

Severity	Error
Description	A required section is missing in the DCF file.
Solutions	Check and correct DCF file. Use the tool <i>dcftest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_MISSING_ENTRY

Error during initialization

Severity	Error
Description	A required keyword is missing in a section of the DCF file.
Solutions	Check and correct DCF file. Use the tool <i>dcftest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_WRONG_PARAMETER

Error during initialization

Severity	Error
Description	A parameter of a certain keyword is invalid or missing.
Solutions	Check and correct DCF file. Use the tool <i>dcftest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_OUT_OF_RANGE

Error during initialization

Severity	Error
Description	The value of a parameter is out of range according to its data type. E.g. the value of an unsigned char parameter is greater than 256.
Solutions	Check and correct DCF file. Use the tool <i>dcftest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_OUT_OF_LIMITS

Error during initialization

Severity	Error
Description	The value of a parameter is out of range according to the given limits with keyword <i>HighLimit</i> and <i>LowLimit</i> .
Solutions	Check and correct DCF file. Use the tool <i>dcftest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_DOUBLE_SECTION

Error during initialization

Severity	Error
Description	A section name is not unique within the DCF file.
Solutions	Check and correct DCF file. Use the tool <i>dcftest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_DOUBLE_KEYWORD

Error during initialization

Severity	Error
Description	A keyword name is not unique within the scope of a section in the DCF file.
Solutions	Check and correct DCF file. Use the tool <i>dcftest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_UNKNOWN_KEYWORD

Error during initialization

Severity	Error
Description	An unknown keyword was found in the DCF file.
Solutions	Check and correct DCF file. Use the tool <i>dctest</i> for more details.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_WRONG_NET_NO

Error during initialization

Severity	Error
Description	The static tables of the master are too small to configure a node on this net number.
Solutions	Choose a lower number for this network.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

CANOPEN_DCF_WRONG_MODULE_NO

Error during initialization

Severity	Error
Description	The module number is out of the valid range $0 < \text{modId} < 128$.
Solutions	Choose a correct module number.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

MCAL_PARA_ERROR

Invalid parameter

Severity	Error
Description	One or more parameter of a function call were invalid.
Solutions	Compare parameter value with ranges given in the manual.
Function	All functions

MCAL_SYS_ERROR

Internal error

Severity	Error
Description	An internal data structure that is necessary for operation was not initialized or could not be initialized.
Solutions	<ul style="list-style-type: none"> • Compare order of initialization with manual. • If an initialization file is given check if the file exist and users access rights for the file are sufficient for the operation.
Function	All functions

MCAL_NO_NET

Net number invalid

Severity	Error
Description	The CANopen master is not initialized at all or a certain net is not initialized.
Solutions	<ul style="list-style-type: none"> • Compare order of initialization with manual. • Check if the net is registered in the master configuration file.
Function	All functions

MCAL_OBJ_NO_MEMORY

Error allocating a resource

Severity	Error
Description	Allocating a resource like memory or a synchronization object that is necessary to complete the operation failed.
Solutions	Increase the available memory for the CANopen master process.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i> <i>mcanopenStart()</i>

MCAL_DOWNLOAD_ABORT_SLAVE

A slave aborted a CAL configuration

Severity	Error
Description	A CAL slave aborted the configuration download.
Solutions	Check CAL slave manual.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i> <i>mcanopenStart()</i>

MCAL_WRONG_SLAVE_ANSWER

Error during CAL configuration

Severity	Error
Description	A CAL slave produced a protocol error during CAL configuration.
Solutions	Check CAL slave manual.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i> <i>mcanopenStart()</i>

MCAL_WRONG_SERVER_ANSWER

Protocol error

Severity	Error
Description	An answer of a SDO server was wrong with respect to the SDO protocol.
Solutions	Check manual of remote CAL/CANopen slave.
Function	<i>mcanopenReadSDO()</i>

MCAL_TOGGLE_ERROR

Protocol error

Severity	Error
Description	An answer of a SDO server was wrong with respect to the SDO protocol.
Solutions	Check manual of remote CAL/CANopen slave.
Function	<i>mcanopenReadSDO()</i> <i>mcanopenWriteSDO()</i>

MCAL_BUFFER_OVERFLOW

Insufficient memory

Severity	Error
Description	The memory that was provided by the caller was insufficient to keep all the data that was received by the SDO server.
Solutions	Increase memory to store data.
Function	<i>mcanopenReadSDO()</i>

MCAL_ABORT_DOMAIN

Protocol error

Severity	Error
Description	The SDO server of a remote CANopen node aborted the transfer.
Solutions	Check error code returned by the call and the manual of the remote CANopen slave to find out the reason of the error condition.
Function	<i>mcanopenReadSDO()</i> <i>mcanopenWriteSDO()</i>

MCAL_NODE_NOT_EXISTEND

Unknown node

Severity	Error
Description	A remote node with a given module id is not managed by the master for the given net.
Solutions	Check if node is registered in the master configuration file and is correctly configured during startup.
Function	<i>mcanopenGetNodeInfo()</i> <i>mcanopenGetNodeState()</i> <i>mcanopenNodeControl()</i> <i>mcanopenResetNode()</i> <i>mcanopenStartNode()</i> <i>mcanopenStopNode()</i>

MCAL_WRONG_GUARD_ID

Guard identifier out of range

Severity	Error
Description	The identifier for node guarding is out of range with respect to the CAL or CANopen draft standard.
Solutions	Compare valid range in standard with given value in configuration file.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

MCAL_NEW_GUARD_PARA

Guard identifier mismatch

Severity	Warning
Description	The identifier of a CANopen slave for node guarding given in configuration file and given in DCF file differ.
Solutions	Use the same identifier or use only DCF file to configure guarding information.
Function	<i>calMStart()</i> <i>mcanopenConfig()</i>

MCAL_NODE_NOT_FREE

Getting node control failed

Severity	Warning / Error
Description	Taking over the node control for a given node on a given net was not possible as the CANopen master itself is accessing the node at the moment.
Solutions	Try taking over the node control later.
Function	<i>mcanopenGetNodeControl()</i>

MCAL_NO_EVENT

Event queue empty

Severity	Warning
Description	If the master event queue is polled this is an indication that the event queue is empty at the moment.
Solutions	N/A
Function	<i>mcanopenEvent()</i>

MCAL_UNSUPPORTED

Unsupported operation

Severity	Warning / Error
Description	The operation could not be performed as this service is unsupported in this version of the CAL/CANopen master.
Solutions	N/A
Function	N/A



CANopen Slave

Software Manual



NOTE

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd gmbh**.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd gmbh** nor the rights of others.

esd electronic system design gmbh

Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd-electronics.com
Internet: www.esd-electronics.com

USA / Canada:

esd electronics Inc.

525 Bernardston Road
Suite 1
Greenfield, MA 01301
USA

Phone: +1-800-732-8006
Fax: +1-800-732-8093
E-mail: us-sales@esd-electronics.com
Internet: www.esd-electronics.us

Manual File:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\CAL-COPN\CANOPEN\Coslv221.wpd
Date of Print:	2008-06-05

Described Software:	CANopen-Slave
Revision:	2.2.x

Order Number:	CANopen-Slave: C.1405.88
----------------------	--------------------------

Changes in the chapters

The changes in the user's manual listed below affect changes in the software, as well as changes in the description of the facts only.

Alterations in the appendix versus previous revisions	Alterations in software	Alterations in documentation
Extended documentation of SYNC and NMT error control objects.	x	x
Documentation of new entry canOpenCreateNetworkEx().	x	x
Documentation of SYNC generation.	x	x
Revised canOpenExtendDictionary() and canOpenInitDictionary()		x
Documentation of new entry canOpenInitDictionaryTs()	x	x
Documentation of the entry canOpenSetParameter()		x
Documentation of the new entry canOpenGetParameter()	x	x
Documentation of object handler with timestamps.	x	x
Documentation of EV_BOOTUP for the node's event handler	x	x
Documentation of new macros BEGIN_DICTIONARY_TABLE_TS and END_DICTIONARY_TABLE_TS		x

Contents	Page
1. Reference	6
2. Introduction	7
3. CANopen Slave	8
3.1 Overview	8
3.2 Object Dictionary.	11
3.3 NMT state machine.	11
3.4 Heartbeat, Node Guarding and Life Guarding	11
3.5 Synchronization (SYNC) Object	12
3.6 Emergency (EMCY) Object	12
4. Program Interface	15
4.1 Management Services	15
canOpenCreateNetwork()	15
canOpenCreateNetworkEx()	15
canOpenRemoveNetwork()	17
canOpenCreateNode()	17
canOpenCreateNodeEx()	20
canOpenDeleteNode()	25
canOpenActivateNode()	25
canOpenGetNodeInfo()	26
canOpenResetNode()	26
canOpenWaitForNodeState()	27
4.2 Local Object Directory	28
canOpenExtendDictionary()	28
canOpenInitDictionary()	30
canOpenInitDictionaryTS()	31
canOpenReadDictionary()	32
canOpenWriteDictionary()	33
canOpenGetDictionaryHnd()	34
canOpenReadDictionaryHnd()	35
canOpenWriteDictionaryHnd()	35
4.3 PDO Services	36
canOpenDefinePDO()	36
canOpenWritePDO()	38
canOpenReadPDO()	39
canOpenRequestPDO()	39
4.4 Error Situations and Emergency (EMCY) Objects	40
canOpenSetError()	43
canOpenResetError()	44
4.5 Assistant Functions	45
canOpenGetVersions()	45
canOpenSetParameter()	46
canOpenGetParameter()	47
4.6 Event handler	48
Object Eventhandler without timestamps	48
Object Eventhandler with timestamps	48

4.7	Macros	51
	Dictionary Entry Tables	51
	PDO Mapping Tables	53
	PDO Tables	54
5.	Error Codes of Slave-Service Functions	57

1. Reference

- /1/: CiA DS-301, CANopen - Application Layer and Communication Profile V4.0.2, February 2002
- /2/: electronic system design gmbh, CAN-Interface Manual, December 1996
- /3/: electronic system design gmbh, CAL/CANopen Systeminterface Manual, December 1996
- /4/: electronic system design gmbh, CAL/CANopen Porting Guide, December 1996
- /5/: CiA DS-102, CAN Physical Layer for Industrial Applications, April 1994
- /6/: CiA DS-201, CAN Reference Model, February 1996
- /7/: CiA DS-401, Device Profile for I/O-Modules, December 1996

2. Introduction

The CANopen slave library allows an easy development of CANopen based slave devices for sophisticated process control of current automation systems or for simulation and test purposes.

Some highlights of the library are:

- ▶ Comprehensive set of services based on the CANopen specification CiA DS-301 V4.1 to easily integrate CANopen slave functionality into an application.
- ▶ Support for several (real-time) operating systems and CAN adapter available with the same OS and hardware independent proven CANopen slave core.
- ▶ Comes as fully multi-threaded shared or static library which can be used by several applications at the same time. All CANopen related tasks like SDO server replies, error control, etc. is handled in background.
- ▶ Allows the implementation of several independent CANopen devices with separated object dictionaries communicating on the same or different physical CAN ports.
- ▶ All CANopen slave functionality is fully configurable at runtime.
- ▶ Consistent API independent of the CPU architecture, operating system or CAN hardware makes a migration to a different platform easy.
- ▶ Support to optionally timestamp received data.

3. CANopen Slave

Based on this library it is possible to create up to 255 independent virtual CANopen slave devices for up to 16 physical CAN nets.

The application programming interface (API) of the CANopen slave library is a procedural API which is defined in the header file scanopen.h.

Depending on the operating system the library has to be either linked to the application or is implemented as a shared library which can be loaded dynamically.

In order to understand this document some basic principles of CANopen are explained in this chapter. For further details please refer to /1/.

3.1 Overview

The application creates one or more CANopen slaves with a node-ID that has to be unique in the physical CAN network. Each CANopen slave node has an individual object directory, at least one service data object (SDO) and one emergency object (EMCY) whose defaults COB-IDs are based on the node-ID (default connection set). The application can extend the object directory with manufacturer specific entries or according to standardized device profiles (/7/) and map the process variables into the object directory as shown in figure 1. The entries of the object dictionary can be mapped into process data objects (PDO) which are transmitted or received using the CAN bus.

