CANopen Slave

- Preliminary -
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
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.

<table>
<thead>
<tr>
<th>Alterations in this manual versus previous version</th>
<th>Alterations in software</th>
<th>Alterations in documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation of macros used in the library.</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Updated documentation of return values.</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Document layout revised.</td>
<td>-</td>
<td>no changes in respect of content</td>
</tr>
</tbody>
</table>
1. Reference ................................................................. 6

2. Introduction .............................................................. 6

3. CANopen Slave ............................................................ 7
   3.1 Mode of Operation .................................................... 7
   3.2 State Transition ..................................................... 10
   3.3 Node-/Lifeguarding .................................................. 10

4. Program Interface ........................................................ 11
   4.1 Management Services ................................................ 11
      canOpenCreateNetwork() ........................................... 11
      canOpenRemoveNetwork() .......................................... 11
      canOpenCreateNode() ................................................. 12
      canOpenCreateNodeEx() ............................................ 14
      canOpenDeleteNode() ................................................ 19
      canOpenActivateNode() ............................................. 19
      canOpenGetNodeInfo() .............................................. 19
      canOpenResetNode() ................................................. 20
      canOpenWaitForNodeState() ....................................... 20
   4.2 Local Object Directory ............................................. 22
      canOpenExtendDictionary() ....................................... 22
      canOpenInitDictionary() .......................................... 23
      canOpenReadDictionary() ......................................... 24
      canOpenWriteDictionary() ........................................ 25
      canOpenGetDictionaryHnd() ...................................... 26
      canOpenReadDictionaryHnd() .................................... 27
      canOpenWriteDictionaryHnd() .................................... 27
   4.3 PDO Services ............................................................ 28
      canOpenDefinePDO() ................................................. 28
      canOpenWritePDO() ................................................ 31
      canOpenReadPDO() .................................................. 32
      canOpenRequestPDO() .............................................. 32
   4.4 Error Situations and Emergency Objects .......................... 33
      canOpenSetError() .................................................. 34
      canOpenResetError() ............................................... 35
   4.5 Assistant Functions ................................................ 36
      canOpenGetVersions() .............................................. 36
   4.6 Event handler ......................................................... 37
      Object Eventhandler ............................................... 37
   4.7 Macros ............................................................... 40
      Dictionary Entry Tables .......................................... 40
      PDO Mapping Tables .............................................. 41
      PDO Tables ........................................................ 42

5. Error Codes of Slave-Service Functions ............................ 45
1. Reference

/1/: electronic system design gmbh, CAN-API for CAL/CANopen, December 1996

/2/: CiA DS-301 V 4.02, CANopen-Application layer and communication profile, February 2002

/3/: CiA DS-401, V 3.0, Device Profile for I/O-Modules, October 2006

2. Introduction

To be revised.
3. CANopen Slave

By means of the module library it is possible to manage up to 16 physical CAN nets with a maximum of 255 independent slave nodes.

The program interface of the CANopen-slave library is a procedural API which is declared in the header file scanopen.h.

Depending on the operating system, the module library either has to be linked statically to the application or is available as shared library (see appendix).

Following explanations about the mode of operation are not intended to be an introduction into CANopen, but assume that the contents of /2/ is known. The explanations only serve as a description of the fundamental mode of operation of the CANopen slave and an explanation of terms in the description of the program interface in the following chapter.

3.1 Mode of Operation

The application generates a CANopen slave with a module number which has to be unique in the physical CAN network. Each slave is assigned an object directory and at least one service data object (SDO) and one emergency object (EMCY) whose defaults COB-IDs are derived from the module number. The application can extend the object directory with manufacturer specific entries or according to standardized device profiles (/3/) and map its process variables into the object directory as shown in figure 2. The entries of the object directory can be mapped into process data objects (PDO) which are transmitted or received using the CAN bus.

![Fig. 1: Operation mode of CANopen slave](image-url)
The COB identifiers of the PDOs, the PDO mapping and a number of further parameters can be configured by the application as well as by a CANopen manager.

The downstream communication between the application and the CANopen library relies on a procedural interface, the upstream communication from the CANopen library to the application is event driven which is realized using callback handlers.

Following steps are required for creating a virtual slave node and make this node start communicating on the CAN:

1. Initializing the CAN bus and starting the NMT daemon calling `canOpenCreateNetwork`.

2. Initializing the virtual slave, assigning the node event handler and defining the supported entries of the object directory in the Communication Profile Area calling `canOpenCreateNodeEx`.

3. Creating additional entries in the Manufacturer Specific Area and the Standardized Device Profile Area of the object directory calling `canOpenExtendDictionary`. Initializing these entries and assigning the object event handler calling `canOpenInitDictionary`.

4. Creating and initializing the PDOs calling `canOpenDefinePDO`.

5. Changing node state to Pre-Operational 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:
A overview for the sequence of API calls can be taken from the flow chart above. A detailed description of the API is described in chapter 4.
3.2 State Transition

To be reviewed.