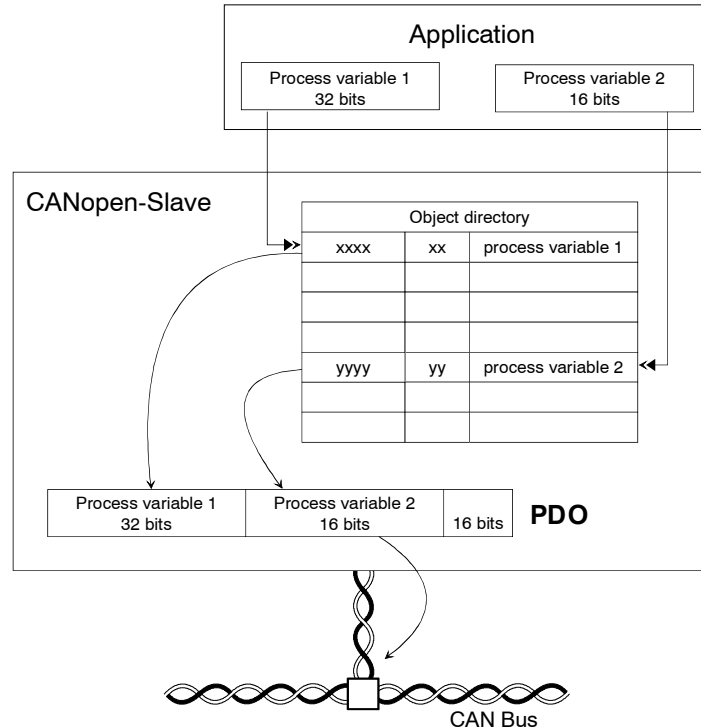


Fig. 1: Operation mode of CANopen slave

The COB identifiers of the PDOs, the PDO mapping and a number of additional parameter can be configured by the application as well as by a CANopen manager (dynamic mapping).

The communication between the application and the CANopen library is based on a procedural interface, the asynchron communication from the CANopen library to the application is event driven based on callback handlers.

The following steps are necessary creating a virtual slave node and make this node available for configuration and control by a CANopen manager and communication with further CANopen slaves.

1. Initialization of the CAN bus and start of the NMT daemon by calling *canOpenCreateNetwork()* or *canOpenCreateNetworkEx()*.
2. Initialization of the virtual slave by attaching the node event handler and defining the entries of the object directory in the **Communication Profile Area** calling *canOpenCreateNodeEx()*.
3. Creation of additional entries in the **Manufacturer Specific Area** and the **Standardized Device Profile Area** of the object directory by calling *canOpenExtendDictionary()*. Initialization of these entries and assignment of the object event handler by calling *canOpenInitDictionary()* or *canOpenInitDictionaryTs()*. Alternatively you can use a set of macros to ease the programming effort.
4. Creation and initialization of the PDOs by calling *canOpenDefinePDO()*. Alternatively you can use a set of macros to ease the programming effort.
5. Change the node state to **Pre-Operational** by calling *canOpenActivateNode()*.
6. If the node state changes to **Operational**, PDOs can be exchanged with other CANopen slave nodes. PDO communication is different for synchronous and asynchronous PDOs and depends on the configured PDO communication parameter:

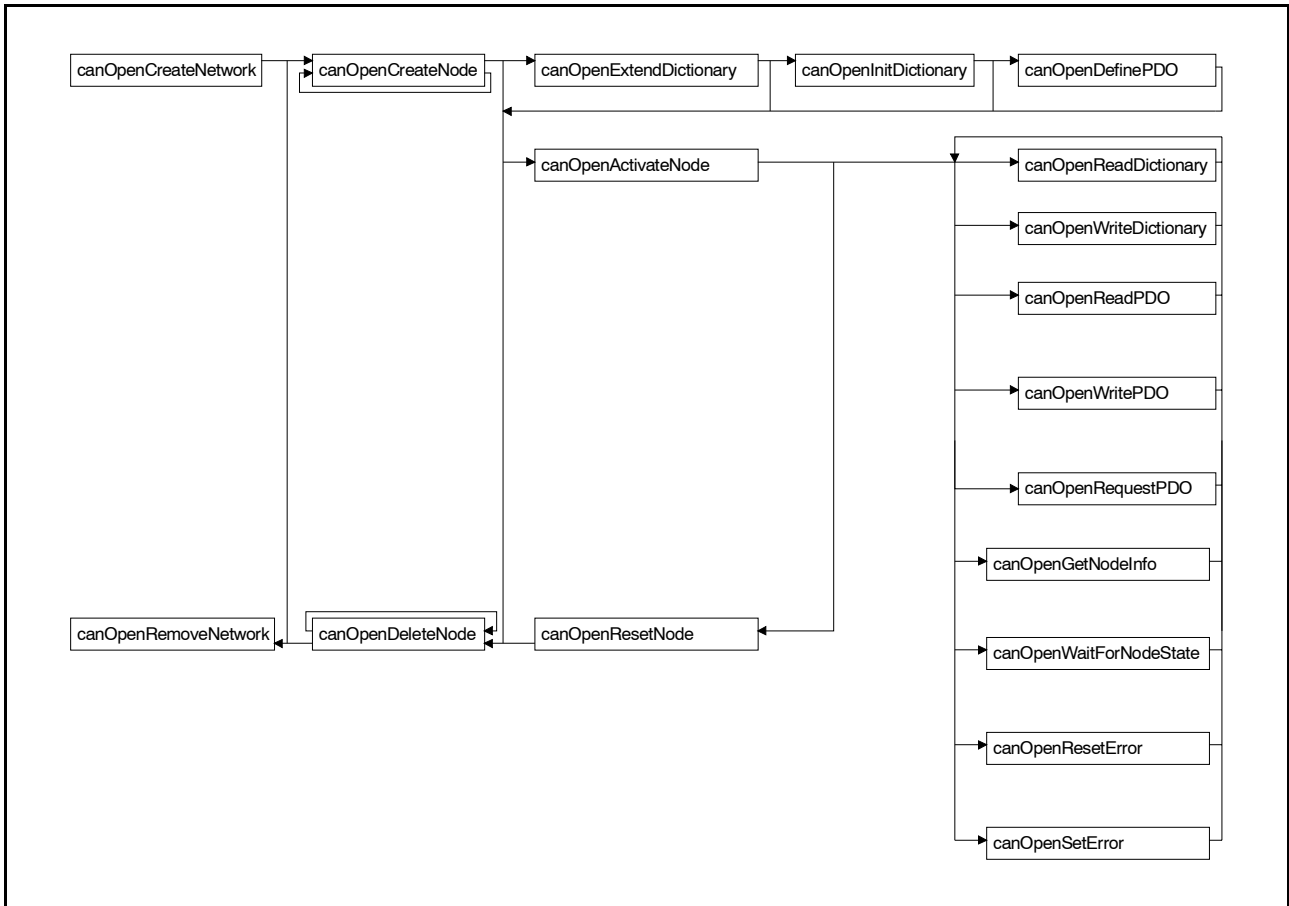


Fig. 2: Flow chart of API Calls

The flow chart above shows the order of API calls to create and manage the CANopen slave. A detailed description of the API is described in chapter 4.

3.2 Object Dictionary.

The object dictionary is the crucial part for process data exchange between the application and the CANopen slave library. The object dictionary entries in the Device Profile Specific Area and the Manufacturer Specific Area are fully configurable by the application. The PDOs as well as the SDO services work directly with these dictionary entries. For an object dictionary entry, mapped into a TPDO, an update performed by the application might, depending on the PDO configuration, immediately cause the transmission of this PDO. For every entry mapped into an RPDO a callback handler can be attached, so the CANopen slave library supports a very fast event-based mechanism to indicate the update caused by another CANopen slave device.

3.3 NMT state machine.

The CANopen slave implements the NMT state machine according to /1/. After creating the node with *canOpenCreateNodeEx()* the slave is in the special state **NodeInit**. In this state it's possible for the application to extend and initialize the local dictionary and define the PDOs. In this state the CANopen slave node isn't active on the CAN bus. After this task is completed a call to **canOpenActivateNode()** changes the node state to **Pre-Operational** or to **Operational** if configured as auto-start device. Further node state changes between **Pre-Operational**, **Operational** and **Stopped** or a node reset are caused by NMT messages of the CANopen manager. The application keeps track of the current node state with the help of it's node event handler and/or the API *canOpenGetNodeInfo()*. The application can switch back into the NodeInit state with the API *canOpenResetNode()*.

3.4 Heartbeat, Node Guarding and Life Guarding

The CANopen specification /1/ defines a Heartbeat and a Node Guarding mechanism for error control which are both supported by the slave stack.

If configured for Node Guarding the NMT manager "polls" the CANopen device for it's current node state on a regular basis to detect failures. In addition the node can setup a timer with each NMT master request and can use the expiration of this timer as an indication that the communication with the NMT master is interrupted (Life Guarding).

If configured for Heartbeat the slave node transmits the heartbeat message with it's current node state autonomously with a configurable heartbeat producer time which is checked by the NMT master. To support a similar mechanism to the Life Guarding the node can also be configured as a heartbeat consumer to monitor the heartbeat of the NMT master and other CANopen nodes.

Today it is recommended to use the heartbeat mechanism instead of the node guarding because it consumes less CAN bus bandwidth (no polling) and is more flexible.

The CANopen stack handles both error control mechanisms completely in background and indicates all error control related events to the application, which can configure an application specific behavior in case of a NMT error control failure.

3.5 Synchronization (SYNC) Object

According to /1/ the synchronous communication in CANopen is based on a SYNC object, which is a special message with no data. The COB-ID of the SYNC object can be configured for each node individually to allow multiple SYNC signals in a system. The common use case is to have only one SYNC object with the default COB-ID of 0x80.

The CANopen slave can be configured as SYNC consumer and/or SYNC generator¹. As a SYNC consumer on reception of the SYNC object all objects mapped into synchronous RPDOs, received since the last SYNC object, are indicated to the application and new data for all objects mapped into synchronous TPDOs is requested by the application.

The generation of the SYNC object requires a special CAN device driver or hardware which supports SYNC generation with a minimized jitter. These device drivers are currently not available for all supported OS platforms and/or CAN devices. If a CANopen node is configured as SYNC generator you have to make sure that there is only one SYNC generator for this SYNC signal on the same network.

3.6 Emergency (EMCY) Object

According to /1/ error states are indicated on the CAN bus by means of the Emergency (EMCY) object. Such an error condition can be assigned to one of the following categories:

Communication and Configuration Errors:

- Errors on CAN controller communication layer.
- Receive buffer overflow.
- Heartbeat or Life Guarding Errors.
- Configured PDO size mismatches.

Application Errors:

All types of errors, which are application specific like problems related to current, voltage, temperature, etc.

Errors which belong to the 1st category are detected by the CANopen stack autonomously. In addition to send an EMCY object the error is indicated to the application via the node's event handler. Errors of the 2nd category have to be indicated to the stack using the related slave API.

¹ The ability to generate SYNC objects depends on the support by the CAN hardware and the CAN driver. Only hardware/driver combinations which support the *Scheduling* of CAN frames support the generation of the SYNC object.

The 8 byte EMCY object has the following structure:

Identifier	Data							
Default:	0	1	2	3	4	5	6	7
0x80 + NodeID	Emergency Error Code		Error Register	Manufacturer Specific Error Code				
Index 0x1014	Index 0x1003 (Bit 0 - 15)		Index 0x1001	Index 0x1003 (Bit 16 - 31)				

The *Emergency Error Code* describes the reason for the error. A list of pre-defined error codes is defined in /1/. Additional error codes may be defined in the CANopen device profiles. If the slave stack is configured to support an error history via the pre-defined error field (0x1003) the *Emergency Error Code* becomes the LSW of the related entry in the error history. The EMCY object also reflects the current state of the *Error Register* (0x1001), which groups errors in certain categories to indicate if further error conditions are pending. The EMCY message also contains a manufacturer-specific part which describes the error in more detail. A repaired error situation is indicated with the *Emergency Error Code* set to 0 (Reset Error).

If the EMCY object is caused by a communication or configuration error detected internally the manufacturer specific part is used as described below and the bytes 3 and 4 of the EMCY object become the MSW of the related entry in the error history.