3.3 Node-/Lifeguarding

To be reviewed.
4. Program Interface

The following chapter describes the program 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

**Synopsis:**

```c
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. `NetNo` for the first CAN interface is 0, for the second 1, etc. Optionally a pointer to a textual description can be given. `Baudrate` is specified in kbit/s. The baudrate support depends on CAN-layer-2 driver.

**Return:**

0 or an error code described in the appendix.

---

**canOpenRemoveNetwork()**

**Name:** canOpenRemoveNetwork() - removing a network

**Synopsis:**

```c
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:**  
```c
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 /2/ 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 /2/.

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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTENDED_BOOTUP</td>
<td>support of the CAL bootup</td>
</tr>
<tr>
<td>State_REGISTER</td>
<td>generate object entry 0x1002</td>
</tr>
<tr>
<td>ERROR_REGISTER</td>
<td>generate object entry 0x1003</td>
</tr>
<tr>
<td>ADDITIONAL_PDOS</td>
<td>generate object entry 0x1004</td>
</tr>
<tr>
<td>SYNCHRON_PDOS</td>
<td>generate object entries 0x1005-0x1007</td>
</tr>
<tr>
<td>MANUFACTURER_INFO</td>
<td>generate object entries 0x1008-0x100A</td>
</tr>
</tbody>
</table>
If **EXTENDED_BOOTUP** is set in options, the CANopen node does the CAL bootup if it receives a **Disconnect Remote Node** command by the NMT master, otherwise the CANopen node performs like a **Minimum Capability Device**.

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 clients\(^1\) can be determined in **ServerObjects** and **ClientObjects**. The default SDO server has to be included in **ServerObjects**.

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 4.5.

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.

---

\(^{1}\) In the current version of the slaves it is not possible to generate additional SDO servers and SDO clients.
canOpenCreateNodeEx()

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

**Synopsis:**
```c
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 */
    HNDO * 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 /2/ 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 4.5.

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 /2/ 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 /2/.

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Flag in ulOptions</th>
<th>Member to initialize</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>Device Type</td>
<td>Created implicitly</td>
<td>ulDeviceType</td>
</tr>
<tr>
<td>0x1001</td>
<td>Error Register</td>
<td>Created implicitly</td>
<td>-</td>
</tr>
<tr>
<td>0x1002</td>
<td>Manufacturer Status</td>
<td>STATE_REGISTER</td>
<td>-</td>
</tr>
<tr>
<td>0x1003</td>
<td>Pre-defined error field</td>
<td>ERROR_REGISTER</td>
<td>ucMaxErrors</td>
</tr>
<tr>
<td>0x1005</td>
<td>COB-ID SYNC, Comm. cycle period,</td>
<td>SYNCHRON_PDOS</td>
<td>ulSyncCobID</td>
</tr>
<tr>
<td>0x1007</td>
<td>Sync. Window length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
<td>Structure/Function</td>
<td>Parameters</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x1008 to 0x100A</td>
<td>Manufacturer device name, HW version and SW version</td>
<td>MANUFACTURER_INFO</td>
<td>pszDevicename, pszHwVersion, pszSwVersion</td>
</tr>
<tr>
<td>0x100C/0x100D</td>
<td>Guard time and Lifetime factor</td>
<td>GUARDING</td>
<td>usGuardTime, ucLifeTime</td>
</tr>
<tr>
<td>0x1010</td>
<td>Store parameters</td>
<td>PARAMETER_STORE</td>
<td></td>
</tr>
<tr>
<td>0x1011</td>
<td>Restore defaults</td>
<td>PARAMETER_RESET</td>
<td></td>
</tr>
<tr>
<td>0x1014</td>
<td>COB-ID EMCY</td>
<td>Created implicitly</td>
<td>ulEmcyCobId</td>
</tr>
<tr>
<td>0x1015</td>
<td>Inhibit time EMCY</td>
<td>Created implicitly</td>
<td>usEmcyInhibit</td>
</tr>
<tr>
<td>0x1016</td>
<td>Consumer Heartbeat Time</td>
<td>CONSUMER_HEARTBEAT</td>
<td>ucMaxConsumerHB, pulListCHBT</td>
</tr>
<tr>
<td>0x1017</td>
<td>Producer Heartbeat Time</td>
<td>PRODUCER_HEARTBEAT</td>
<td>usProducerHBTime</td>
</tr>
<tr>
<td>0x1018</td>
<td>Identity Object</td>
<td>Created implicitly</td>
<td>ucMaxIdentityObject, ulVendorId, ulProductCode, ulRevisionNumber, ulSerialNumber</td>
</tr>
<tr>
<td>0x1020</td>
<td>Verify Configuration</td>
<td>PARAMETER_STORE</td>
<td></td>
</tr>
<tr>
<td>0x1028</td>
<td>Emergency consumer</td>
<td>EMCY_CONSUMER</td>
<td></td>
</tr>
<tr>
<td>0x1029</td>
<td>Error behaviour</td>
<td>ERROR_BEHAVIOUR_OBJECT</td>
<td>ucErrorBehaviour</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>ADDITIONAL_SDOS</td>
<td>ucServerSDO, ucClientSDO</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Structure/Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>usRxPDO</td>
<td>Mandatory</td>
<td>Defines the maximum number of Rx-PDOs of this node.</td>
</tr>
<tr>
<td></td>
<td>usTxPDO</td>
<td>Mandatory</td>
<td>Defines the maximum number of Tx-PDOs of this node.</td>
</tr>
<tr>
<td></td>
<td>ucServerSDO</td>
<td>Mandatory if ADDITIONAL_SDOS is set</td>
<td>Defines the maximum number of SDO server. If ADDITIONAL_SDOS isn’t set the default SDO server will be created.</td>
</tr>
<tr>
<td></td>
<td>ucClientSDO</td>
<td>-</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>
### Program Interface