Data Byte	3	4	5	6	7
Description	Temporary Bits	Sticky Bits	Reason	Info1	Info2

The *Temporary Bits* indicate temporary error conditions which are reset if the error is repaired:

Bit	Description
0	NMT Error Control (Guarding/Heartbeat error).
1	CAN Controller Passive.
2	CAN Controller Bus Off
3-4	Reserved for future use by the CANopen stack.
5-7	Application specific temporary error.

The *Sticky Bits* indicate error conditions which are indicated even if the error is already repaired.

Bit	Description
0	CAN Controller Error.
1	Receive FIFO Overrun.
2	PDO Length Error.
3-4	Reserved for future use by the CANopen stack.
5-7	Application specific temporary error.

The parameter *Reason*, *Info1* and *Info2* contain additional information to an internal generated EMCY object because of a communication or configuration error. The table below lists the internally generated EMCY messages and the meaning of the related manufacturer-specific parameter.

Error Code	Description	Reason	Info1	Info2
0x8100	Message Lost Error	1 = Rx Daemon FIFO	# of Lost Messages	0
0x8110	CAN overrun (objects lost)	0	0	0
0x8120	CAN in error passive mode	0	0	0
0x8130	Life Guard or Heartbeat Error	0	0	0
0x8140	Recover from Bus-Off	0	0	0
0x8210	PDO not processed (length error)	Internal PDO number	CAN msg length	PDO length
0x8220	PDO length exceeded	Internal PDO number	CAN msg length	PDO length

4. Program Interface

The following chapter describes the interface of the CANopen slave. The meaning of error codes of the returned values is shown in the appendix.

4.1 Management Services

The services described below serve the initialization, control and monitoring of CANopen networks and CANopen slaves.

canOpenCreateNetwork()

Name: `canOpenCreateNetwork()` - initializing the network (**deprecated**)

Synopsis:

```
int canOpenCreateNetwork
(
    int          NetNo,          /* number of CAN interface */
    char *       NetName,       /* textual description */
    unsigned short Baudrate     /* baudrate */
)
```

Description: This routine initializes interface *NetNo* and generates a network object in the internal database. Optionally a pointer to a textual description can be given. *Baudrate* is specified in kbit/s. The support of baudrates depends on CAN-layer-2 driver.

Return: 0 or an error code described in the appendix.

canOpenCreateNetworkEx()

Name: `canOpenCreateNetworkEx()` - Extended initialization of the CANopen network

Synopsis:

```
int canOpenCreateNetworkEx
(
    int          NetNo,          /* Number of CAN interface */
    SLAVE_NET_INFO *pNetInfo,   /* Network configuration */
)
```

Description: This routine initializes interface *NetNo* and generates a network object in the internal database. The caller configures the parameter with the pointer *pNetInfo* to an initialized structure of the type `SLAVE_NET_INFO` described below.

The structure `SLAVE_NET_INFO` comprises all crucial network or stack specific parameter. The complete structure should be filled with zeros before it is initialized. Some flags of *ulOptions* just indicate which other members of the structure have to be initialized with proper values or can be left set to 0. The

following table should provide an overview of *ulOptions*.

Flag in <i>ulOptions</i>	Affected structure member or Description
THREAD_PRIOS_OVERWRITE	sPrioNMT, sPrioSDO, sPrioPDO and sPrioEMCY
THREAD_PRIOS_NATIVE	sPrioNMT, sPrioSDO, sPrioPDO and sPrioEMCY
NORMALIZE_TIMESTAMPS	Timestamps of the node's data callback handler are normalized to 1 us instead using raw values.

pNetName	N/A
Optional pointer to a textual description of this network or NULL.	

usBaudrate	N/A
The CAN bit rate which should be used for CAN communication.	

usDebugMask	N/A
In special debug builds of the slave stack this parameter configures a mask to control the debug trace. In release builds of the stack this parameter is ignored.	

sPrioNMT	THREAD_PRIO_OVERWRITE and THREAD_PRIOS_NATIVE
Thread priority of the NMT thread.	

sPrioSDO	THREAD_PRIO_OVERWRITE and THREAD_PRIOS_NATIVE
Thread priority of the SDO thread.	

sPrioPDO	THREAD_PRIO_OVERWRITE and THREAD_PRIOS_NATIVE
Thread priority of the PDO thread.	

sPrioEMCY	THREAD_PRIO_OVERWRITE and THREAD_PRIOS_NATIVE
Thread priority of the EMCY thread.	

Return: 0 or an error code described in the appendix.

canOpenRemoveNetwork()

Name: canOpenRemoveNetwork() - removing a network

Synopsis:

```
int canOpenRemoveNetwork
(
    int NetNo /* number of CAN interface */
)
```

Description: This routine removes the network object of net *NetNo* from the database.

Return: 0 or an error code described in the appendix.

canOpenCreateNode()

Name: canOpenCreateNode() - Initialize a CANopen node (**deprecated**).

Synopsis:

```
int canOpenCreateNode
(
    int NetNo, /* number of CAN interface */
    char * NodeName, /* name of slave node */
    int ModID, /* module number of node */
    unsigned long DevType, /* device type */
    int Options, /* default properties */
    int MaxErrors, /* size of error history */
    char * DeviceName, /* device name */
    char * HardwareVers, /* hardware-version number */
    char * SoftwareVers, /* software-version number */
    unsigned short GuardTime, /* default guardtime in ms */
    unsigned short LifeTime, /* default lifetime factor */
    unsigned short ServerObjects, /* number of additional SDO servers */
    unsigned short ClientObjects, /* number of additional SDO clients */
    int (* EventHandler)(int, int, int, int) /* event handler of CANopen node */
    HNDO * HNode /* handle of this CANopen node */
)
```

Description: Using this API is deprecated as improvements and extensions introduced with DS-301 V4.x can not be configured and the node event handler only supports a limited number of possible events. New applications should use *canOpenCreateNodeEx* instead. This API remains only for backward compatibility of existing applications.

This function generates a CANopen-node object with object directory for net *NetNo*. The entries **DeviceType** (0x1000) and **Error Register** (0x1001) required following */1/* as well as the optional entry **Node-ID** (0x100B) are automatically created in the object directory.

NodeName is a pointer to a textual description of the node with module number *ModID* in the range of 1 to 127. The module number determines the COB identifiers for the SDO server, the identifier for node guarding and the emergency object according to */1/*.

DevType is the device type which is returned after reading out directory entry 0x1000. The 16 LSB are the **Device Profile Number**, the MSB contain device- and/or profile-specific information.

The bitmask set in *options* determines the additional entries in the object directory and the validity of the following parameters.

Option	Meaning
BLOCK_TRANSFER	support of the SDO block transfer.
STATE_REGISTER	generate object entry 0x1002
ERROR_REGISTER	generate object entry 0x1003
ADDITIONAL_PDOS	generate object entry 0x1004
SYNCHRON_PDOS	generate object entries 0x1005-0x1007
MANUFACTURER_INFO	generate object entries 0x1008-0x100A
GUARDING	generate object entries 0x100C-0x100E
PARAMETER_STORE	generate object entry 0x100F
PARAMETER_RESET	generate object entry 0x1010
ADDITIONAL_SDOS	generate object entry 0x1011

If `BLOCK_TRANSFER` is set in *options*, the SDO server of the CANopen node support the SDO block transfer in addition to the standard SDO transfers.

If `State_REGISTER` is set in *options*, the entry for the state register in the object directory is generated.

If `ERROR_REGISTER` is set in *options*, *MaxErrors* determines the size of the error history.

If `SYNCHRON_PDOS` is set in *options*, the directory entries **COB-ID SYNC message** (0x1005), **communication cycle period** (0x1006) and **synchronous window length** (0x1007) are generated. The definition of synchronous PDOs is only possible, if this flag has been set.

If `MANUFACTURER_INFO` is set in *options*, it is possible to store the device name and the hardware and software versions in the object directory by means of *DeviceName*, *HardwareVers* and *SoftwareVers*.

The strings transferred have to be in a static area of the application and not on the stack, because the slave only refers to these areas by pointers.

If `GUARDING` is set in *options*, the node supports **life-** and **nodeguarding**. The default values for **guard time** and **life-time factor** can be defaulted by means of *GuardTime* and *LifeTime*.

If `ADDITIONAL_SDOS` is set in *options*, the number of additional SDO servers and SDO clients² can be determined in *ServerObjects* and *ClientObjects*. The default SDO server has to be included in *ServerObjects*.

² In the current version of the slaves it is not possible to generate additional SDO servers and SDO clients.

It is possible to connect a callback function by means of *EventHandlers*. If an event occurs, the code of this handler is executed. A detailed description of the callback handler can be taken from section 4.6.

If the returned value of the call is 0, the handle with which it is possible to access the node at further API calls is in *HNode*. If initialization was successful the node enters state **NodeOffline**.

Return: 0 or an error code described in the appendix.

canOpenCreateNodeEx()

Name: `canOpenCreateNodeEx()` - Extended initialization of a CANopen node.

Synopsis:

```
int canOpenCreateNodeEx
(
  int          iNetNo,          /* Number of logical CAN network */
  int          iModID,         /* Module number of node */
  int (* EventHandler)(SLAVE_EVENT *pEvent), /* Application event handler */
  SLAVE_NODE_INFO *pSlaveInfo, /* Ptr to node configuration */
  HND0 *      HNode           /* handle of this CANopen node */
)
```

Description: This API call initializes a CANopen node with the Node-ID *iModID* for the logical CAN net *iNetNo*. The caller determines the extend of “Communication Profile Area” objects */1/* and their default values with the pointer *pSlaveInfo* to an initialized structure of the type `SLAVE_NODE_INFO` which is described below. One member of this structure affects the kind of node events which are handled in the node event handler *EventHandlers*. A detailed description of the node events can be found in section 4.6.

If the API call returned without errors the node handle which is the argument for further API calls is stored at the memory location given by *Hnode*. After successful initialization the node enters the node state **NodeOffline**.

The structure `SLAVE_NODE_INFO` comprises all crucial information to describe extend and default values of the “Communication Profile Area” and other node specific configuration values. The complete structure should be filled with zeros before it is initialized..The basic idea of this structure is that the *ulOptions* member is a bitmask that defines which other members of the structure have to be initialized with proper values or can be left set to 0. The following table should provide an overview which flag in *ulOptions* causes which entry in the object dictionary to be created, which entries are created implicitly as CANopen */1/* defines them as mandatory and which other member variables in the `SLAVE_NODE_INFO` structure must be initialized. An index that is not listed in this table is either not supported or is reserved in */1/*.

Index	Name	Flag in <i>ulOptions</i>	Member to initialize
0x1000	Device Type	Created implicitly	<i>ulDeviceType</i>
0x1001	Error Register	Created implicitly	-
0x1002	Manufacturer Status	STATE_REGISTER	-
0x1003	Pre-defined error field	ERROR_REGISTER	<i>ucMaxErrors</i>
0x1005 to 0x1007	COB-ID SYNC, Comm. cycle period, Sync. Window length	SYNCHRON_PDOS SYNC_GENERATION	<i>ulSyncCobID</i> <i>ulCyclePeriod</i>

0x1008 to 0x100A	Manufacturer device name, HW version and SW version	MANUFACTURER_INFO	<i>pszDevicename, pszHwVersion, pszSwVersion</i>
0x100C to 0x100D	Guard time and Lifetime factor	GUARDING	<i>usGuardTime, ucLifeTime</i>
0x1010	Store parameters	PARAMETER_STORE	-
0x1011	Restore defaults	PARAMETER_RESET	-
0x1014	COB-ID EMCY	Created implicitly	<i>ulEmcyCobId,</i>
0x1015	Inhibit time EMCY	Created implicitly	<i>usEmcyInhibit</i>
0x1016	Consumer Heartbeat Time	CONSUMER_HEARTBEAT	<i>ucMaxConsumerHB, pulListCHBT</i>
0x1017	Producer Heartbeat	PRODUCER_HEARTBEAT	<i>usProducerHBTime</i>
0x1018	Identity Object	Created implicitly	<i>ucMaxIdentityObject, ulVendorId, ulProductCode, ulRevisionNumber, ulSerialNumber</i>
0x1020	Verify Configuration	PARAMETER_STORE	-
0x1028	Emergency consumer	EMCY_CONSUMER	
0x1029	Error behaviour	ERROR_BEHAVIOUR_OBJECT	<i>ucErrorBehaviour</i>
-	-	ADDITIONAL_SDOS	<i>ucServerSDO, ucClientSDO</i>
-	Support SDO block transfer.	BLOCK_TRANSFER	-

The following tables provide a description about every supported member in the SLAVE_NODE_INFO structure.

usRxPDO	Mandatory
Defines the maximum number of Rx-PDOs of this node.	

usTxPDO	Mandatory
Defines the maximum number of Tx-PDOs of this node.	

ucServerSDO	Mandatory if ADDITIONAL_SDOS is set
Defines the maximum number of SDO server. If ADDITIONAL_SDOS isn't set the default SDO server will be created.	

ucClientSDO	-
Reserved for future use	

ucMaxErrors	Mandatory if ERROR_REGISTER is set
Defines the maximum number of errors (1-127) that can be stored in the error history.	
ucMaxIdentityObject	Mandatory
Defines the number of subindices (1-4) of entry Identity Object (0x1018).	
ulVendorId	Mandatory
CiA registered vendor id for this device	
ulProductCode	Mandatory if ucMaxIdentityObject > 1
Vendor specific product code for this device	
ulRevisionNumber	Mandatory if ucMaxIdentityObject > 2
Vendor specific revision number for this device	
ulSerialNumber	Mandatory if ucMaxIdentityObject = 4
Vendor specific serial number for this device	
ucMaxConsumerHB	Mandatory if CONSUMER_HEARTBEAT is set
Number of subentries (1-127) of the Consumer Heartbeat object.	
ucLifetime	Mandatory if GUARDING is set
Default lifetime factor used by this device for life guarding.	
usGuardTime	Mandatory if GUARDING is set
Default guardtime used by this node for life guarding.	
ulDeviceType	Mandatory
Device type of this device	
pszDeviceName	Mandatory if MANUFACTURER_INFO is set
NULL terminated string for Manufacturer Info of this device	
pszHwVersion	Mandatory if MANUFACTURER_INFO is set
NULL terminated string for Manufacturer Hardware Version of this device.	
pszSwVersion	Mandatory if MANUFACTURER_INFO is set
NULL terminated string for Manufacturer Software Version of this device.	

ulSyncCobID	Mandatory if SYNCHRON_PDOS or SYNC_GENERATION is set
Defines the COB-ID of the SYNC object for this node as SYNC producer and/or SYNC consumer. Initialize to DEFAULT_SYNC_COBID which becomes 0x80 to use the standard /1/ default as SYNC consumer. To configure the node as SYNC generator you have to set the bit SYNC_PRODUCE, too.	
ulCyclePeriod	Mandatory if SYNC_GENERATION is set
Defines the cycle time of the SYNC object in us as SYNC generator. SYNC generation is only started if the SYNC_PRODUCE bit in <i>ulSyncCobID</i> is set and this value is not 0. Note: SYNC generation has to be supported by the CAN driver. Only CAN driver > V 3.x.x support this feature.	
ulSyncWindowLen	-
Reserved for future use	
ulTimestampCobId	-
Reserved for future use	
ulEmcyCobId	Optional
Defines the COB-ID of the EMCY object. If this is set to 0 or DEFAULT_EMCY_COBID the value becomes 0x80 + <i>iModId</i> is used.	
usEmcyInhibit	Optional
Defines the inhibit time for the EMCY object in ms. If this is set to 0 or DEFAULT_EMCY_INHIBIT_TIME there is no inhibit time to produce EMCY messages for the device.	
usProducerHbTime	Mandatory if PRODUCER_HERTBEAT is set
Defines the producer heartbeat time of this device in ms. If this is set to 0 or DEFAULT_PRODUCER_HEARTBEAT_TIME heartbeat is disabled on startup.	
*pulListCHBT	Mandatory if EMCY_CONSUMER is set
Defines the list of default emergency consumer entries. The argument is a pointer to an array of unsigned long values. Each entry has to be defined with the macro CHBT_ENTRY which takes two arguments. The first argument is the node number that is to be monitored, the second argument the heartbeat time in ms. The list has to be terminated with the entry END_OF_CHBT_LIST. The number of entries should shouldn't exceed the number of entries given with the parameter <i>ucMaxConsumerHB</i> . Example:	

ucErrorBehaviour	Mandatory if ERROR_BEHAVIOUR_OBJECT is set.
This parameter defines the default behaviour of the slave if an fatal error occurred. Possible values are: - ERROR_BEHAVIOUR_DEFAULT - Change to node state Pre-Operational - ERROR_BEHAVIOUR_NO_CHANGE - No change in node state. - ERROR_BEHAVIOUR_STOP - Change to node state STOPPED.	
ucMaxMapped	Conditional for <i>multimap</i> support.
This parameter defines in how many different PDOs the same object dictionary can be mapped if the this object dictionary entry is created supporting this feature. If this parameter is 0 the default value of 8 will be used.	
usPdoRxQueusize	Size of PDO daemon receive queue.
Defines the size of the Rx daemon receive queue in multiple of PDO messages. The default value is 256.	

Return: 0 or an error code described in the appendix.

canOpenDeleteNode()

Name: canOpenDeleteNode() - deleting a CANopen node

Synopsis:

```
int canOpenDeleteNode
(
    HNODE          HNode          /* handle of the CANopen node */
)
```

Description: Deletes a node object including the object directory and all COB identifiers used from this node from the internal database. Calling this function is only possible in node state **NodeOffline**.

Return: 0 or an error code described in the appendix.

canOpenActivateNode()

Name: canOpenActivateNode() - activating CANopen node.

Synopsis:

```
int canOpenActivateNode
(
    HNODE          HNode          /* handle of the CANopen node */
)
```

Description: Prepares the slave node for establishing connections. New node state is **PreOperational**. Node- and lifeguarding are active and accessing the object directory is possible.

Return: 0 or an error code described in the appendix.

canOpenGetNodeInfo()

Name: canOpenGetNodeInfo() - Return current node state

Synopsis:

```
int canOpenGetNodeInfo
(
    HNODE      HNode,      /* Handle of the CANopen node */
    int *      State,      /* Current node state */
    int *      LastErr     /* Last error state */
)
```

Description: This call returns the current state of the node referenced by *Hnode*.

Valid values for *State* are:

NodeInit	NodePreOperational
NodeStopped	NodeOperational

In *LastErr* the error number of the last error is returned.

Return: 0 or an error code described in the appendix.

canOpenResetNode()

Name: canOpenResetNode() - resetting CANopen node.

Synopsis:

```
int canOpenDeleteNode
(
    HNODE      HNode      /* handle of the CANopen node */
)
```

Description: The slave is reset to state **NodeOffline**. Nodeguarding and SDO-server processes are terminated. All used COB identifiers are freed and all entries in the object directory are reset to default values.

Return: 0 or an error code described in the appendix.

canOpenWaitForNodeState()

Name: `canOpenWaitForNodeState()` - Block until transision in given node state

Synopsis:

```
int canOpenWaitForNodeState
(
    HNODE          HNode, /* handle of the CANopen node */
    unsigned short StateMask /* state mask */
)
```

Description: The application is blocked until the node is in a determined state. It is possible to wait for one or more state.

StateMask is the logical OR combination of the following constants describing the node states to wait for. Valid parameters are:

```
WFNS_INIT          WFNS_STOPPED
WFNS_PRE_OPERATIONAL  WFNS_OPERATIONAL
```

Return: Current node status or an error code described in the appendix.

4.2 Local Object Directory Services

The services described in this section are used to extend the node's object dictionary by custom object entries as well as to provide read and write access to the local object dictionary.

Extending the object dictionary is only possible in the *Manufacturer Specific Area* (Index 0x2000 to 0x5FFF) and the *Standardized Device Profile Area* (Index 0x6000 - 0x9FFF). The application has full control about the object type, data type, access rights, default values, etc. The objects may be mapped into PDOs as described in the following chapter. The subindex 0xFF, which describes the structure of the object dictionary entry, is created automatically. An event handler can be assigned to every object.

External read access to the object dictionary entries by the CANopen manager or another slave on the CAN bus with an SDO service is processed asynchronously to the running application by the SDO server.

External write access to the object dictionary entries by the CANopen manager or another slave on the CAN bus with an SDO service is indicated to the application with the object event handler. The application can validate the data and prevent an update.

canOpenExtendDictionary()

Name: canOpenExtendDictionary() - Extending the local Object Dictionary

Synopsis:

```
int canOpenExtendDictionary
(
    HNODE           HNode,           /* Handle of the CANopen node */
    unsigned short  Index,          /* Index in object directory */
    unsigned short  Subentries,     /* Number of subentries */
    unsigned short  ObjectType,     /* Object type of entry */
    const char *    DataType,      /* Data type description */
)
```

Description: Extends the local object dictionary of the CANopen node with the node handle *HNode* in the **Manufacturer Specific Area** or the **Standardized Device Profile Area**. This function fails if called in another node state but **NodeOffline**.

Index is the index in the object directory in the range from 0x2000 to 0x9FFF and *Subentries* has to be set to the number of subentries of this entry in the range from 0-254.

ObjectType is either the simple data type `OBJ_VAR` or one of the complex data types `OBJ_ARRAY` or `OBJ_RECORD`. Simple data types only support sub-index 0.

DataType is a zero terminated descriptor array with only one entry for the data types `OBJ_VAR` and `OBJ_ARRAY`. For entries of the data type `OBJ_RECORD` the descriptor array contains the data type of every sub-index .

Dictionary entries of the data type `OBJ_ARRAY` and `OBJ_RECORD` store the number of sub-entries in the format **UNSIGNED8** as an RO entry at subindex 0. For arrays this entry is created automatically. For records this isn't the case for historical reasons, which means the application has to define the entry at subindex 0 as `TYP_UINT8` in the descriptor string to be compatible with the current revision of the specification /1/.

Example for a single value or array descriptor of data type `INTEGER32`:

```
const char *DescrSimple[] = {TYP_INT32, 0};
```

Example for a record descriptor of a `INTEGER32` at sub-index 1 and an `INTEGER16` at sub-index 2. The `UNSIGNED8` entry at sub-index 0 is also defined:

```
const char *DescrComplex[] = {TYP_UINT8, TYP_INT32, TYP_INT16, 0}
```

Return: 0 or an error code described in the appendix.

canOpenInitDictionary()

Name: `canOpenInitDictionary()` - Initialize local dictionary and attach handler

Synopsis:

```
int canOpenInitDictionary
(
    HNODE                               Hnode,           /* Node handle */
    unsigned short                      Index,           /* Index */
    unsigned short                      Subindex,        /* Subindex */
    const char *                        EntryName,       /* Textual description */
    unsigned short                      Flags,           /* Properties of entry */
    pDictionaryData                    Data,            /* Default data */
    PFN_COS_DATA_HANDLER                Handler         /* Data event handler */
)
```

Description: Initialize a single dictionary entry of the CANopen nodes *Hnode* object dictionary. The entry has to be created previously with a call to *canOpenExtendDictionary()*. This initialization has to take place for every entry in the object dictionary before the object can be used. If the function is called in any other node state but **NodeOffline** it will return with an error.

Index has to be in range of the **Manufacturer Specific Area** or the **Standardized Device Profile Area** (0x2000 to 0x9FFF) and *Subindex* in the range from 0x00 to 0xFE. The entry at sub-index 0xFF which describes the structure of this object dictionary entry according to /1/ is initialized implicitly. For complex data types of OBJ_ARRAY the sub-index 0 is initialized automatically to the number of sub-entries. For complex data types of OBJ_RECORD the sub-index 0 has to be initialized to the number of sub-entries by the application.

EntryName is an optional textual description of the entry. This description is only important for configuration file generation. Usually set this parameter to NULL, because this description extends the memory requirements for an individual subentry.

The parameter *flags* defines the access rights and other properties of the object dictionary entry. Supported values are READ_ACCESS and WRITE_ACCESS. To allow the mapping into a PDO the entry has to be marked as MAPPABLE. If the entry should be mappable more than once it has to be marked as MULTI_MAP.

Data is a pointer to a union of structures of type *DictionaryData*. The application has to initialize the data type related part of the union and is responsible. For numerical data types the structure has members for the current value, the default value and the lower and upper limits. The memory to keep these values is managed by the CANopen slave library.

For multibyte data types the structure has to be initialized with a pointer to an application defined memory region, the length of this memory range and the length of the current string.

Handler is the object event handler of this entry which is called by the CANopen slave library to indicate data changes to the application. Refer to chapter 4.6 for a detailed description of the data event handler.