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucMaxErrors</td>
<td>Mandatory if ERROR_REGISTER is set. Defines the maximum number of errors (1-127) that can be stored in the error history.</td>
</tr>
<tr>
<td>ucMaxIdentityObject</td>
<td>Mandatory. Defines the number of subindices (1-4) of entry Identity Object (0x1018).</td>
</tr>
<tr>
<td>ulVendorId</td>
<td>Mandatory. CiA registered vendor id for this device.</td>
</tr>
<tr>
<td>ulProductCode</td>
<td>Mandatory if ucMaxIdentityObject &gt; 1. Vendor specific product code for this device.</td>
</tr>
<tr>
<td>ulRevisionNumber</td>
<td>Mandatory if ucMaxIdentityObject &gt; 2. Vendor specific revision number for this device.</td>
</tr>
<tr>
<td>ulSerialNumber</td>
<td>Mandatory if ucMaxIdentityObject = 4. Vendor specific serial number for this device.</td>
</tr>
<tr>
<td>ucMaxConsumerHB</td>
<td>Mandatory if CONSUMER_HEARTBEAT is set. Number of subentries (1-127) of the Consumer Hearbeat object.</td>
</tr>
<tr>
<td>ulDeviceType</td>
<td>Mandatory. Device type of this device.</td>
</tr>
<tr>
<td>pszDeviceName</td>
<td>Mandatory if MANUFACTURER_INFO is set. NULL terminated string for Manufacturer Info of this device.</td>
</tr>
<tr>
<td>pszHwVersion</td>
<td>Mandatory if MANUFACTURER_INFO is set. NULL terminated string for Manufacturer Hardware Version of this device.</td>
</tr>
<tr>
<td>ulLifetime</td>
<td>Mandatory if GUARDING is set. Default lifetime factor used by this device for life guarding.</td>
</tr>
<tr>
<td>usGuardTime</td>
<td>Mandatory if GUARDING is set. Default guardtime used by this node for life guarding.</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pszSwVersion</td>
<td>Mandatory if MANUFACTURER_INFO is set. NULL terminated string for Manufacturer Software Version of this device.</td>
</tr>
<tr>
<td>ulSyncCobID</td>
<td>Mandatory if SYNCHRON_PDOS is set. Defines the COB-ID of the SYNC object for this device. Initialize to DEFAULT_SYNC_COBID which becomes 0x80 to use the standard /2/ default.</td>
</tr>
<tr>
<td>ulCyclePeriod</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>ulSyncWindowLen</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>ulTimestampCobId</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>ulEmcyCobId</td>
<td>Optional. Defines the COB-ID of the EMCY object. If this is set to 0 or DEFAULT_EMCY_COBID the value becomes 0x80 + iModId is used.</td>
</tr>
<tr>
<td>usEmcyInhibit</td>
<td>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.</td>
</tr>
<tr>
<td>usProducerHbTime</td>
<td>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.</td>
</tr>
<tr>
<td>*pulListCHBT</td>
<td>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 be less than the number of entries given with the parameter ucMaxConsumerHB. Example:</td>
</tr>
<tr>
<td>ucErrorBehaviour</td>
<td>Mandatory if ERROR_BEHAVIOUR_OBJECT is set.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>ucMaxMapped</th>
<th>Conditional for multimap support.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This parameter defines in how many different PDOs the same object dictionary can be mapped if the the object dictionary entry is created supporting this feature. If this parameter is 0 the default value of 8 will be used.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>usPdoRxQueusize</th>
<th>Size of PDO daemon receive queue.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defines the size of the Rx daemon receive queue in multiple of PDO messages. The default value is 256.</td>
<td></td>
</tr>
</tbody>
</table>

**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() - determine node state

Synopsis: int canOpenDeleteNode
(HNODE HNode, /* handle of the CANopen node */
int * State, /* actual state */
int * LastErr /* last error state */)
Description: This call returns the current state of the node designated by \textit{Hnode}.

Valid values for \textit{State} are:

- \texttt{NodeNone}
- \texttt{NodeOffline}
- \texttt{NodeInit}
- \texttt{NodePreOperational}
- \texttt{NodeDisconnected}
- \texttt{NodeConnecting}
- \texttt{NodeConnected}
- \texttt{NodePreparing}
- \texttt{NodePrepared}
- \texttt{NodeOperational}

In \textit{LastError} the error number of the last error is returned.

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

---

\textbf{canOpenResetNode()}

Name: \texttt{canOpenResetNode()} - resetting CANopen node.

Synopsis: \begin{verbatim}
int canOpenDeleteNode(HNODE HNode) /* handle of the CANopen node */
\end{verbatim}

Description: The slave is reset to state \texttt{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.

---

\textbf{canOpenWaitForNodeState()}

Name: \texttt{canOpenWaitForNodeState()} - Block until transition in given node state

Synopsis: \begin{verbatim}
int canOpenWaitForNodeState(HNODE HNode, unsigned short StateMask) /* handle of the CANopen node */
\end{verbatim}

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

\textit{StateMask} is the logical OR combination of the following constants describing the node states to wait for. Valid parameters are:
WFNS_INIT    WFNS_DISCONNECTED
WFNS_CONNECTING    WFNS_PREPARING
WFNS_PREPARED    WFNS_OPERATIONAL
WFNS_PRE OPERATIONAL

Return: Current node status or an error code described in the appendix.
4.2 Local Object Directory Services

The functions explained below provide an application-specific configuration of the local object directory and the reading and writing access.

The reading access to the object directory by means of the CANopen master or further CANopen slaves is processed by SDO-server processes in the background.

In writing access the entry is checked on data type consistency and the value ranges before updating the entry. In addition the callback handler of the directory entry is called. The callback handler can optionally be declared when initializing the directory entries. The performance in writing access to the entries of the Communication Profile Area of the slave cannot be influenced by the application.

---

canOpenExtendDictionary()

**Name:** canOpenExtendDictionary() - Extending the local Object Dictionary

**Synopsis:**

```c
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 * Data type /* datatype designator */
)
```

**Description:**

Generates a new entry in the object directory of the CANopen node in the Manufacturer Specific Area or the Standardized Device Profile Area. This function can only be called in node state NodeOffline.

*Index* indicates the index in the object directory in the range of 0x2000 to 0x9FFF and *subentries* indicates the number of subentries to be generated (0-254).

*ObjectType* is OBJ_VAR, OBJ_ARRAY or OBJ_RECORD.

*Data type* is a single descriptor followed by a 0 for entries of types OBJ_VAR and OBJ_ARRAY. Objects of type OBJ_ARRAY store the number of subentries in data format unsigned 8 at subindex 0 of the entry. For entries of type OBJ_RECORD a structure description has to be given. For a data structure of an unsigned long, word and byte, the structure description would have the following form:

```c
const char *MyMapping[] = {TYP_INT32, TYP_INT16, TYP_INT8, 0}
```

**Return:**

0 or an error code described in the appendix.
**canOpenInitDictionary()**

**Name:** canOpenInitDictionary() - initializing local directory entry.

**Synopsis:**
```c
int canOpenInitDictionary(
    HNODE Hnode,   /* handle of the CANopen node */
    unsigned short Index,   /* index in object directory */
    unsigned short Subindex,   /* subindex in object directory */
    const char * EntryName,   /* textual description of the entry */
    unsigned short Flags,   /* properties of entry */
    pDictionaryData Data,   /* default data */
    int (* Handler)(int, int, int, int, void *)  /* callback handler of entry */
)
```

**Description:**
Initializing an entry in the object directory of the CANopen node designated by `Hnode` in the **Manufacturer Specific Area** or the **Standardized Device Profile Area**. This function has to be called for each individual subindex of this object before it is possible to access the entry by means of an SDO server. This function is only to be called in node state **NodeOffline**.

*Index* has to be in the range of 0x2000 to 0x9FFF and *subindex* in the range of 0x00 to 0xFE. Entries of object type **OBJ_ARRAY** automatically initialize the entry at subindex 0. Entries of type **OBJ_VAR** make only possible an initialization of subindex 0.

*EntryName* is an optional textual description of the entry. This description is only important when generating a DCF file. If no description is required, set this parameter to NULL, because this description extends the memory demand for an individual subentry into the object directory remarkably, depending on the length of the text string.

By means of *flags* access rights and properties of the entry are determined. Possible values are **READ_ACCESS** and **WRITE_ACCESS**. By means of **LIMITS_VALID** the limits specified in *data* are declared as being valid and in writing accesses to the object directory keeping these limits is checked at numerical data types. By means of **MAPPABLE** it is determined that this entry can be mapped into a PDO.

*Data* is a pointer to a union of structures of type **DictionaryData**. This union has to be initialized in dependency on the data type. The application is responsible for the data validity concerning the data type.

For numerical data types the structure consists of the current value, the default value and the lower and upper limit of the value range.

For string-data types the structure consists of a pointer to a memory range within the application, the length of this memory range and the length of the current string.
**Handler** is the object event handler of this entry. A detailed explanation about the handlers can be taken from section 4.5.

**Return:**

0 or an error code described in the appendix.

---

### canOpenReadDictionary()

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

**Synopsis:**

```c
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:**
```c
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.