Return: 0 or an error code described in the appendix.

canOpenInitDictionaryTS()

Name: canOpenInitDictionary() - Initialize dictionary and attach timestamp handler

Synopsis:

```
int canOpenInitDictionaryTs
(
    HNODE                Hnode,           /* Node handle */
    unsigned short       Index,          /* Index */
    unsigned short       Subindex,       /* Subindex */
    const char *         EntryName,      /* Textual description */
    unsigned short       Flags,          /* Properties of entry */
    pDictionaryData      Data,           /* Default data */
    PFN_COS_DATA_HANDLER_TS Handler     /* Data event handler */
)
```

Description: Initialize a single dictionary entry of the CANopen nodes *Hnode* object dictionary. The entry has to be created previously with a call to *canOpenExtendDictionary()*. This initialization has to take place for every entry in the object dictionary before the object can be used. If the function is called in any other node state but **NodeOffline** it will return with an error.

Index has to be in range of the **Manufacturer Specific Area** or the **Standardized Device Profile Area** (0x2000 to 0x9FFF) and *Subindex* in the range from 0x00 to 0xFE. The entry at sub-index 0xFF which describes the structure of this object dictionary entry according to /1/ is initialized implicitly. For complex data types of OBJ_ARRAY the sub-index 0 is initialized automatically to the number of sub-entries. For complex data types of OBJ_RECORD the sub-index 0 has to be initialized to the number of sub-entries by the application.

EntryName is an optional textual description of the entry. This description is only important for configuration file generation. Usually set this parameter to NULL, because this description extends the memory requirements for an individual subentry.

The parameter *flags* defines the access rights and other properties of the object dictionary entry. Supported values are READ_ACCESS and WRITE_ACCESS. To allow the mapping into a PDO the entry has to be marked as MAPPABLE. If the entry should be mappable more than once it has to be marked as MULTI_MAP.

Data is a pointer to a union of structures of type *DictionaryData*. The application has to initialize the data type related part of the union and is responsible. For numerical data types the structure has members for the current value, the default value and the lower and upper limits. The memory to keep

these values is managed by the CANopen slave library.

For multibyte data types the structure has to be initialized with a pointer to an application defined memory region, the length of this memory range and the length of the current string.

Handler is the object event handler of this entry which is called by the CANopen slave library to indicate data changes to the application. In comparison to *canOpenInitDictionary()* this handler indicates a timestamp in addition to the values which are indicated with the standard handler. Refer to chapter 4.6 for a detailed description of the data event handler.

Return: 0 or an error code described in the appendix.

canOpenReadDictionary()

Name: `canOpenReadDictionary()` - reading a local directory entry

Synopsis:

```
int canOpenReadDictionary
(
    HNODE          HNode, /* handle of the CANopen node */
    unsigned short Index, /* index in the object directory */
    unsigned short Subindex, /* subindex of entry */
    void *         Data /* pointer to data sink */
)
```

Description: This function reads an entry in the local object directory. It can be called in every node state.

Index is the index in the object directory and *subindex* is the subindex.

Data is a pointer to an application-memory area in which the data is stored. This memory range must have a size of at least 4 bytes. In numerical data *data* is a pointer to the data, in other data types it is a pointer to a pointer to the data.

Return: 0 or an error code described in the appendix.

canOpenWriteDictionary()

Name: canOpenWriteDictionary() - modifying a local directory entry

Synopsis:

```
int canOpenWriteDictionary
(
  HNODE           HNode,           /* handle of the CANopen node */
  unsigned short  Index,          /* index in the object directory */
  unsigned short  Subindex,       /* subindex of entry */
  void *          Data            /* pointer to data source */
)
```

Description: This function modifies an entry in the local object directory. If the entry is mapped into a PDO, the PDO data are automatically updated. It can be called in every node state.

Index shows the index in the object directory and *subindex* shows the subindex.

Data is a pointer to the new data in an application-memory area. Following table shows in which way data has to be provided by the application and the column Copy shows whether data is copied into the slave memory. If the values are not copied into the slave memory, like strings for instance, the pointers in the transferred structures have to refer to static memories, because they are being referenced at a read or write access by the slave.

CANopen data type	Reference type	Copy
Bool	Pointer to new data (1 byte)	yes
Int8, Int16, Int32	Pointer to new data (1 byte, 2 bytes, 4 bytes)	yes
UInt8, UInt16, UInt32	Pointer to new data (1 byte, 2 bytes, 4 bytes)	yes
Float	Pointer to new data (4 bytes)	yes
Visible String	Pointer to structure of RecVisString type	no
Octet String	Pointer to structure of RecOctString type	no
Time Of Day	Pointer to structure of CAN_TIME_OF_DAY type	yes
Time Difference	Pointer to structure of CAN_TIME_DIFFERENCE type	yes
Domain	Pointer to structure of RecDomain type	no

If data is NULL for asynchronous auto-notify PDO the current data would be transmitted.

Return: 0 or an error code described in the appendix.

canOpenGetDictionaryHnd()

Name: canOpenGetDictionaryHnd() - Handle of a local directory entry

Synopsis:

```
int canOpenGetDictionaryHnd
(
  HNODE          HNode, /* handle of the CANopen node */
  unsigned short Index, /* index in the object directory */
  unsigned short Subindex, /* subindex of the entry */
  HDICT *        HDict /* return of teh handle */
)
```

Description: This function returns a handle to an entry in the object directory. It can be executed in every node condition. By means of the calls **canOpenWriteDictionaryHnd()** and **canOpenReadDictionaryHnd()**, described below, you can gain both read and write access to this directory entry via this handle. This causes an increased performance compared to an access via index/subindex by **canOpenWriteDictionary()** or **canOpenReadDictionary()**, because the accroding entry does not have to be searched for in the interlinked directory entries. This is particularly of advantage in CANopen nodes with many entries in the local object directory.

Index specifies the index in the object directory, and *Subindex* specifies the subindex.

In *Hdict* the handle of the indexed directory entry is returned if the function returned faultless, otherwise a NULL is returned.

Return: 0 or an error code as described in the appendix.

canOpenReadDictionaryHnd()

Name: `canOpenReadDictionaryHnd()` - Reading a local directory entry

Synopsis:

```
int canOpenReadDictionary
(
    HDICT          HDict,          /* handle of object-directory entry */
    void *         Data           /* pointer to the data sink */
)
```

Description: By means of this function an entry, indexed by *Hdict*, in the local object directory is read. This function can be called in every node status.

Data is a pointer to an address range of the application in which data is stored. This memory range must have a capacity of at least 4 bytes. For numerical data *Data* is a pointer to the data, for other types of data it is a pointer to a pointer to the data.

Return: 0 or an error code as described in the appendix.

canOpenWriteDictionaryHnd()

Name: `canOpenWriteDictionaryHnd()` - Changing a local directory entry

Synopsis:

```
int canOpenWriteDictionaryHnd
(
    HDICT          HDict,          /* handle of object-directory entry */
    void *         Data           /* pointer to data source */
)
```

Description: By means of this function an entry, indexed by *Hdict*, in the local object directory is changed. If the entry is mapped on a PDO, the PDO data is automatically actualized. The function can be executed in every node status.

The kind of data referred to by *Data* depends on the according CANopen-variable type and is explained under `canOpenWritePDO`.

Return: 0 or an error code as described in the appendix.

4.3 PDO Services

Following services serve the definition of a *process data object (PDO)*, the determination of a *default mapping* of entries of the object directory into the PDO and the asynchronous transmission and reception of data.

For normal asynchronous transfer PDOs the transmission has to be explicitly arranged for by means of the application. The same goes for the waiting for new data or the request in asynchronous receive PDOs. In addition asynchronous PDOs can also be marked as *auto notify*, though, so that transfer PDOs are immediately transmitted when updating their data and that the eventhandler(s) of the mapped objects are executed when data for a Rx PDO is received.

The transmission of synchronous transfer PDOs is internally arranged for by means of the CANopen slave after receiving the SYNC object in view of the configured cycle period. The application only has to care about updating the data. The application is informed about received data after the reception of the SYNC object in view of the configured cycle period by means of calling the object eventhandlers of the mapped directory entries.

canOpenDefinePDO()

Name: canOpenDefinePDO() - initializing a PDO

Synopsis:

```
int canOpenDefinePDO
(
  HNODE          HNode,          /* handle of the CANopen node */
  const char *   Name,           /* designator of this PDO */
  UINT32         COBid,         /* default-COB identifier of PDO */
  UINT16         TransMode,     /* transfer mode of PDO */
  INT32          InhibitTime,   /* inhibit time of this PDO */
  UINT16         TxTout,       /* transmit timeout of this PDO */
  UINT16         RxTout,       /* receive timeout of this PDO */
  INT32          iEventTimer,   /* Event timer in ms of this PDO */
  UINT16 *       Mapping,      /* default mapping of this PDO */
  HPDO          hpdo           /* PDO handle */
)
```

Description: This function creates and initializes an additional PDO for the CANopen node. The total number of RPDOs/TPDOs, which is supported by this node instance, is a defined with canOpenCreateNodeEx(). The attempt to create more TPDOs/RPDOs results in an error. The PDO configuration after bootup or reset is defined by these passed configuration parameters. The node's object directory entries in the **PDO Communication Parameters** and the **PDO Mapping Parameters** area are generated implicitly. The position within the node's object directory is determined by the order of calls to this function in the application code.

Name is legacy parameter which is no longer supported and is ignored by the library. Always set this parameter to NULL.

The parameter *COBid* defines the PDO's default COB-ID which according to /1/ consists of the CAN-ID and additional control bits. To apply these control bits you have to combine them with the CAN-ID by a logical OR operation. To define the node's n-th default PDO you can use `DEFAULT_PDO_N` with $N=1..4$ for the CAN-ID instead using a numerical value. In this case the CAN-ID is derived from the Node-ID according to the pre-defined connection set /xxx/. The CAN-ID part of the COB-ID might be changed by a CANopen manager. The valid control bit can be set to `PDO_VALID` or `PDO_INVALID` to determine, which PDOs are used in the NMT node state **Operational**. The 4 default PDOs can always be set to valid. All additional PDOs should be set to invalid in order to prevent conflicts with other CANopen slave nodes. If a non-default PDO is initially set to valid the application is responsible for the CANopen network integrity. This COB-ID control bit might be changed by a CANopen manager. The RTR control bit `RTR_ALLOW` or `RTR_DISALLOW` define whether a transmit PDO might be RTR requestable or not³. The configured value of this COB-ID control bit can not be changed by a CANopen manager.

The parameter *TransMode* defines the transmission type of the PDO. In addition to /1/ this parameter consists also of several proprietary control bits which describe the type and the behavior of the PDO. To apply these control bits you have to combine them with the PDO transmission type by a logical OR operation. The PDO type control bit can be either set to `TRANSMIT_PDO` or `RECEIVE_PDO` with `SYNCHRON_PDO`/`ASYNCHRON_PDO` and a numerical value between 0 and 255 which is given in /1/ according to following table. In addition it is possible to mark an asynchron PDO by `AUTO_NOTIFY`. This PDO has the properties described above. For an asynchron transmit PDO it is possible to define via the flag `TX_DONE_PDO`, whether the CAN-driver function /2/ `canCalWrite` instead of `canCalSend` is used for the data transfer.

Value	PDO-transmission mode				
	cyclical	acyclical	synchronous	asynchronous	only RTR
0		x	x		
1-240	x		x		
241-251	reserved				
252			x		x
253				x	x
254 ⁴				x	
255 ⁵				x	

³ The number of Tx objects which automatically transmit the data at the reception of an RTR frame can possibly be limited by the CAN-controller hardware. Because this feature is also used by the node/life guarding mechanism, the number of Tx objects which support this is to be kept as small as possible.

⁴ The transmission of this PDO is triggered by means of a manufacturer-specific event.

⁵ The transmission of this PDO is triggered by means of a device-specific event.

A value of 0 describes a transfer PDO which is transmitted once at the reception of the SYNC object or a receive PDO whose data is taken over by the application at the reception of the SYNC object.

A value *n* between 1 and 240 describes a cyclical, synchronous transfer PDO which is only transmitted at the reception of every *n*th SYNC object.

InhibitTime determines in ms how long after transmission of this PDO this isn't allowed to be transmitted again.

TxTout and *RxTout* are the timeout intervals at transmission or reception of data.

The parameter *iEventTimer* defines the time in ms after which an asynchron TPDO is sent in either case even it's data hasn't changed.

The parameter *Mapping* describes the mapping of directory entries into the PDO by specifying index and subindex. The list has to be terminated by a zero for index and subindex. Dummy mapping according to /1/ is supported by specifying a value between 0x01 and 0x07 as index and a 0 as subindex.

In *Hpdo* the handle for this PDO is stored.

Return: 0 or an error code described in the appendix.

canOpenWritePDO()

Name: `canOpenWritePDO()` - asynchronous transmission of a transmit PDO

Synopsis:

```
int canOpenWritePDO
(
    HPDO          hpdo,          /* handle of PDO */
    void *        buffer        /* pointer to the data sink */
)
```

Description: Transmitting the asynchronous transfer PDO. This service is only possible in node state **Operational**.

If *buffer* is NULL, the PDO is transmitted as is. Otherwise the specified data is taken over and the updated PDO is transmitted. The corresponding mapping entries of the Object Dictionary are also updated by doing this.

Return: 0 or an error code described in the appendix.

canOpenReadPDO()

Name: canOpenReadPDO() - waiting for the reception of data

Synopsis:

```
int canOpenReadPDO
(
    HPDO          hpdo,          /* handle of PDO */
    void *        buffer        /* pointer to the data sink */
)
```

Description: The application waits for the reception of data for a given PDO. The timeout values assigned in the PDO definition are valid.

Buffer is a pointer to an application memory area (at least 8 bytes) in which the received data can be stored. If NULL the callback handler of the mapped directory entries are called, otherwise this is suppressed.

Return: 0 or an error code described in the appendix.

canOpenRequestPDO()

Name: canOpenRequestPDO() - asynchronous request of data

Synopsis:

```
int canOpenRequestPDO
(
    HPDO          hpdo,          /* handle of PDO */
    void *        buffer        /* pointer to the data sink */
)
```

Description: By means of this function a client requests the transmission of a server PDO by means of a RTR frame. The timeout values assigned in the PDO definition are valid.

Buffer is a pointer to an application-memory range (at least 8 bytes) in which the received data can be stored. If NULL the callback handlers of the mapped directory entries are called, otherwise this is suppressed.

Return: 0 or an error code described in the appendix.

4.4 Error Situations and Emergency (EMCY) Objects

The CANopen slave implements an error state machine which can be either in the state *Error-Free* or in the state *Error*. A state change can be caused by the application layer using the API described in this chapter or is caused internally if a communication or configuration error situation is detected or resolved. The picture below describes the possible transitions between the error-free and the error state:

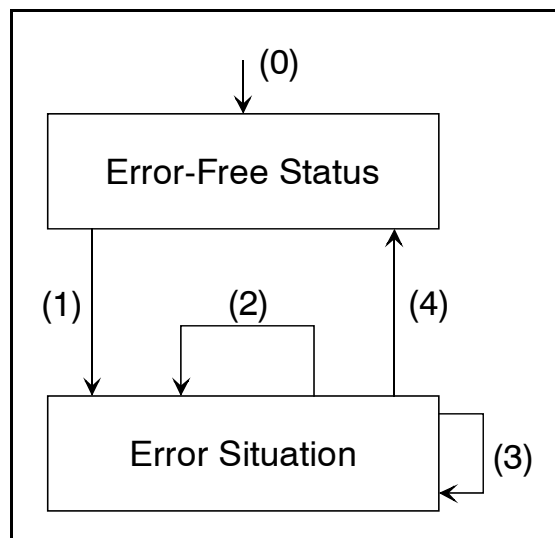


Fig. 3: Error State Transition Diagram

- (0) After calling **canOpenActivateNode()** the CANopen node gets into error-free state.
- (1) If **canOpenSetError()** is called by the application or an internal communication or configuration error is detected, the error is indicated as described in the following abstract and the CANopen node changes into the error state. The internal node error counter is incremented .
- (2) If **canOpenSetError()** is called again, the previous tasks are repeated and the CANopen node remains in the error state.
- (3) If **canOpenResetError()** is called by the application or an internal communication or configuration error condition is solved, the error is indicated as described in the following abstract and the node's internal error counter is decremented. As long as the counter doesn't reach 0 the CANopen node remains in the error state.
- (4) If the internal error counter becomes 0 during the previous step the node changes back into the error-free state.

An error situation or an repaired error is indicated to the application layer and on the CAN bus in the following ways:

- The error is indicated in the mandatory Error Register Object (0x1001) according to /1/.
- An Emergency (EMCY) Object according to /1/ is transmitted on the CAN-Bus. The details of the EMCY object is described below. The CAN identifier of this object can be configured with the parameter `ulEmcyCobId` of the structure `SLAVE_NODE_INFO` which is described together with `canOpenCreateNodeEx()`.
- If the slave is initialized to support an error history via the Pre-defined error field (0x1003), the latest error event is inserted at the top of this array.
- If the error is detected internally because of a communication or configuration problem in addition to the previous operations the error is indicated to the application via the node's event handler.

The 8 byte EMCY object according to /1/ has the following structure:

2 Bytes	1 Byte	5 Byte
Emergency Error Code	Error Register	Manufacture-specific error code

A list of pre-defined Emergency Error Codes is described in /1/ and defined in the header `scanopen.h` starting with the prefix `EMCY`. If an error event is caused by the application calling the API functions `canOpenSetError()` or `canOpenResetError()` the 5 bytes of manufacturer-specific error information can be used without any restrictions. If the EMCY object is transmitted because of an internal communication or configuration error the 5 bytes are used in the following way:

Temporary Bits	Sticky Bits	Reason	Info1	Info2
----------------	-------------	--------	-------	-------

The *Temporary Bits* indicate temporary error conditions which are reset if the error is repaired:

Bit	Description
0	NMT Error Control (Guarding/Heartbeat error).
1	CAN Controller Passive.
2	CAN Controller Bus Off
3-4	Reserved for future use by the CANopen stack.
5-7	Application specific temporary error.

The *Sticky Bits* indicate error conditions which are indicated even if the error is already repaired.

Bit	Description
0	CAN Controller Error.
1	Receive FIFO Overrun.
2	PDO Length Error.
3-4	Reserved for future use by the CANopen stack.
5-7	Application specific temporary error.

The parameter *Reason*, *Info1* and *Info2* contain additional information to an internal generated EMCY object because of a communication or configuration error. The table below lists the internally generated EMCY objects and the meaning of the related manufacturer-specific parameter.

Error Code	Description	Reason	Info1	Info2
0x8100	Message Lost Error	1 = Rx Daemon FIFO	# of Lost Messages	0
0x8110	CAN overrun (objects lost)	0	0	0
0x8120	CAN in error passive mode	0	0	0
0x8130	Life Guard or Heartbeat Error	0	0	0
0x8140	Recover from Bus-Off	0	0	0
0x8210	PDO not processed (length error)	Internal PDO number	CAN msg length	PDO length
0x8220	PDO length exceeded	Internal PDO number	CAN msg length	PDO length

The application can participate using this error schema by setting

canOpenSetError()

Name: canOpenSetError() - setting an error

Synopsis:

```
int canOpenSetError
(
    HNODE          Hnode,          /* handle of the CANopen node */
    unsigned short ErrorCode,      /* error code according to /1/ */
    unsigned short ErrorInformation, /* error information */
    unsigned short ErrorRegister,  /* flags in error register */
    unsigned char * ErrorField     /* pointer to error field */
)
```

Description: The CANopen slave changes from error-free state into error state.

In *ErrorCode* the **Emergency Error Code** of the EMCY object is determined. This EMCY object consists of an application specific error code in the range of 0x00 - 0xFF. This error code has to be connected to one of the following CANopen-error codes:

EMCY_GENERIC_ERROR	EMCY_CURRENT
EMCY_CURRENT_INPUT	EMCY_CURRENT_INSIDE
EMCY_CURRENT_OUTPUT	EMCY_VOLTAGE
EMCY_VOLTAGE_INPUT	EMCY_VOLTAGE_INSIDE
EMCY_VOLTAGE_OUTPUT	EMCY_TEMPERATURE
EMCY_TEMPERATURE_AMBIENT	EMCY_TEMPERATURE_DEVICE
EMCY_DEVICE_HARDWARE	EMCY_DEVICE_SOFTWARE
EMCY_DEVICE_SOFTWARE_INTERNAL	EMCY_DEVICE_SOFTWARE_USER
EMCY_DEVICE_SOFTWARE_DATA_SET	EMCY_ADDITIONAL_MODULES
EMCY_MONITORING	EMCY_MONITORING_COMMUNICATION
EMCY_EXTERNAL_ERROR	EMCY_ADDITIONAL_FUNCTIONS
EMCY_DEVICE_SPECIFIC	

ErrorInformation determines the information to be stored in the two MS bytes of the error history under directory entry 0x1003. If this optional entry has not been made when initializing the CANopen node, *ErrorInformation* is ignored.

A mask with flags is given as *ErrorRegister*. These flags are to be set in the error register (directory entry 0x1001). The mask is logically OR'ed to the current value of the entry, before the value is entered into the EMCY object. Possible values are:

ERROR_GENERIC	ERROR_CURRENT
ERROR_VOLTAGE	ERROR_TEMPERATURE
ERROR_COMMUNICATION	ERROR_DEVICE_SPECIFIC
ERROR_MANUFACTURER_SPECIFIC	

ErrorField is a pointer to a 5-byte string which contains an application-specific description of the error and is transmitted by means of the EMCY object.

Return: 0 or an error code described in the appendix.

canOpenResetError()

Name: canOpenResetError() - resetting an error

Synopsis:

```
int canOpenWritePDO
(
    HNODE          Hnode,          /* handle of the CANopen node */
    unsigned short ErrorRegister,  /* flags in error register */
    unsigned char * ErrorField    /* pointer to error field */
)
```

Description: An error of the CANopen slave is reseted. An EMCY object with **ErrorReset** in the error-code field is transmitted. If this was the last error, the node changes from error state into error-free state.

ErrorRegister is a mask of flags to reset in the error register (directory entry 0x1001). Possible values are:

```
ERROR_GENERIC          ERROR_CURRENT
ERROR_VOLTAGE         ERROR_TEMPERATURE
ERROR_COMMUNICATION    ERROR_DEVICE_SPECIFIC
ERROR_MANUFACTURER_SPECIFIC
```

ErrorField is a pointer to a 5-byte-long character chain which contains an application-specific state description and is transmitted by means of the EMCY object.

Return: 0 or an error code described in the appendix.

4.5 Assistant Functions

canOpenGetVersions()

Name: canOpenGetVersions() - Return version of slave components.

Synopsis:

```
void canOpenGetVersions
(
    CANOPEN_VERSIONS    *versions    /* pointer to version structure */
)
```

Description: This function returns the version numbers of the components described in the introduction.

A pointer to the data structure below which is initialized by the CANopen slave library.

```
typedef struct
{
    unsigned short cos;
    unsigned short sdm;
    unsigned short pdm;
    unsigned short nmt;
    unsigned short dbt;
    unsigned short cms;
    unsigned short sys;
    unsigned short can;
} CANOPEN_VERSIONS;
```

The revision number of each component is a 16-bit value with the following format:

Bits 15...12	Bits 11...8	Bits 7...0
level	revision	change

Return: N/A.

canOpenSetParameter()

Name: `canOpenSetParameter()` - Configure parameter of CANopen stack

Synopsis:

```
int canOpenSetParameter
(
    HNODE    hNode,        /* Node handle    */
    UINT32   uiCommand,   /* Command       */
    VOID     *pArg        /* Argument      */
)
```

Description: Configure the behavior of the CANopen stack or a single node at runtime. The argument type depends on the command according to this table:

PARAM_DISABLE_AUTO_TRANSMISSION:

Disable the automatic transmission of objects mapped into PDO which are configured as asynchron PDOs and marked with the `AUTO_NOTIFY` bit. The argument has to be set to `NULL`. This call allows the application to update all objects of a PDO without forcing a transmission of after each update.

PARAM_ENABLE_AUTO_TRANSMISSION:

Enable the automatic transmission of objects mapped into PDO which are configured as asynchron PDOs and marked with the `AUTO_NOTIFY` bit. The argument has to be set to `NULL`. All asynchron PDOs with mapped objects which are updated since the call to `canOpenSetParameter()` with the command `PARAM_DISABLE_AUTO_TRANSMISSION` are send immediately.