<table>
<thead>
<tr>
<th>CANopen data type</th>
<th>Reference type</th>
<th>Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>Pointer to new data (1 byte)</td>
<td>yes</td>
</tr>
<tr>
<td>Int8, Int16, Int32</td>
<td>Pointer to new data (1 byte, 2 bytes, 4 bytes)</td>
<td>yes</td>
</tr>
<tr>
<td>Uint8, Uint16, Uint32</td>
<td>Pointer to new data (1 byte, 2 bytes, 4 bytes)</td>
<td>yes</td>
</tr>
<tr>
<td>Float</td>
<td>Pointer to new data (4 bytes)</td>
<td>yes</td>
</tr>
<tr>
<td>Visible String</td>
<td>Pointer to structure of RecVisString type</td>
<td>no</td>
</tr>
<tr>
<td>Octet String</td>
<td>Pointer to structure of RecOctString type</td>
<td>no</td>
</tr>
<tr>
<td>Time Of Day</td>
<td>Pointer to structure of CAN_TIME_OF_DAY type</td>
<td>yes</td>
</tr>
<tr>
<td>Time Difference</td>
<td>Pointer to structure of CAN_TIME_DIFFERENCE type</td>
<td>yes</td>
</tr>
<tr>
<td>Domain</td>
<td>Pointer to structure of RecDomain type</td>
<td>no</td>
</tr>
</tbody>
</table>

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:**

```c
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 the 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 according 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 canOpenReadDictionaryHnd(
                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 event handler(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 configurated 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 configurated cycle period by means of calling the object event handlers of the mapped directory entries.

---

canOpenDefinePDO()

**Name:** canOpenDefinePDO() - initializing a PDO

**Synopsis:**

```c
int canOpenDefinePDO
(  
  HNODE HNode, /* handle of the CANopen node */
  const char * Name, /* designator of this PDO */
  unsigned long COBid, /* default-COB identifier of PDO */
  unsigned short TransMode, /* transfer mode of PDO */
  int InhibitTime, /* inhibit time of this PDO */
  unsigned short TxTout, /* transmit timeout of this PDO */
  unsigned short RxTout, /* receive timeout of this PDO */
  int Prio, /* CMS-priority group of this PDO */
  unsigned short * Mapping, /* default mapping of this PDO */
  HPDO hpdo /* PDO handle */
)
```

**Description:** Calling this function creates an additional PDO for the CANopen node. The PDO configuration after bootup or reset is defined by these parameters. The related mandatory node’s object directory entries for PDO Communication Parameters and the PDO Mapping Parameters 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 \textit{COBid} defines the PDO’s default COB-ID which according to /xxx/ 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 \texttt{DEFAULT\_PDO\_N} with \texttt{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 \texttt{PDO\_VALID} or \texttt{PDO\_INVALID} to determine, which PDOs are used in the NMT node state \textbf{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 \texttt{RTR\_ALLOW} or \texttt{RTR\_DISALLOW} define whether a transmit PDO might by RTR requestable or not\(^2\). The configured value of this COB-ID control bit can not be changed by a CANopen manager.

The parameter \textit{TransMode} defines the transmission type of the PDO. In addition to /xxx/ this parameter consists also of several proprietary control bits which describe the type and the behaviour of the PDO. To apply these control bits you have to combine the with the PDO transmission type by a logical OR operation. The PDO type control bit can be either set to \texttt{TRANSMIT\_PDO} or \texttt{RECEIVE\_PDO} with \texttt{SYNCHRON\_PDO}/\texttt{ASYNCHRON\_PDO} and a numerical value between 0 and 255 which is determined in /2/ according to following table. In addition it is possible to mark an asynchronous PDO by \texttt{AUTO\_NOTIFY}. This PDO has the properties described above. In an asynchronous transmit PDO it is possible to determine by means of specifying \texttt{TX\_DONE\_PDO}, whether the CAN-driver function /1/ \texttt{canCalWrite} instead of \texttt{canCalSend} is to be used for doing the transfer.

\(^{2}\) 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.
The transmission of this PDO is triggered by means of a manufacturer-specific event.

The transmission of this PDO is triggered by means of a device-specific event.
A value n between 1 and 240 describes a cyclical, synchronous transfer PDO which is only transmitted at the reception of every nth SYNC object.

InhibitTime determines in ms how long after transmitting a PDO this is not be transmitted again.

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

By means of Prio the desired CMS-priority class (0-7) for identifier distribution via DBT can be determined.

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 /2/ 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 call back 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 call back 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 Objects

Error states of a CANopen node are indicated by transmitting an *Emergency Object* whose COB identifier results from the module number according to /2/. The *Emergency Object* is available directly after initializing the CANopen node and consists of an *Emergency Error Code* whose meaning is determined in /2/, the contents of the *error register* and a *Manufacturer Specific Error Field*. The transitions between error-free state and an error situation can be taken from following figure:

![Situation Diagram for Error Condition](image)

*(0)* After the initialization the CANopen node gets into error-free state.

*(1)* If a `canOpenSetError()` is called, the CANopen node changes into error state. An EMCY object is given out, the error is marked by an according flag in the *error register* (0x1001) and the error cause is hold in the *pre-defined error field* (0x1003), if this optional entry is supported by the CANopen node. The internal counter for the node errors is incremented.

*(2)* If a `canOpenSetError()` is called again, the actions listed under (1) are executed and the CANopen node remains in error state.

*(3)* If a `canOpenResetError()` is called, an EMCY object is transmitted with a *reset error* message according to /2/ and the internal counter for node errors is decremented. If this counter is not zero, the node remains in error state.

*(4)* If the internal error counter is zero after calling `canOpenResetError()` and following decrementation, the node changes into error-free state again.
canOpenSetError()

**Name:**
canOpenSetError() - setting an error

**Synopsis:**
```c
int canOpenSetError(
    HNODE Hnode, /* handle of the CANopen node */
    unsigned short ErrorCode, /* error code according to /2/ */
    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_OUTPUT
- EMCY_VOLTAGE
- EMCY_VOLTAGE_INPUT
- EMCY_VOLTAGE_OUTPUT
- EMCY_TEMPERATURE
- EMCY_TEMPERATURE_AMBIENT
- EMCY_VOLTAGE_INSIDE
- EMCY_VOLTAGE_OUTPUT
- EMCY_TEMPERATURE
- EMCY_DEVICE_HARDWARE
- EMCY_DEVICE_SOFTWARE
- EMCY_MONITORING
- EMCY_EXTERNAL_ERROR
- EMCY_ADDITIONAL_MODULES
- 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_VOLTAGE
- ERROR_COMMUNICATION
- ERROR_MANUFACTURER_SPECIFIC
- ERROR_CURRENT
- ERROR_TEMPERATURE
- ERROR_DEVICE_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 reset. 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() - Determine software version numbers.