Return: 0 or an error code described in the appendix.

canOpenGetParameter()

Name: `canOpenGetParameter()` - Get a parameter from the CANopen stack

Synopsis:

```
int canOpenGetParameter
(
    HNODE    hNode,        /* Node handle    */
    UINT32   uiCommand,   /* Command       */
    VOID     *pArg        /* Argument      */
)
```

Description: Get a configuration parameter of the CANopen stack or a single node at runtime. The argument type depends on the command according to this table:

PARA_GET_TIMESTAMP_FREQUENCY:

Returns the frequency of the timestamp counter (if supported by the CAN hardware and/or CAN driver) of the physical CAN port the CANopen node is using to send and receive messages. The data is returned as an UINT64 value.

PARA_GET_TIMESTAMP:

Returns the current value of the timestamp counter (if supported by the CAN hardware and/or CAN driver) of the physical CAN port the CANopen node is using to send and receive messages. The data is returned as an UINT64 value.

Return: 0 or an error code described in the appendix.

4.6 Event handler

The base for the event driven interaction between the CANopen slave library and the application are event handler (direct callbacks because of performance). Each node has an event handler to indicate node specific events or error situations to the application. To every entry in the object dictionary an event handler can be attached which is called by the CANopen library if the data of the object is changed or the application is requested to provide new data for this object.

Every event handler is called directly from within the threads/processes of the CANopen slave library. For this reason the handler should be programmed thread-safe, should reduce the execution time to a minimum and is never allowed to use blocking calls.

Object Eventhandler without timestamps

If the data of an entry in the object directory is changed by an external PDO or SDO service or the application is requested to update the data, the attached object event handler is called with the five arguments below:

1. Net number (*int*)
2. Module number (*int*)
3. Index (*int*)
4. Subindex (*int*)
5. Pointer to received data (*void **)

The pointer to the data gets invalid after return from the event handler. It is possible to define the same event handler for all nets, nodes and dictionary entries and dispatch the first 4 parameter to relate data to the object.

Object Eventhandler with timestamps

If the data of an entry in the object directory is changed by an external PDO or SDO service or the application is requested to update the data, the attached object event handler is called with the five arguments below:

1. Net number (*int*)
2. Module number (*int*)
3. Index (*int*)
4. Subindex (*int*)
5. Pointer to received data (*void **)
6. Timestamp (*UINT64*)

The pointer to the data gets invalid after return from the event handler. It is possible to define the same event handler for all nets, nodes and dictionary entries and dispatch the first 4 parameter to relate data to the object.

The timestamp is captured with the reception of the PDO or at the end of an SDO service. It is either a raw value which has to be normalized by the application or the CANopen stack can be configured to normalize the timestamps to us with *canOpenCreateNetworkEx()*.

Node Eventhandler

The node event handler that is defined in *canOpenCreateNodeEx()* is called every time the CANopen slave has to indicate an event or error to the application. The application can define an event mask with events that are to be indicated using the parameter *ulEventMask* of the structure `SLAVE_NODE_INFO` which is a parameter of *canOpenCreateNodeEx()*.

The event handler itself has to follow the syntax:

```
int EventHandler(SLAVE_EVENT *pEvent);
```

The handler should always return `SCANOPEN_OK` and shouldn't block. The argument of the event handler is a pointer to the following structure:

```
typedef struct {
    unsigned short  usNetNo;
    unsigned short  usModId;
    unsigned long   ulEvent;
    unsigned long   ulArg1;
    union {
        unsigned long ulArg2;
        void *        pArg2;
    } arg;
} SLAVE_EVENT;
```

The member variable *usNetNo* and *usModId* describe the logical net number and the local slave Node-ID of the event source, so a common event handler might be used for all local slaves on all configured networks. The member variable *ulEvent* is the event type. The event types are the same that are used to define the event mask in the structure `SLAVE_NODE_INFO` mentioned above. The argument *ulArg1* is the first subargument of the event type. The second subargument is either another decimal or a pointer to a data structure whose type depends on the main event type.

The following table summarizes the possible event types with their subarguments.

ulEvent	ulArg1	ulArg2/pArg2	Event reason
EV_GUARDING	EV_START	---	Node/Lifeguarding is started
	EV_TIMEOUT	---	Lifeguarding timed out
	EV_STOP	---	Node/Lifeguarding is stopped
EV_STATE_CHANGE	<i>state</i>	---	node has changed into <i>state</i>
EV_RESET	EV_COMMUNICATION	EV_BEGIN	Enter <i>Reset Communication</i> state
	EV_COMMUNICATION	EV_END	Leave <i>Reset Communication</i> state
	EV_APPLICATION	EV_BEGIN	Enter <i>Reset Application</i> started state
	EV_APPLICATION	EV_BEGIN	Leave <i>Reset Application</i> state
EV_CONFIGURATION	EV_STORE	-	Store configuration request
	EV_RESTORE	-	Restore configuration request

ulEvent	ulArg1	ulArg2/pArg2	Event reason
EV_CAN	EV_CONTROLLER_OK	-	Recovery from CAN bus-off state
	EV_CONTROLLER_WARN	-	CAN Controller enters error passive state
	EV_CONTROLLER_BUS_OFF	-	CAN Controller enters bus-off state
	EV_FIFO_OVERRUN	-	CAN controller overrun error
	EV_PDO_FIFO_OVERRUN	Number of lost PDOs	Receive FIFO of PDO daemon is overrun
	EV_PDO_RX_ERROR	CAN driver error code	The receive request of the PDO daemon returned with an unexpected error.
	EV_PDO_NOT_PROCESSED	PDO number	A received PDO isn't processed because the length of the received PDO is smaller than the length according to the current mapping.
	EV_PDO_LENGTH_EXCEEDED	PDO number	A received PDO isn't processed because the length of the received PDO exceeds the length according to the current mapping for this PDO.
EV_EMCY	Node-ID of the EMCY producer	Ptr to EMCY object	EMCY object received
EV_CONSUMER_HEARTBEAT	EV_START	---	Heartbeat monitoring started
	EV_STOP	---	Heartbeat monitoring stopped
	EV_BOOTUP	---	Bootup message received
EV_WRITE_DICTIONARY	Index	Subindex	Write access to object dictionary entry

Deprecated event handler

If the CANopen node is created with the deprecated API `canOpenCreateNode()` an event handler of the following syntax is called:

```
int EventHandler(int, int, int, int);
```

The handler should always return `SCANOPEN_OK` and shouldn't block. The four parameter that are indicated to the application are:

1. Net number and module number
2. Event cause (as `ulEvent` described above)
3. Subargument 1 (as `ulArg1` described above)
4. Subargument 2 (as `ulArg2` described above)

Only the event types `EV_GUARDING`, `EV_STATE_CHANGE` and `EV_RESET` are supported. With the help of the macros `CANOPEN_NET` and `CANOPEN_NODE` the logical net number and node number can be extracted from the first parameter which combines these two values.

4.7 Macros

The CANopen slave library comes with a set of several useful macros which simplify common programming tasks and make the code more readable.

The base concept of several macros is implementing a static table with entries in the local scope of your source module to define the CANopen slave related objects (Object dictionary entries, PDO mapping tables, PDOs). After creating the CANopen slave node with *canOpenCreateNodeEx()* and before activating the node with *canOpenActivateNode()* you write a further macro directly in your code which expands to the API calls which are usually create and/or initialize these objects, processing the defined table. As these macros simply expand to the standard API calls, a mixed usage of macros and API calls in the code to setup and initialize the CANopen slave node is possible.

Dictionary Entry Tables

The following macros are used in lieu of repetitive calls to *canOpenExtendDictionary()* and *canOpenInitDictionary()* or *canOpenInitDictionaryTs()*.

```
BEGIN_DICTIONARY_TABLE(DictionaryName)
```

Begins the definition of a dictionary table for object dictionary entries with attached handlers without timestamps. You can define more than one dictionary table defining different values for *DictionaryName*. You have to define the dictionary table either in the local scope of your source module or in the local scope of code that is implementing `DECLARE_DICTIONARY`.

```
END_DICTIONARY_TABLE
```

Ends the definition of a dictionary table for object dictionary entries with attached handlers without timestamps..

```
BEGIN_DICTIONARY_TABLE_TS(DictionaryName)
```

Begins the definition of a dictionary table for object dictionary entries with attached handlers with timestamps. You can define more than one dictionary table defining different values for *DictionaryName*. You have to define the dictionary table either in the local scope of your source module or in the local scope of code that is implementing `DECLARE_DICTIONARY`.

```
END_DICTIONARY_TABLE_TS
```

Ends the definition of a dictionary table for object dictionary entries with attached handlers with timestamps.

```
DICTIONARY_ENTRY(Index, Subindex, ObjectType, DataType,  
                 Flags, Data, Handler, EntryName)
```

Defines a new dictionary entry. Please refer to the documentation of *canOpenExtendDictionary()* for the data types and possible values of *Index*, *Subindex*, *ObjectType* and *DataType*. Please refer to the documentation of *canOpenInitDictionary()* for the data types and possible values of *Flags*, *Data*, *Handler* and *EntryName*.

The parameter *EntryName* isn't supported at the moment and has to be set to NULL. If dictionary entries of the object type `OBJ_ARRAY` or `OBJ_RECORD` are defined, the read only dictionary entry for subindex 0 with data type `Uint8` initialized to the number of subentries is created implicitly.

One disadvantage of using macros extending and initializing the object dictionary is that for the object type `OBJ_ARRAY` and `OBJ_RECORD` the individual subentries can not be initialized to different default values, access attributes or handler as they all get initialized with the same parameters. If you want to force individual values, you can override the initialization performed by the macro with required calls of *canOpenInitDictionary()* after `DECLARE_DICTIONARY` and before *canOpenActivateNode()* is called.

```
DECLARE_DICTIONARY(hNode, DictionaryName)
```

Extends and initialize the slave node with a previously defined dictionary table. The parameter *hNode* is the node handle which is returned by *canOpenCreateNodeEx()*. The parameter *DictionaryName* defines the dictionary table which is started with `BEGIN_DICTIONARY_TABLE..`

Internally these macros define and use arrays of the type `_COS_DICT_ENTRY` with the variable name *DictionaryName* prefixed by the string “*_Dict_Entry_*” which do not need accessed directly by the application. At the end of the explanation of the PDO Table related macros you will find an example using these macros.

PDO Mapping Tables

The following macros are used to define a default mapping of entries in the object dictionary to the PDO of the slave node. Their only purpose is to provide the possibility of a more clearly laid out code for the mapping data structure which is referenced in *canOpenDefinePDO()*.

```
BEGIN_MAPPING_TABLE (MappingName)
```

Begins the definition of a PDO mapping table. You can define more than one PDO mapping defining different values for *MappingName*. You have to define the PDO mapping table either in the local scope of your source module or in the local scope of code that is implementing `DECLARE_PDO`.

```
END_MAPPING_TABLE
```

Ends the definition of a PDO mapping table.

```
MAPPING_ENTRY (Index, Subindex)
```

Defines a new mapping entry. Please refer to the documentation of the parameter *Mapping* for *canOpenDefinePDO()* for more details about the macro parameter *Index* and *Subindex*.

Internally these macros define arrays of the type `unsigned short` with the variable name *MappingName* prefixed by the string “*_Mapping_*” which do not need accessed directly by the application. At the end of the explanation of the PDO Table related macros you will find an example using these macros.

PDO Tables

The following macros are used in lieu of repetitive calls to *canOpenDefinePDO()*

`BEGIN_PDO_TABLE (PDO_Name)`

Begins the definition of a table with PDO descriptions. You can define more than one PDO table defining different values for *PDO_Name*. You have to define the PDO table either in the local scope of your source module or in the local scope of code that is implementing `DECLARE_PDO`.

`END_PDO_TABLE`

Ends the definition of a PDO table.

`PDO_ENTRY (COBid, TransMode, InhibitTime, TxTout, RxTout, Reserved, Mapping)`

Defines a new PDO entry with a default mapping. Please refer to the documentation of *canOpenDefinePDO()* for the data types and possible values of *COBid*, *TransMode*, *InhibitTime*, *TxTout*, and *RxTout*. Use the parameter *MappingName* of the macro `BEGIN_MAPPING_TABLE` of the intended default mapping as argument for this macro parameter *Mapping*.