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

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

The address of a structure with the following declaration is transferred:

```c
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;
    } VERSIONS;
```

The structure of the revision number is identical for the eight shown software modules with the following format:

<table>
<thead>
<tr>
<th>Bits 15…12</th>
<th>Bits 11…8</th>
<th>Bits 7…0</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>revision</td>
<td>change</td>
</tr>
</tbody>
</table>

Since the version of the CAN-interface driver is determined by means of canCalGetVersion() (see /1/), the cell of can has to be called with the net number for which the CANopen slave is initialized.

**Return:** N/A.
4.6 Event handler

The basis for the event driven interaction between CANopen slave and the application is constituted by the event handlers which are realized as call back functions, here. Each node has an event handler for node-specific events or error situations. This event handler has to inform the application and an event handler can be linked to each entry in the object directory. At the reception of data this event handler is executed.

```
Every event handler is called directly from within the threads/processes of the CANopen slave and therefore must be reentrant, should last a short runtime and must not block the application.
```

Object Eventhandler

If an entry in the object directory is mapped into a receive PDO, and if data are received for this PDO, the configured object event handler is called. Five arguments are given to the event handler:

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

Within the application it is possible to use the same event handler for all nets, nodes and directory entries and decide from the first 4 parameters, which operation is to be executed.

Node Event handler

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.

<table>
<thead>
<tr>
<th>ulEvent</th>
<th>ulArg1</th>
<th>ulArg2/pArg2</th>
<th>Event reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV_GUARDING EV_START</td>
<td>---</td>
<td></td>
<td>Node/Lifeguarding is started</td>
</tr>
<tr>
<td>EV_GUARDING EV_TIMEOUT</td>
<td>---</td>
<td></td>
<td>Lifeguarding timed out</td>
</tr>
<tr>
<td>EV_GUARDING EV_STOP</td>
<td>---</td>
<td></td>
<td>Node/Lifeguarding is stopped</td>
</tr>
<tr>
<td>EV_STATE_CHANGE state</td>
<td>---</td>
<td></td>
<td>node has changed into state</td>
</tr>
<tr>
<td>EV_RESET EV_COMMUNICATION EV_BEGIN</td>
<td></td>
<td>Enter Reset Communication state</td>
<td></td>
</tr>
<tr>
<td>EV_RESET EV_COMMUNICATION EV_END</td>
<td></td>
<td>Leave Reset Communication state</td>
<td></td>
</tr>
<tr>
<td>EV_RESET EV_APPLICATION EV_BEGIN</td>
<td></td>
<td>Enter Reset Application started state</td>
<td></td>
</tr>
<tr>
<td>EV_RESET EV_APPLICATION EV_BEGIN</td>
<td></td>
<td>Leave Reset Application state</td>
<td></td>
</tr>
<tr>
<td>EV_APPLICATION EV_BEGIN</td>
<td></td>
<td>Store configuration request</td>
<td></td>
</tr>
<tr>
<td>EV_APPLICATION EV_BEGIN</td>
<td></td>
<td>Restore configuration request</td>
<td></td>
</tr>
<tr>
<td>EV_CONFIGURATION EV_STORE</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>EV_CONFIGURATION EV_RESTORE</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>EV_CAN EV_CONTROLLER_OK</td>
<td></td>
<td>Recovery from CAN bus-off state</td>
<td></td>
</tr>
<tr>
<td>EV_CAN EV_CONTROLLER_WARN</td>
<td></td>
<td>CAN Controller enters error passive state</td>
<td></td>
</tr>
<tr>
<td>EV_CAN EV_CONTROLLER_BUS_OFF</td>
<td></td>
<td>CAN Controller enters bus-off state</td>
<td></td>
</tr>
<tr>
<td>EV_CAN EV_FIFO_OVERRUN</td>
<td></td>
<td>CAN controller overrun error</td>
<td></td>
</tr>
<tr>
<td>EV_CAN EV_PDO_FIFO_OVERRUN</td>
<td></td>
<td>Number of lost PDOs</td>
<td></td>
</tr>
<tr>
<td>EV_CAN EV_PDO_FIFO_OVERRUN</td>
<td></td>
<td>Receive FIFO of PDO daemon is overrun</td>
<td></td>
</tr>
</tbody>
</table>
### Deprecated event handler

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

```c
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.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV_PDO_RX_ERROR</td>
<td>CAN driver error code</td>
</tr>
<tr>
<td>EV_PDO_NOT_PROCESSED</td>
<td>PDO number</td>
</tr>
<tr>
<td>EV_PDO_LENGTH_EXCEEDED</td>
<td>PDO number</td>
</tr>
<tr>
<td>EV_EMCY</td>
<td>Node-ID of the EMCY producer</td>
</tr>
<tr>
<td>EV_CONSUMER_HEARTBEAT</td>
<td>EV_START, EV_STOP</td>
</tr>
<tr>
<td>EV_WRITE_DICTIONARY</td>
<td>Index, Subindex</td>
</tr>
<tr>
<td></td>
<td>The receive request of the PDO daemon returned with an unexpected error.</td>
</tr>
<tr>
<td></td>
<td>A received PDO isn’t processed because the length of the received PDO is smaller than the length according to the current mapping.</td>
</tr>
<tr>
<td></td>
<td>A received PDO isn’t processed because the length of the received PDO exceeds the length according to the current mapping for this PDO.</td>
</tr>
<tr>
<td></td>
<td>EMCY object received</td>
</tr>
<tr>
<td></td>
<td>Heartbeat monitoring started</td>
</tr>
<tr>
<td></td>
<td>Heartbeat monitoring stopped</td>
</tr>
<tr>
<td></td>
<td>Write access to object dictionary entry</td>
</tr>
</tbody>
</table>

The receive request of the PDO daemon returned with an unexpected error.

A received PDO isn’t processed because the length of the received PDO is smaller than the length according to the current mapping.

A received PDO isn’t processed because the length of the received PDO exceeds the length according to the current mapping for this PDO.

Node-ID of the EMCY producer

EMCY object received

Heartbeat monitoring started

Heartbeat monitoring stopped

Write access to object dictionary entry
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()`.

```
BEGIN_DICTIONARY_TABLE(DictionaryName)

Begins the definition of a dictionary table. 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.

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:

```c
#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.

```c
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 won’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).

<table>
<thead>
<tr>
<th>Severity</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The operation was executed without any errors.</td>
</tr>
<tr>
<td>Function</td>
<td>All functions.</td>
</tr>
</tbody>
</table>

**SCANOPEN_WRONG_INDEX**
The parameter *index* is invalid.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The object entry that should be referenced by the parameter <em>index</em> does not exist.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Create an entry in the object dictionary with this index before you reference it.</td>
</tr>
<tr>
<td>Function</td>
<td>canOpenInitDictionary()</td>
</tr>
<tr>
<td></td>
<td>canOpenReadDictionary()</td>
</tr>
<tr>
<td></td>
<td>canOpenWriteDictionary()</td>
</tr>
</tbody>
</table>
## SCANOPEN_WRONG_SUBINDEX
The parameter `subindex` is invalid.

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

## SCANOPEN_OUT_OF_MEMORY
Error allocating a resource.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Allocating a resource like memory or a synchronization object that is necessary to complete the operation failed.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Increase the available memory for the CANopen slave process.</td>
</tr>
</tbody>
</table>
| Function   | canOpenActivateNode()  
canOpenCreateNetwork()  
canOpenCreateNode()  
canOpenDefinePDO()  
canOpenExtendDictionary() |

## SCANOPEN_WRONG_BAUDRATE
An unsupported CAN baudrate was used.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The CANopen slave should be initialized with a CAN baudrate that is unsupported by this implementation.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Use a supported baudrate.</td>
</tr>
<tr>
<td>Function</td>
<td>canOpenCreateNetwork()</td>
</tr>
</tbody>
</table>
SCANOPEN_CANNOT_START_DAEMON
Error creating an internal thread.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>During CANopen slave initialization a necessary internal CANopen protocol thread could not be started.</td>
</tr>
</tbody>
</table>
| Solutions | • 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.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>One or more parameter of a function call were invalid.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Compare parameter value with ranges given in manual</td>
</tr>
<tr>
<td>Function</td>
<td>All functions</td>
</tr>
</tbody>
</table>