`PDO_ENTRY_UNMAPPED (COBid, TransMode, InhibitTime, TxTout, RxTout, Reserved)`

Defines a new PDO entry without a default mapping. Please refer to the documentation of *canOpenDefinePDO()* for the data types and possible values of *COBid*, *TransMode*, *InhibitTime*, *TxTout*, and *RxTout*.

`DECLARE_PDO (hNode, PDO_Name)`

Extends and initialize the slave node with a previously defined PDO table. The parameter *hNode* is the node handle which is returned by `canOpenCreateNodeEx()`. The parameter *PDO_Name* defines the dictionary table which is started with `BEGIN_PDO_TABLE..`

Internally these macros define and use arrays of the type `_COS_PDO_ENTRY` with the variable name *PDO_Name* prefixed by the string “*_PDO_Table_*” which normally do not need accessed directly by the application. If the application needs the PDO handle which is returned by `canOpenDefinePDO()` to use direct PDO services for reading or writing PDOs this handle is stored in member *handle* of the structure `_COS_PDO_ENTRY`.

The following example shows how to create dictionary tables, mapping tables and PDO tables using the macros described above. This code is usually located in the module that implements initialization and setup of the CANopen slave node:

```
#include <scanopen.h>

/* Forward declarations */
static DictionaryData udtDefaultData;

int DataEventHandler(int NetNo, int NodeNo, int index, int subindex,
                    void *data);

/* Defines */
#define WRITE_STATE_32_OUTPUT_LINES 0x6320
#define READ_INPUT_32_BIT 0x6120

/* Definition of local Object Dictionary */
BEGIN_DICTIONARY_TABLE(AsyncIo)
    DICTIONARY_ENTRY(WRITE_STATE_32_OUTPUT_LINES, 2, OBJ_ARRAY,
                    MAP_UINT32, MAPPABLE | READ_ACCESS | WRITE_ACCESS,
                    &udtDefaultData, DataEventHandler, NULL)
    DICTIONARY_ENTRY(READ_INPUT_32_BIT, 2, OBJ_ARRAY,
                    MAP_UINT32, MAPPABLE | READ_ACCESS,
                    &udtDefaultData, DataEventHandler, NULL)
END_DICTIONARY_TABLE()

/* Definition of Default Mapping Table of PDOs. */
BEGIN_MAPPING_TABLE(OutputMapping1)
    MAPPING_ENTRY(WRITE_STATE_32_OUTPUT_LINES, 1)
    MAPPING_ENTRY(WRITE_STATE_32_OUTPUT_LINES, 2)
END_MAPPING_TABLE()

BEGIN_MAPPING_TABLE(InputMapping1)
    MAPPING_ENTRY(READ_INPUT_32_BIT, 1)
    MAPPING_ENTRY(READ_INPUT_32_BIT, 2)
END_MAPPING_TABLE()

/* Definition of PDOs.*/
BEGIN_PDO_TABLE(AsyncIo)
    PDO_ENTRY(DEFAULT_PDO1,
              RECEIVE_PDO | ASYNCHRON_PDO | AUTO_NOTIFY_PDO | 255,
              0, 5000, 5000, 0, OutputMapping1)
    PDO_ENTRY(DEFAULT_PDO1,
              TIMER_DRIVEN_PDO | TRANSMIT_PDO | AUTO_NOTIFY_PDO | 255,
              0, 5000, 5000, 0, InputMapping1)
END_PDO_TABLE()
```

The following example shows some pseudo code how to setup the object dictionary and PDOs using the definition in the previous example. Please refer to example1.c, which comes with your CANopen library distribution, for a fully working example.

```
HNODE Node;          /* Node handle */

/* Create slave node */
canOpenCreateNodeEx(..., &Node);
                .
                .
                .

/* Initialize object default data and create application specific */
udtDefaultData.uint32.defval = 0;
udtDefaultData.uint32.val    = 0;

/* Create the dictionary */
DECLARE_DICTIONARY(Node, AsyncIo);

/* Declare PDOs */
DECLARE_PDO(Node, AsyncIo);
                .
                .
                .

/* Activate slave and enter state state Pre-Operational */
```

5. Error Codes of Slave-Service Functions

The following tables list the possible error codes that can be returned by the slave library API calls. Some error codes defined in the header of the slave library are only used internally. As they wwon't be returned by any API call they are not documented here.

When evaluating return values you should never use the numerical values but should always use the constants defined for this error codes.

SCANOPEN_OK

Success (no warning or error).

Severity	Success
Description	The operation was executed without any errors.
Function	All functions.

SCANOPEN_WRONG_INDEX

The parameter *index* is invalid.

Severity	Error
Description	The object entry that should be referenced by the parameter <i>index</i> does not exist.
Solutions	Create an entry in the object dictionary with this index before you reference it.
Function	canOpenInitDictionary() canOpenReadDictionary() canOpenWriteDictionary()

SCANOPEN_WRONG_SUBINDEX

The parameter *subindex* is invalid.

Severity	Error
Description	The object entry that should be referenced by the parameter <i>index</i> exist but the subindex does not exist.
Solutions	Create an entry in the object dictionary with this index and subindex before you reference it.
Function	canOpenInitDictionary() canOpenReadDictionary() canOpenWriteDictionary()

SCANOPEN_OUT_OF_MEMORY

Error allocating a resource.

Severity	Error
Description	Allocating a resource like memory or a synchronization object that is necessary to complete the operation failed.
Solutions	Increase the available memory for the CANopen slave process.
Function	canOpenActivateNode() canOpenCreateNetwork() canOpenCreateNode() canOpenDefinePDO() canOpenExtendDictionary()

SCANOPEN_WRONG_BAUDRATE

An unsupported CAN baudrate was used.

Severity	Error
Description	The CANopen slave should be initialized with a CAN baudrate that is unsupported by this implementation.
Solutions	Use a supported baudrate.
Function	canOpenCreateNetwork()

SCANOPEN_CANNOT_START_DAEMON

Error creating an internal thread.

Severity	Error
Description	During CANopen slave initialization a necessary internal CANopen protocol thread could not be started.
Solutions	<ul style="list-style-type: none"> • Increase the available memory for the CANopen slave process.. • Make sure that the CAN driver is started properly
Function	canOpenCreateNetwork() canOpenCreateNode() canOpenActivateNode()

SCANOPEN_WRONG_PARAMETER

Invalid parameter.

Severity	Error
Description	One or more parameter of a function call were invalid.
Solutions	Compare parameter value with ranges given in manual
Function	All functions

SCANOPEN_VALUE_TOO_HIGH

Parameter value exceeds maximum.

Severity	Error
Description	A dictionary object value exceeds the given maximum for this entry.
Solutions	Compare value with defined maximum of this entry
Function	canOpenWriteDictionary() canOpenWriteDictionaryHnd()

SCANOPEN_VALUE_TOO_LOW

Parameter value below minimum.

Severity	Error
Description	A dictionary object value is below the given minimum for this entry.
Solutions	Compare value with defined minimum of this entry
Function	canOpenWriteDictionary() canOpenWriteDictionaryHnd()

SCANOPEN_WRONG_TYPE

Wrong data type.

Severity	Error
Description	The data type is not supported by the CANopen slave or the given data type does not match the referenced entry of the object dictionary.
Solutions	<ul style="list-style-type: none">• Use supported data types listed in manual.• Check defined data type for this object dictionary entry.
Function	canOpenExtendDictionary() canOpenReadDictionary() canOpenReadDictionaryHnd()

SCANOPEN_WRONG_OBJECT_TYPE

Wrong object type.

Severity	Error
Description	The object type is not supported by the CANopen slave.
Solutions	Use supported object types listed in manual.
Function	canOpenExtendDictionary()

SCANOPEN_PDO_MAPPING_ERROR

An error occurred during PDO mapping .

Severity	Error
Description	An error occurred while the default mapping list for a PDO is checked. Reasons for the failures are that an object dictionary entry referenced by index/subindex does not exist, is not mappable, has wrong access rights or is already mapped to another PDO.
Solutions	<ul style="list-style-type: none">• Check if an object with this index/subindex exist.• Check if this object is marked as mappable• Check is the access rights are correct for the PDO type.• Check if this object is not already mapped to another PDO.
Function	canOpenDefinePDO()

SCANOPEN_TOO_MANY_OBJECTS

A certain object type exceeds internal limits.

Severity	Error
Description	During initialization the built in maximum for a certain internal object type like number of CANopen nodes is exceeded
Solutions	<p>If this is the return value of <code>canOpenDefinePDO()</code> check if one of the following error conditions is met:</p> <ul style="list-style-type: none">• The number of created RPDOs/TPDOs exceed the number of supported PDOs defined by <code>canOpenCreateNodeEx()</code>.• Map an object into different PDOs without defining the <code>MULTI_MAP</code> flag for this object.• The same object is mapped into more different PDOs than the maximum allowed number configured with <code>canOpenCreateNodeEx()</code>. <p>In other cases contact esd gmbh if it is possible to get a version of the CANopen slave with a greater built in maximum for this object type.</p>
Function	<code>canOpenCreateNode()</code> <code>canOpenCreateNodeEx()</code> <code>canOpenDefinePDO()</code>

SCANOPEN_WRONG_NODESTATE

Wrong nodestate for this operation.

Severity	Warning / Error
Description	A requested operation could not be performed because the CANopen slave is not in the correct nodestate.
Solutions	<ul style="list-style-type: none">• If this happens during initialization make sure that the CANopen slave is not already started.• If this happens for an operation that should cause a data transmission this is a warning that the transmission was not performed because of the wrong node state.
Function	All functions

SCANOPEN_SERVICE_NOT_ALLOWED

Requested operation aborted.

Severity	Error
Description	A requested operation was not completed because of internal reasons
Solutions	<ul style="list-style-type: none"> • If you want to delete a network make sure that all nodes that belong to this network haven been deleted previously.. • If you want to write/read a PDO check that the PDO type that belongs to this handle matches the operation.
Function	canOpenRemoveNetwork() canOpenWritePDO() canOpenReadPDO() canOpenRequestPDO

SCANOPEN_LENGTH_MISMATCH

PDO length error.

Severity	Error
Description	The length of a received PDO does not match the PDO definition.
Solutions	Make sure that the configuration of the PDO transmitter matches the receiver configuration. Use dummy mapping for PDO bytes that your application is not interested in.
Function	canOpenReadPDO() canOpenRequestPDO()

SCANOPEN_INIT_ERRORS

Error during initialization.

Severity	Error
Description	An initialization operation could not be completed because of a CAN driver error.
Solutions	Make sure that the CAN driver for the network that is used from the CANopen slave is installed and initialized correctly.
Function	canOpenCreateNwtwork() canOpenCreateNode()

SCANOPEN_INVALID_HANDLE

Function call with invalid handle

Severity	Error
Description	An operation could not be completed because the given handle is invalid.
Solutions	<ul style="list-style-type: none">• Check if a variable to store a CANopen handle is used for other things during operation.• Check that after a canOpenDeleteNode() call the node handle is no longer used for further calls.
Function	All functions using a handle as parameter

SCANOPEN_ACCESS_ERROR

Operation failed because of access rights.

Severity	Error
Description	The operation could not be performed because the referenced object in the object dictionary has the wrong access rights.
Solutions	The referenced object exist but the access rights are incorrect for this operation. If you want e.g. writing to an object dictionary entry that is marked as “read only” you will get this error.
Function	canOpenWriteDictionary() canOpenReadDictionary()

SCANOPEN_PDO_PARAMETER_ERROR

Invalid communication parameter.

Severity	Error
Description	The operation could not be performed because at least one PDO communication parameter is invalid.
Solutions	Check communication parameter.
Function	canDefinePDO()

SCANOPEN_NOT_IMPLEMENTED

The functionality isn't implemented.

Severity	Error
Description	The operation could not be performed because this feature isn't implemented in this version of the CANopen slave library.
Solutions	Contact esd GmbH.
Function	N/A

SCANOPEN_INHIBITED

PDO inhibit time not reached.

Severity	Error
Description	A PDO can not be send because the configured inhibit time isn't exceeded since the last transmission.
Solutions	Try to repeat the failed operation later.
Function	canOpenWriteDictionary() canOpenWritePDO()