SCANOPEN_VALUE_TOO_HIGH
Parameter value exceeds maximum.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A dictionary object value exceeds the given maximum for this entry.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Compare value with defined maximum of this entry</td>
</tr>
</tbody>
</table>
| Function    | canOpenWriteDictionary()  
            canOpenWriteDictionaryHnd() |
### SCANOPEN_VALUE_TOO_LOW
Parameter value below minimum.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>A dictionary object value is below the given minimum for this entry.</td>
</tr>
<tr>
<td><strong>Solutions</strong></td>
<td>Compare value with defined minimum of this entry</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>canOpenWriteDictionary() canOpenWriteDictionaryHnd()</td>
</tr>
</tbody>
</table>

### SCANOPEN_WRONG_TYPE
Wrong data type.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
| **Solutions** | • 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.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The object type is not supported by the CANopen slave.</td>
</tr>
<tr>
<td><strong>Solutions</strong></td>
<td>Use supported object types listed in manual.</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>canOpenExtendDictionary()</td>
</tr>
</tbody>
</table>
### SCANOPEN_PDO_MAPPING_ERROR

An error occurred during PDO mapping.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
| **Solutions** | • 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.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>During initialization the built in maximum for a certain internal object type like number of CANopen nodes is exceeded</td>
</tr>
<tr>
<td>Solutions</td>
<td>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.</td>
</tr>
<tr>
<td>Function</td>
<td>canOpenCreateNode()</td>
</tr>
<tr>
<td></td>
<td>canOpenDefinePDO()</td>
</tr>
<tr>
<td></td>
<td>canOpenCreateServerSDO()</td>
</tr>
</tbody>
</table>

### SCANOPEN_WRONG_NODESTATE
Wrong nodestate for this operation.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Warning / Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A requested operation could not be performed because the CANopen slave is not in the correct node state.</td>
</tr>
<tr>
<td>Solutions</td>
<td>• If this happens during initialization make sure that the CANopen slave is not already started.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td>Function</td>
<td>All functions</td>
</tr>
</tbody>
</table>

### SCANOPEN_SERVICE_NOT_ALLOWED
Requested operation aborted.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A requested operation was not completed because of internal reasons</td>
</tr>
<tr>
<td>Solutions</td>
<td>• If you want to delete a network make sure that all nodes that belong to this network haven been deleted previously.</td>
</tr>
<tr>
<td></td>
<td>• If you want to write/read a PDO check that the PDO type that belongs to this handle matches the operation.</td>
</tr>
<tr>
<td>Function</td>
<td>canOpenRemoveNetwork()</td>
</tr>
<tr>
<td></td>
<td>canOpenWritePDO()</td>
</tr>
<tr>
<td></td>
<td>canOpenReadPDO()</td>
</tr>
<tr>
<td></td>
<td>canOpenRequestPDO</td>
</tr>
</tbody>
</table>
SCANOPEN_LENGTH_MISMATCH
PDO length error.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The length of a received PDO does not match the PDO definition.</td>
</tr>
<tr>
<td>Solutions</td>
<td>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.</td>
</tr>
<tr>
<td>Function</td>
<td>canOpenReadPDO() canOpenRequestPDO()</td>
</tr>
</tbody>
</table>

SCANOPEN_INIT_ERRORS
Error during initialization.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>An initialization operation could not be completed because of a CAN driver error.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Make sure that the CAN driver for the network that is used from the CANopen slave is installed and initialized correctly.</td>
</tr>
<tr>
<td>Function</td>
<td>canOpenCreateNetwork() canOpenCreateNode()</td>
</tr>
</tbody>
</table>
### SCANOPEN_INVALID_HANDLE
Function call with invalid handle

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>An operation could not be completed because the given handle is invalid.</td>
</tr>
</tbody>
</table>
| Solutions | • 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.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The operation could not be performed because the referenced object in the object dictionary has the wrong access rights.</td>
</tr>
<tr>
<td>Solutions</td>
<td>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.</td>
</tr>
</tbody>
</table>
| Function | canOpenWriteDictionary()  
canOpenReadDictionary() |

### SCANOPEN_PDO_PARAMETER_ERROR
Invalid communication parameter.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The operation could not be performed because at least one PDO communication parameter is invalid.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Check communication parameter.</td>
</tr>
<tr>
<td>Function</td>
<td>canDefinePDO()</td>
</tr>
</tbody>
</table>
SCANOPEN_NOT_IMPLEMENTED
The functionality isn’t implemented.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The operation could not be performed because this feature isn’t implemented in this version of the CANopen slave library.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Contact esd GmbH.</td>
</tr>
<tr>
<td>Function</td>
<td>N/A</td>
</tr>
</tbody>
</table>

SCANOPEN_INHIBITED
PDO inhibit time not reached.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A PDO can not be send because the configured inhibit time isn’t exceeded since the last transmission.</td>
</tr>
<tr>
<td>Solutions</td>
<td>Try to repeat the failed operation later.</td>
</tr>
<tr>
<td>Function</td>
<td>canOpenWriteDictionary() canOpenWritePDO()</td>
</tr>
</tbody>
</table>